

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

Interfacing System Description Languages to Formal Verification

by

Wendell Craig Baker

B.S. (University of California at Berkeley) 1987

M.S. (University of California at Berkeley) 1992

A dissertation submitted in partial satisfaction of the requirements for the

degree of

Doctor of Philosophy

in

Engineering-Electrical Engineering and Computer Science

in the

GRADUATE DIVISION

of the

UNIVERSITY of CALIFORNIA at BERKELEY

Committee in charge:

Professor A. Richard Newton, Chair

Professor Robert K. Brayton

Professor Karlene H. Roberts

1996

UMI Number: 9703053

Copyright 1996 by
Baker, Wendell Craig

All rights reserved.

UMI Microform 9703053
Copyright 1996, by UMI Company. All rights reserved.

**This microform edition is protected against unauthorized
copying under Title 17, United States Code.**

UMI
300 North Zeeb Road
Ann Arbor, MI 48103

The dissertation of Wendell Craig Baker is approved:

Arthur Richard Newton 16 January 96
Chair Date

R. K. Brayton 9 Feb 96
Date

Carlene Roberts 13 February 96
Date

University of California at Berkeley

1996

Interfacing System Description Languages to Formal Verification

Copyright © 1996

by

Wendell Craig Baker

All rights reserved

Table of Contents

Chapter 1. Introduction	1
1.1 Verification Problems	3
1.1.1 Approaches to Design Verification	4
1.1.2 The State Explosion Problem	6
1.2 System Descriptions and Formal Methods	9
1.2.1 The Significance of Semantic Models	10
1.2.2 Languages, Semantics and Models	11
1.3 Transition Semantics of Finite Models	14
1.3.1 Full Abstraction and Discrete Time	15
1.3.2 Non-Abstractness and δ -Time	17
1.3.3 The Substitutability Condition	17
1.4 Limits on Semantic Models	21
1.4.1 Properties of Semantic Models	22
1.4.2 The RMC Barrier	24
1.4.3 Microsemantics	25
1.4.4 Beyond the RMC Barrier	26
1.5 Analysis and Design of Semantics, Languages and Models	27
1.5.1 Applied Semantics	27
1.5.2 System Description Languages	28
1.6 The Nondeterministic Abstract Machine	30
1.7 Review	31
Chapter 2. Semantics and Models	33
2.1 Semantic Specifications	35
2.1.1 Axiomatic Semantics	35
2.1.2 Denotational Semantics	37
2.1.3 Operational Semantics	43
2.1.4 Focus	46
2.2 The Standard Semantic Models	49
2.2.1 Trace Theory	50
2.2.2 Process Algebras	53

2.2.3	The Petri Net as a Model	58
2.2.4	The Kripke Structure	61
2.2.5.	The ω -Automata	69
2.2.6	Denotational Models	74
2.2.7	Focus	77
2.3	Some Non-Standard Models	77
2.3.1	Non-Deterministic Event Sequences (NES)	78
2.3.2	The 2-adic Integers (${}_2Z$)	84
2.3.3	Focus	89
2.4	Review	89
Chapter 3. Computational Semantics		93
3.1	Semantic Domain Theory	97
3.1.1	Primitive Domains	97
3.1.2	Functions on Domains	102
3.1.3	Fixed Points of Functions	103
3.1.4	Functions as Fixed Points	105
3.1.5	Constructed Domains	106
3.1.6	Focus	109
3.2	Behavioral Domains	110
3.2.1	The Behavioral Domain $B = IN \rightarrow (OUT \times B)$	111
3.2.2	The Problems with B	112
3.2.3	Focus	114
3.3	Microsemantic Domains	115
3.3.1	Functional Domains in a Relational World	116
3.3.2	The Relational μ -Calculus	118
3.3.3	The Microsemantic Model	119
3.3.4	Symbolic Representation of Microsemantic Models	120
3.4	Microsemantic Analysis	126
3.4.1	Structure of the Analysis	126
3.4.2	The Fully Abstract Semantics	129
3.4.3	A Non-Abstract δ -Time Semantics	134
3.4.4	A Non-Abstract σ -Time Semantics	149
3.4.5	Focus	162
3.5	Review	163
Chapter 4. Limits on Microsemantics		167
4.1	Orthogonal Aspects of a Semantics	168
4.1.1	Responsiveness (R)	169
4.1.2	Modularity (M)	174
4.1.3	Causality (C)	179
4.2	Theorem of the RMC Barrier	179
4.3	Microsemantics	184
4.3.1	The R , M and C Properties	185
4.3.2	Micro States and Output Variables	186

4.3.3	Microstep Paths	188
4.3.4	Example Microsemantics	191
4.4	Beyond the RMC Barrier	208
4.4.1	The Two-of-Three Choice	209
4.4.2	Separated Semantics	211
4.4.3	Static Restriction to RMC	211
4.4.4	Semantic Restriction to RMC	212
4.4.5	Vacated Semantics	213
4.5	Review	217
Chapter 5. Applied Semantics		219
5.1	Asynchronous Shared Memory (ASM)	220
5.2	Selection/Resolution (S/R)	222
5.3	Combinational/Sequential (C/S)	225
5.4	“Codesign” Finite State Machine (CFSM)	227
5.5	Communicating Sequential Processes (CSP)	230
5.5.1	Theoretical Communicating Sequential Processes (TCSP)	230
5.5.2	Finitely Recursive Processes (FRP)	231
5.5.3	Reduced Communicating Sequential Processes (RCSP)	235
5.6	The Clarke Languages	236
5.7	The Synchronous Languages	239
5.7.1	Defining Attributes	241
5.7.2	The SL Languages	246
5.7.3	Synchronous Programming Languages	257
5.8	Communicating Reactive Processes (CRP)	260
5.9	Review	262
Chapter 6. System Description Languages		265
6.1	An Evolutionary View of Specification Languages	267
6.1.1	Simulation Orientation	268
6.1.2	Synthesis Orientation	270
6.1.3	Verification Orientation	274
6.2	Hardware Description Language Standards	277
6.2.1	Discrete-Event Semantics	278
6.2.2	VHDL and Verilog	279
6.2.3	Event-Driven Semantics	285
6.3	Beyond Discrete-Event Semantics	294
6.3.1	The Annotation Language Approach	294
6.3.2	The Synchronous VHDL Subset	298
6.4	Extending the Standards	316
6.4.1	VHDL ‘93	316
6.4.2	SpecCharts	322
6.5	Review	329

Chapter 7. The Non-Deterministic Abstract Machine	331
7.1 The Abstract Machine	333
7.1.1 Design Constraints	334
7.1.2 Structural Constructors	337
7.1.3 Behavioral Constructors	341
7.1.4 Examples	347
7.2 Causality Checking	348
7.2.1 Problem Definition	350
7.2.2 Known Approaches	352
7.2.3 The Flattened Transition-Graph	360
7.2.4 Checks on the Flattened Estimate Graph	369
7.2.5 Focus	371
7.3 The Semantics of the Machine	371
7.3.1 The Operational Semantics of δ -Time	371
7.3.2 The Relational Semantics of δ -Time	376
7.3.3 Focus	378
7.4 High-Level Language Compilation	379
7.4.1 Esterel	379
7.4.2 Synchronous VHDL	380
7.4.3 (Synchronous) SpecCharts	382
7.5 A Runtime Implementation	384
7.5.1 Code Generation	385
7.5.2 The Runtime System	386
7.5.3 Other Schemes	392
7.6 Review	394
Chapter 8. Conclusion	399
References	407
Appendix A. The Assembly Language of the NDAM	477
A.1 Structure Description	477
A.2 The Process	479
A.3 Data Declarations	481
A.4 Executable Instructions	488
Appendix B. The Flatten Algorithm	507
B.1 BasicBlock::generate	508
B.2 BasicBlock::estimate_TCWC	509
B.3 BasicBlock::estimate_wait_halt	510
B.4 BasicBlock::estimate_misc	511
Appendix C. Compilation of Esterel	513
C.1 An Esterel Example	513
C.2 NDAM Assembly Code	514
C.3 Generated C++ Code	516

Appendix D. The NDAM Runtime System	525
D.1 Data Structures	526
D.2 Support Code	535
D.3 Debugger Support	540

List of Figures

Chapter 1. Introduction	1
Figure 1-1. A Schematic of a Formal Verification System.....	7
Figure 1-2. A Language, its Semantics and the Model	12
Figure 1-3. The One-Level Time of a Fully Abstract Semantics	16
Figure 1-4. The Two-Level Time of a Non-Abstract Semantics	18
Figure 1-5. The Forward and Backward Simulation Operations.....	18
Figure 1-6. The Forward and Backward Image Computations	20
Figure 1-7. A Language, its Fully Abstract and Its Non-Abstract Semantics ...	21
Figure 1-8. The Origins of Structural Properties in Semantics Models	24
Chapter 2. Semantics and Models	33
Figure 2-1. The Three-Phase Construction of a Language and its Semantics...	35
Figure 2-2. The Two-Level Approach with Prefix Operator Terms	43
Figure 2-3. Branching-Time Structures Distinguished by Their Failure Sets ...	54
Figure 2-4. The Syntax of a Simple Process Algebra.....	55
Figure 2-5. The Operational Semantics of a Simple Process Algebra	56
Figure 2-6. The Unraveling of M_K into the Infinite Tree M_{S2S}	66
Figure 2-7. Syntax of CTL and CTL*	68
Figure 2-8. The Semantics of CTL and CTL*.....	69
Figure 2-9. The Branching-Time Semantics in SMV.....	70
Figure 2-10. A Finite State Machine in the UDL/I Language.....	79
Figure 2-11. The Structure of the Semantics of UDL/I.....	80
Figure 2-12. A Partial Order on NES Events Defines a Trace Set	81
Figure 2-13. The Execution of an NES Machine	83
Figure 2-14. A Matrix Multiplier in the 2Z Language	85
Figure 2-15. The Structure of the Semantics of 2Z	86

Chapter 3. Computational Semantics	93
Figure 3-1. A Language, its Abstract Semantics and the Model	94
Figure 3-2. A Language, its Non-Abstract Semantics and the Model.....	95
Figure 3-3. Examples of Flat Domains on Countable Sets.....	100
Figure 3-4. The Domain on the Power Set of $S = \{ a, b, c, d \}$	101
Figure 3-5. Some of the Many Variants of the Binary Decision Diagram	123
Figure 3-6. The Asymptotic Cost of OBDD-Based Computations	124
Figure 3-7. The Eight Steps of Microsemantic Analysis.....	127
Figure 3-8. The Single Level of a Fully Abstract Time.....	131
Figure 3-9. Communication in a Fully Abstract Semantics	133
Figure 3-10. The Two Levels of a Non-Abstract -Time	135
Figure 3-11. The Number of δ -Steps is State- and Input-Dependent.....	135
Figure 3-12. The Primitive Domain of a Single Output Variable.....	138
Figure 3-13. The Constructed Domain of an Output Variable Pair	138
Figure 3-14. The Forward and Backward Trajectories in the Domain M	139
Figure 3-15. Absorbing End Conditions for Forward and Backward δ -Paths..	141
Figure 3-16. Communication in a Non-Abstract σ -Time Semantics.....	150
Figure 3-17. The Two Levels of a Non-Abstract σ -Time.....	151
Figure 3-18. The Number of σ -Steps Are Defined by the Network Structure ..	151
Figure 3-19. The Lattice Structure of S_T	154
Figure 3-20. An Example of the Primitive Domain S_T	155
Figure 3-21. The Utility of \perp and \top in the Domain S_T	156
Chapter 4. Limits on Microsemantics	167
Figure 4-1. The Origins of Structural Properties in Semantics Models	167
Figure 4-2. The Responsive State Transition Structure $R(i,o)$	172
Figure 4-3. The Communication Structure of a Concurrent Composition	176
Figure 4-4. The Communication Structure of $T_1 \parallel T_2$	181
Figure 4-5. An Output's Status and Value are Independent	190
Figure 4-6. The Microstructure of Time for Example 4.3.4.1	192
Figure 4-7. Concurrent Composition for Semantics of Example 4.3.4.2	194
Figure 4-8. Systems T_α and T_β Showing the \bar{M} of Example 4.3.4.2	197
Figure 4-9. A Semantics of Microsteps defining Macrosteps	202
Figure 4-10. Two Microstep Traces for Composition in Example 4.3.4.5	203
Figure 4-11. The Three-Level Model of Time for Example 4.3.4.6.....	206
Figure 4-12. BLIF-MV Tables for the Common Gates	215
Figure 4-13. A Non-Causal Circuit Under a Vacated Semantics	216
Chapter 5. Applied Semantics	219
Figure 5-1. The Communication Model of Asynchronous Shared Memory...220	

Figure 5-2.	A Network of Selection/Resolution Processes	223
Figure 5-3.	Two Views of Time under Selection/Resolution.....	223
Figure 5-4.	The Parallel Composition of Two C/S Networks.....	225
Figure 5-5.	The Serial Composition of Two C/S Networks	226
Figure 5-6.	Selection Nondeterminism Supports for Modular Abstraction	246
Figure 5-7.	The Primitive Instructions of the SL Languages	248
Figure 5-8.	The SL Languages	249
Figure 5-9.	The Multi-Dimensional Structure of Time under Rule (iii).....	250
Figure 5-10.	The Redefinition of <i>mux</i> in SL_4	255
Figure 5-11.	A Chronogram Showing <i>multiplexer</i> in Action.....	256
Figure 5-12.	The Linearization of Time under Rule (iv)	257
Figure 5-13.	The Separated Semantics of CRP	261
Chapter 6. System Description Languages		265
Figure 6-1.	The Simulator Event Queues of the Discrete Event Model.....	280
Figure 6-2.	The VHDL Simulation Cycle	281
Figure 6-3.	The Traditional View of Two-Level Time in VHDL	282
Figure 6-4.	Interaction of a Verilog Task and Non-Blocking Assignment.....	283
Figure 6-5.	Verilog and VHDL Signal Assignment	284
Figure 6-6.	The Three-Levels of Time in VHDL Discrete Event Semantics ..	287
Figure 6-7.	An Example of a Non-Reactive VHDL Program	288
Figure 6-8.	The Classes of Signals in a VHDL Program	290
Figure 6-9.	An VHDL Program Illustrating $R\overline{M}\overline{C}$ Behavior.....	292
Figure 6-10.	The Execution of Figure 6-7 on an Odd-Valued Input	292
Figure 6-11.	The Execution of Figure 6-7 on an Even-Valued Input	293
Figure 6-12.	The Preemptive Aspect of VHDL Inertial Delay.....	295
Figure 6-13.	A VAL Annotation Specifying a Buffer with Delay	296
Figure 6-14.	The Preemptive Aspect of VHDL Transport Delay.....	297
Figure 6-15.	Violation Conditions for the VAL Assertion Flavors.....	298
Figure 6-16.	An Entity with VAL Annotations	299
Figure 6-17.	Extracting Synchronous Time from Discrete Event Time.....	302
Figure 6-18.	An Example of the Effect of \overline{R}_g in VHDL.....	304
Figure 6-19.	Execution where $R\overline{M}\overline{C}$ and $RM\overline{C}$ Semantics Differ.....	305
Figure 6-20.	A Subtle Modification of Figure 6-18 to Avoid \overline{M}	306
Figure 6-21.	Execution where and Semantics Match	307
Figure 6-22.	A VHDL Program where Modularity Checking is Simple.....	310
Figure 6-23.	An Example Distinguishing the Event and the Transaction	312
Figure 6-24.	An Example of the Confusion of State and Output	314
Figure 6-25.	An Example of the unaffected Waveform Constant	317
Figure 6-26.	Examples of the “Pulse-Width Rejection” Inertial Delay.....	318

Figure 6-27.	The Nondeterminism of a Shared Variable.....	320
Figure 6-28.	The VHDL '93 Simulation Cycle.....	321
Figure 6-28.	The AND and OR States of a StateChart.....	323
Figure 6-29.	SpecCharts adds Program Fragments to StateCharts.....	324
Figure 6-30.	A Sample SpecCharts Specification	326
Figure 6-31.	Equivalent SpecCharts Graphical Representation	327
Chapter 7.	The Non-Deterministic Abstract Machine	331
Figure 7-1.	The NDAM Representation as a Non-Abstract Semantics.....	332
Figure 7-2.	A Network of Processes Communicating via Signals	338
Figure 7-3.	A Network and its Process Tree.....	339
Figure 7-4.	The Use of ' require ' to Ensure Definition Before Use.....	344
Figure 7-5.	The Fully General Form of the TCWC Instruction	345
Figure 7-6.	Euclid's Algorithm in C and in NDAM Assembly	348
Figure 7-7a.	The Classic Stopwatch in Esterel.....	349
Figure 7-7b.	The NDAM Network for the Stopwatch.....	350
Figure 7-8.	Causal Consistency on a δ -Path.....	351
Figure 7-9.	The Manifestation of Causality Problems on a δ -Path	352
Figure 7-10.	Causality Checking by Explicit Enumeration.....	353
Figure 7-11.	Causality from the Perspective of the Image Computation	355
Figure 7-12.	Causality Checking by Implicit Enumeration.....	361
Figure 7-13.	The halting / halted structure	363
Figure 7-13.	The Split-Phase Structure of halting / halted	363
Figure 7-14.	The Fully-Loaded Structure of a halting / halted	364
Figure 7-15.	The Priority-Based Structure of the raise Estimate	365
Figure 7-16.	The Estimate Graph Template for a TCWC Instruction	366
Figure 7-17.	The Signal Handling Estimates.....	367
Figure 7-18.	The Control-Flow Estimates	367
Figure 7-19.	The Flatten Algorithm	368
Figure 7-20.	The Operational Semantics of the NDAM.....	372
Figure 7-21.	The Relational Semantics of the NDAM.....	376
Figure 7-22.	The Two-Level Approach with NDAM Assembly	379
Figure 7-23a.	A Recipe for Compiling Esterel into NDAM Assembly	380
Figure 7-23b.	A Recipe for Compiling Esterel into NDAM Assembly	381
Figure 7-23c.	A Recipe for Compiling Esterel into NDAM Assembly	382
Figure 7-23d.	A Recipe for Compiling Esterel into NDAM Assembly	383
Figure 7-24.	Templates for C Code Generation from NDAM Assembly.....	386
Figure 7-25.	The Code Template for the TCWC	388
Figure 7-26.	The Network EVAL Algorithm	389
Figure 7-27.	An Example of Static Signal Potentials.....	390

Figure 7-29. The Reasons for Variance in POT at a TCWC	390
Figure 7-28. An Example of Dynamic Signal Potentials at a TCWC	391
Figure 7-30. The MARK Algorithm	392
Figure 7-31. Implementation of Esterel in Automata	394
Figure 7-32. Implementation of Esterel in Circuits	395
Chapter 8. Conclusion	399
Figure 8-1. Substitutability of the Non-Abstract S_0 for the Fully Abstract S	400
Figure 8-2. The Eight Steps of Microsemantic Analysis	401
Figure 8-3. The Mathematical Baggage of the Non-Abstract S_0	402
Figure 8-4. The Methods of Surpassing the RMC Barrier	403
References	407
Appendix A. The Assembly Language of the NDAM	477
Figure A-1. Processes in a Network Communicating by Signals	478
Figure A-2. An Network Declaration and Its Process Tree	479
Figure A-3. Two Simple Processes	480
Figure A-4. Some Commonly-Used Type Declarations	482
Figure A-5. Type Declarations Using a Bounded Range	482
Figure A-6. The Various Sorts of Register Declaratoins	486
Figure A-7. The Various Sorts of Signal Declaration	487
Figure A-8. A Nondeterministic Goto Instruction	495
Figure A-9. A Code Fragment Using the If Instruction	495
Figure A-10. A Code Fragment Using the Case Instruction	496
Figure A-11. A Code Fragement Using the Present Instruction	497
Figure A-12. Some Example Signal Expressions	498
Figure A-13. A Code Fragment Using the 'select' Instruction	498
Figure A-14. A Code Fragment the Emit Instruction	499
Figure A-15. The Expansion of a wait Instruction	505
Appendix B. The Flatten Algorithm	507
Appendix C. Compilation of Esterel	513
Appendix D. The NDAM Runtime System	525

Acknowledgments

I am indebted to my advisor, Prof. Richard Newton for his continued support both before and during my graduate studies at U.C. Berkeley. It is entirely due to his encouragement and standards that this work progressed beyond its original disorganized state. I am grateful to Prof. Bob Brayton for agreeing to be both on my qualifying examination and dissertation committee. I have also to thank Prof. Karlene Roberts for her time in reading this dissertation.

This research was sponsored by the Advanced Research Projects Agency under Contract JFBI 90-073. Much of it was conducted in the CAD Group computing environment provided by Digital Equipment Corporation. The work on the Esterel compiler prototype in the early phase was supported by the Semiconductor Research Corporation under project DC-324-035.

Much of the work presented in this dissertation would not have been possible without the insight provided by Profs. Gérard Berry of l'École des Mines de Paris and Albert Benveniste of l'Institut National de Recherche en Informatique et Automatique. I am grateful for their financial support of my visit, during the spring of 1994, to Centre de Mathématiques Appliquées at Sophia-Antipolis. As well, they were both very generous with their personal time. My understanding of the Synchronous Languages and of the purpose and power of denotational semantics was enhanced tremendously by what I learned during that time.

I have also to thank Prof. Sue Graham both for chairing my qualifying examination committee and for her continued interest in my research. Also appreciated is Prof. Phil Colella for serving on my qualifying exam committee on such short notice. Over the years I have benefited greatly from interactions, debates and conversations with Profs. Alberto Sangiovanni-Vincentelli, Kathy Yelick and John Wawrzynek.

There are a number of people who contributed to my time at Berkeley in one way or another: Doug Fairbairn, David Henkel-Wallace, Ken Keller, Rick McGeer, Jim Rowson, Michael Tiemann and Dan Yoder. Also appreciated is the support for the CAD Group

for the CAD Group computing environment provided by Brad Krebs and his support staff of Mike Kiernan and Judd Reiffen. I thank them all.

The cheerful and knowledgeable service of Heather Levien and the rest of the staff in the E.E. Graduate Office over time, Mary Byrne, Carole Stewart and Genvieve Thiebaut have been very much appreciated. No matter what the issue, they always knew the answer and countless times expended extra effort to guide us all through and around the various administrative procedures involved with being a graduate student at Berkeley.

I thank my family, my parents Bruce and Joann as well as Gary, Alice, Lizzy and Louise Gray for their continued encouragement and understanding over the years. Finally, none of this would have been possible without the love and support of my wife Mary. I dedicate this work to her.

1 Introduction

The importance of embedded software and software models as the key part of a product has grown dramatically in recent years as the sophistication of semiconductor fabrication and system-level manufacturing capabilities have increased world-wide. It is becoming increasingly more difficult to differentiate a product based solely on its silicon or manufacturing content. Increasingly, the value in a product is derived from the intellectual capability embodied in its complexity and architecture, from the functionality encoded in embedded software, or at the hardware/software boundary. Coincidentally, as silicon manufacturing has allowed for more sophisticated designs to be produced, hardware design methods have evolved as well. In order to manage the increased complexity, design methods based on hardware description languages (HDLs) such as VHDL [384] [387] and Verilog [570] and automated synthesis procedures have become dominant. An HDL-based design methodology is distinguished by the primacy of software models in the design process. Thus, in a very real sense, as silicon fabrication capability has become more advanced, the design process for complex systems, whether delivered in hardware, software or some mixture in between, has come to require the ability to design, simulate and debug software and software-rendered models.

The class of systems of interest in this research have been dubbed *reactive systems*¹ by virtue of the three characteristics that distinguish them from the more familiar transforma-

tion or interactive systems. Reactive systems are distinguished from *transformational* systems by virtue of being in constant and continual interaction with their environment. Their very behavior is defined by these interactions and by the fact that they do not halt. The transformational system, on the other hand, carries with it subliminal notions of batch processing and questions about the potential and necessity of halting. Neither of these notions are suitable for the specification context.

A second aspect of reactive systems is the primacy of time in their specification. The system must respond within a certain time bound. In this sense, the reactive system must always respond marginally faster than its environment produces. It may never fall behind because the environment will not wait for it to catch up. Such a requirement is characteristic of control or supervisory applications. This distinguishes reactive systems from *interactive* systems such as a mainframe operating system where the environment (a user) is ready, willing and able to slow down and wait when the system becomes too overloaded. Finally, there is the issue of concurrency. Reactive systems are fundamentally concurrent. At the very least there is concurrency between the system and its environment, but more typically, there is some amount of internal coordination as well.

To say that a system is reactive implies that in a fundamental sense, it is driven by its environment. The environment produces events to which the system responds. In practice, designers are often concerned with a slightly broader class of systems which can impose their will on the environment of their own accord. These systems are called synchronous systems, reflecting their ability to model reactive-type behavior but in addition may emit events for which there is no direct causal connection with the environment. This dissertation is concerned with the properties of synchronous programs which are used as specifications in the design of reactive system.

1. The term *reactive system* has been proposed in a variety of places [599] [332] [601] [75]. The idea is generally attributed to Pnueli.

With the increase in complexity of system design has come the need to *verify* the functionality of the design before it is manufactured in volume. Traditionally the term *verification* when used in an industrial setting, has meant something akin to ‘exhaustive simulation.’ When a simulated exercise of all possible behaviors of the design is not feasible, some notion of coverage, functional or structural, is substituted in its stead. Many times though, even this restricted set of cases is not tractable so engineers are forced to settle for simulating according to a pre-characterized workload. An example might be the workload of booting the operating system that will run on the design and executing it for several (simulated) seconds thereafter. A single run might be expected to take from several tens-of-minutes to several days or even weeks depending on the granularity of the model and the complexity of the design.

In addition with increased design complexity comes the need for higher confidence in the correctness of the design before it is deployed. Estimates vary but it is generally understood that the cost of a mistake increases one order of magnitude for each stage of the design/manufacture/deployment process that it remains undetected. The cost of an error when discovered in the field may amount to many times the original sale price of the unit. In response to this need and in light of growing complexity there has been an increased interest in verification methods that provide higher levels of confidence than the traditional workload-based simulation paradigm. Such methods have been called *formal verification* because they center around proving that all possible behaviors of the design are correct under some formally-defined mathematical assumptions.

1.1 Verification Problems

The focus of verification, formal or otherwise, is the development of confidence about the design for all possible input sequences. Within that broad goal one can distinguish several distinct sorts of verification. That is, there are several different sorts of confidence that

need to be developed:¹

- Design Verification

This can be thought of as the question “*is what I asked for what I really want?*” The design, in all its volume, detail and glory must be checked against some isolated aspect behavior. The key at this level is the isolated aspect of the property. Design properties are fundamentally of the form “no, matter what else this thing does, property X must hold.”

- Implementation Verification

This sort of verification asks “*is what I asked for what I got?*” The design in all its volume, detail and glory has been transformed by some means into a low-level (manufacturable) description with even more volume and detail. The fundamental question is whether the two representations still compute the same thing in some abstract sense.

- Production Verification

The final question is of course “*can I tell a good one from a bad one?*” The implementation is being produced by the factory at a rate of X per day. Some percentage Y of these don't work enough to be useful. Define tests that will distinguish the nonsaleable ones.

The concern here is exclusively with the design verification problem.

1.1.1 Approaches to Design Verification

Traditionally there have been two ways to approach the design verification problem. The first is via methods such as theorem-proving whereby some general statement is made about the design using a logic. The statements are then checked for consistency using an automated theorem prover.² Typically such efforts have required a tremendous amount of designer intervention to complete the proof and to demonstrate the correctness of the design.

The second method, which has attracted much interest in recent years, involves an exhaustive exploration of the states of the system with the specific property to be verified being checked for every possible reachable state of the system. Here too, the property to

1. These definitions were first proposed by Devadas, Keutzer and Newton [234].

2. The following are representative of this approach: Cohn [203], Gordon [299], Boyer and Moore [102], Bevier *et al.* [87], Hunt [379] and Hwang [381].

be proved is stated in some sort of logic. In contrast with the theorem proving approach, the logics used in state exploration contexts tend to have highly restricted expressiveness. Such is the price of automation.

Within the state exploration approach there are at least three somewhat complementary formulations to the problem: *model checking*, *language containment*¹ and *bisimulation*. The three approaches are related in the sense that all are schemes for checking that a design provides a model for a formula in some logic. They are distinguished by the manner in which the design property is expressed and the details of the algorithms that can be used to verify that all possible configurations of the design obey the property.

In the *model checking* approach [249] [608] [518], properties are declared as a sentence in an appropriate temporal logic [248]. The verification step determines that the design at hand is a valid model for the formula. This is a much simpler task than proving that the property is valid in all possible models as is the case in the theorem proving approach. At the user level, model checking is distinguished by the fact that the design property is expressed in a sentential form that is fundamentally different than the form of the design itself.

A second alternative that uses state exploration is called *language containment* [706] [449] [454]. In this scheme, the property is declared in the form of yet another design component, one whose only purpose is to monitor the system's behavior. Both the design and the property to be checked are modeled as finite automata, typically automata accepting infinite strings. The language containment check ensures that the behaviors produced

1. Unfortunately there are two different languages intrinsic to the subject matter at hand. One is a *programming language* which is used by humans to express the design and its properties. This sort of language is referred to in this work as the *high level language*, the *specification language*, the *design description language*, or simply as the *language* where the context is clear. The other referent is the automata-theoretic set of strings or sequences generated or recognized by a particular finite automata. This language is a mathematical quantity and referred to as the *language of an automaton*. It is denoted by $L(A)$ for some automaton A or simply L where the automaton is plain.

by the design are a subset of the behaviors allowed by the property. In simple terms, language containment ensures that the design doesn't do *more* than the property, or equivalently, the design doesn't have any behaviors that are *actively disallowed* by the property.

The third state exploration alternative is *bisimulation* [531] [580] [341] which is based on an abstract notion of testing. In this scheme, two systems are compared according to the kinds of activities or tests in which they can participate. Both the design and the property to be checked are modeled as finite automata. In this case the automata are systems of states with labeled transitions among the states. The labels on the transitions model actions in which the system can engage. Two states are bisimilar if for every action and every successor state in the first system, there is always exists a similar successor state in the second system and vice versa. Two systems are bisimilar if every reachable state in each is bisimilar. In simple terms, bisimulation allows a property to be proved by filtering and reducing down a hugely complex design relative to a simple property automata. The property automata is simple enough to be shown correct by inspection.

Even though it is less general than theorem proving, the state exploration approach is attractive for it requires little or no designer intervention: the design and the property to be proved are presented, the algorithm churns for some amount of time, finally an answer is returned. Additionally, the algorithms can be extended to return not only the *pass/fail* answer but also a reason *why* the failure occurred in the form of an error trace to the point that the error occurred. From a high level, any formal verification scheme based on state exploration can be viewed as following the schematic of Figure 1-1.

1.1.2 The State Explosion Problem

Unfortunately however, all state exploration methods suffer from a problem known as *state explosion*. Intuitively that problem is that the number of states of the composite system grows with the product of the number of states of the individual components. Thus in

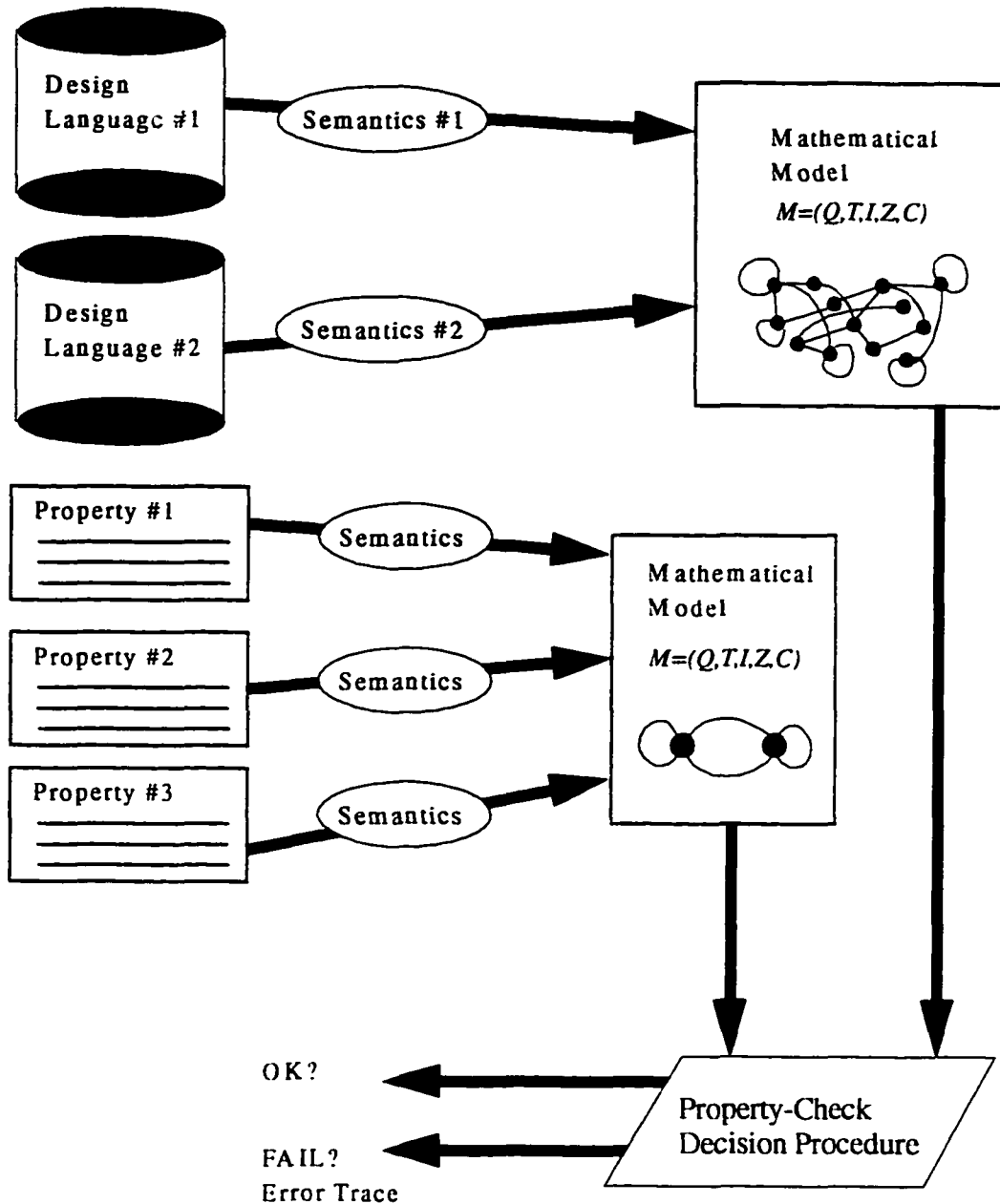


Figure 1-1. A Schematic of a Formal Verification System

general the system's state space is exponentially related to the size of the state space of any individual component. Recently however, techniques known as *symbolic methods*¹ have been developed which avoid the state explosion problem in many cases. The technique involves conceptually thinking about a (finite) set of states in terms of its character-

istic function [324].¹ The characteristic function is represented as a Boolean formula in Bryant's Ordered Binary Decision Diagram [125]. The power of the OBDD approach to symbolic representation is drawn from the empirical observation that the size of an OBDD is often much smaller than the set of states that it represents. Thus in many cases state explosion is avoided. Unfortunately in many other common cases the size of the OBDD representation is larger than an equivalent non-symbolic representation. This represents a limitation of the naive application of symbolic methods to formal verification. Indeed, a certain number of size bounds are known about OBDDs [126] [233] [518], though the general thrust of all the results is that the size of an OBDD data structure bears little relation to any intuitive notions of the complexity of the functions that they represent. What is known however is that for a large class of simple functions,² the OBDDs are much smaller than the sets which they represent and the technique works quite well.

This investigation concentrates on the effect, *at the semantic level*, of use of chains of simple steps, represented symbolically in an OBDD, to approximate the complex macrostep behavior of a synchronous systems. The focus is exclusively on the conditions which must occur for a semantics of small-step paths for to approximate the usual single-step form. As such, though the guiding measure throughout is the use of symbolic methods and the asymptotic size of the associated OBDDs, specific measurements are sup-

1. There is at this point an extensive literature on the so called symbolic methods for avoiding the state explosion problem. Almost all of them use Bryant's OBDD data structure [125] [105] [127] or some variant of it. Overviews on the use of symbolic methods in formal verification schemes can be found in McMillan [518] and Kurshan [454]. All symbolic representation schemes however are severely limited by the strongly heuristic nature of OBDD methods. More detailed comments on the relevance and utility of OBDD methods in this work are found in Section 3.3.4.

1. Though the characteristic function formulation of the finite sets was previously known [324], the introduction of the use of an arbitrary boolean formula to represent a set of states is attributed variously to Coudert *et al.* [209], Touati *et al.* [690], and Burch *et al.* [137] because of their almost simultaneous publication of techniques for using Bryant's OBDD in this manner.

2. Only the most vague notion of what constitutes a *simple function* is necessary for this work. More in-depth treatments of function complexity and their effects on OBDD size are given in Bryant [126], Devadas [233] and McMillan [518]. Non-simple functions are typically data-path oriented computations such as multiplication or shift-store-and-rotate where every bit of input is somehow related to every bit of output.

pressed in the name of a general claim about the conditions necessary for a small-step path-based semantics to be equivalent to a big-step single-step semantics.

1.2 System Descriptions and Formal Methods

Design verification is considered to be a *formal method* because it occurs at a symbolic level, based on the mathematical properties of the system and its underlying computational model it is complete in terms of what it guarantees. Any formal method consists of three components [729]:¹

1. a pair of languages; one to describe the system and one to describe its properties,
2. a semantics or mathematical model of computation,
3. an algorithm that checks properties of the design via the mathematical model.

The traditional focus in formal methods has been on algorithm complexity, both asymptotic and expected, given a material representation (*i.e.* a data structure and a strategy for manipulating it), an arbitrary design instance and a semantics on one of the standard mathematical models. This is because, in the general case, design property checking is understood to have high complexity, although polynomial-time algorithms are known for some mildly-restricted scenarios. To a lesser extent, there has been some debate about the utility and relationship between the various mathematical models themselves: the various kinds of ω -automata or the logics of linear or branching time. These arguments have almost always been conducted with an eye towards the intrinsic algorithmic complexity of the model and how hard it is, in a practical sense, to check a property under that model. Nearly all investigations have ignored the contribution of the first component: the representation languages used for the design and its properties.

The design representation languages of any formal method admit to a finer analysis.

1. Actually Wolper [729] separates out the program description language and property description language as being fundamentally distinct. There are four elements in that presentation.

They can be thought of as having three relevant sub-components:

1. a *behavioral* aspect declaring the *temporal evolution* of the system,
2. a *structural* aspect for aggregating and composing smaller description fragments,
3. some *property declarations* which state the correct behaviors of the system.

Of these, the first two are the ones of primary interest here for they have been studied the least.

If one is to interpret a representation of *pure* behavior as one which provides no hint of or constraint on a possible implementation then, within the synchronous model, a behavior can be nothing more substantial than an exhaustive listing of pairs of inputs and outputs that are allowed by the system. Common examples of pure behavior are a primitive combinational gate in a library, behaviorally specified by its truth table, or an instruction in an instruction set architecture, operationally specified by a microarchitecture or mathematically by a transition relation.

Primitive pure behavior is not enough to describe systems. It is infeasible to give an exhaustive listing of I/O pairings for a system of any interesting size. As an answer to this, a *structural* means for aggregating smaller units of behavior into larger ones is generally admitted in any design description language. Common examples of this are the state machine defined as the synchronous product of two component machines or a reactive software system where the “instantaneous” response is actually produced through a sequence of micro-responses computed across a number of submodules. In these cases the behaviors of the whole are defined in terms of the behaviors of the components under an aggregation rule.

1.2.1 The Significance of Semantic Models

The question then in formal methods is to what extent and in what ways can the structure of the design representation be exploited. This somewhat vague notion gives rise to

two specific questions which are the central focus of this dissertation:

1. How can the design representation language, inclusive of both its behavioral and structural aspects, be related to the mathematical model and to what extent should the structural artifacts of the design representation be made visible in the mathematical model?
2. How should the structural aspects of the design representation language and their visibility within the mathematical model be related to the performance and the implementation of a property-checking algorithm?

Both of these questions focus on *language semantics* which identifies the relationship between a class of programs, as textual artifacts, and an underlying computational model, which is a mathematical construction.

Language semantics, as an area of investigation, must involve the study of the properties of semantic models themselves as well as the association of these models with programs. The concern here clearly is with the computational properties of these semantic models and the property-checking algorithms that are built on top of them. This focus could properly be called *computational semantics* as understood to mean the design and analysis of semantics from a computational perspective as well as the study of efficient algorithms for deciding questions within the framework of a semantic model. As such, it concentrates on the interactions between language semantics, semantic models and formal verification algorithms.¹ It is the construction and properties of such semantics which is the subject of this investigation. The following sections present the argument of this thesis in summary.

1.2.2 Languages, Semantics and Models

The semantics of a programming language is said to define the *meaning* of the programs in that language. The association of a mathematically meaningful structure with raw program text is accomplished through a number of techniques. All of them however, in some way, associate with each program instance, a corresponding instance of a general mathe-

1. The sense of this definition is as an amalgam of the traditional sense of *computational logic* [103] [338] as applied to a logic defining a computation and that of *computational geometry* [606] [730] which focuses on the algorithmic aspects of problems in geometry.

mathematical model. A language, its semantics and its semantic model are a system (L, M, S) as follows:

- language*: a set L of programs that can be written or drawn in some notation,
- model*: a mathematical structure M such as a logic, a state- or even a function space.
- semantics*: a semantic map $S:L \rightarrow M$ which is written $S[[\text{program}]] = \text{model}$.

This gives a structure defining a relationship between the language and the model as shown in Figure 1-2. That diagram is presented at this point, in its utter simplicity, because it will be embellished in a subsequent section. This investigation focuses on the properties of S and M as they relate both to those embellishments and to the fundamental computations of formal verification.

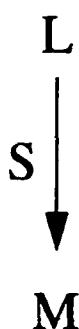


Figure 1-2. A Language, its Semantics and the Model

A semantics is said to be *denotational* when the denotation, in M , of a program, in L , depends on the denotations of its constituent terms and not on their structure. This means that S is defined using one or more composition functions $\circ : M \times M \rightarrow M$ so that:

$$S[[\text{statement}_1 ; \text{statement}_2]] = \text{model}_1 \circ \text{model}_2 = \text{model}_{1,2}$$

The language composition term (the ‘;’ above) may vary according to the kind of statement, just as the composition function $\circ : M \times M \rightarrow M$ may vary according to the seman-

tics S and model M . The essential point is that association at the language level denotes composition at the mathematical level.

A semantics is said to be *extensional* when two denotations in the model are the same when they behave the same way under some notion of behavior. Typically a requirement of extensionality is discharged through the use of a unique element $m \in M$ to represent each distinguishable behavior. For example, the transition relation of a combinational network is fully abstract whereas a sum-of-products “cover” for the network’s function is not extensional.

A related notion is *full abstraction* which is the condition that language terms have the same denotations whenever the terms are interchangeable in complete programs. Full abstraction is generally considered a good thing in a semantics because it means that there is no extraneous information in it. A less abstract semantics (*i.e.* a non-abstract semantics) would have extra information in it. These extra artifacts are properly called implementation details.

Unfortunately however, fully abstract semantics for finite state models are known to be too computationally expensive in the general case, even when symbolic methods are used to combat the state explosion problem. This dissertation therefore is concerned with a notion of *substitutability* wherein a non-abstract semantics which is computationally more regular can be said to stand in for the fully abstract one and vice versa. Substitutability rests on three propositions which are argued in the sequel: first, that the transition relation is the fundamental semantic model for finite state systems; second, that full abstraction implies a single-level discrete time; and third, that non-abstractness implies a fine structure within the discrete time that can be properly called δ -time. The highlight of this notion of δ -time is that it is defined mathematically and wholly independent of any simulator event loop. Substitutability is thus a condition on when δ -time is a well-defined fine

structure within a semantics.

1.3 Transition Semantics of Finite Models

In formal methods, the most widely used semantic model is one that can be seen to have a direct analogy to both sequential hardware systems as well as to synchronous software systems with finite state. It is a temporal model consisting of a set of states, a transition relation and a labeling of the states:¹

$$M = (Q, T)$$

where:

Q is a set of states,

$T \subseteq Q \times Q$ is the transition relation describing the step-by-step evolution of M .

Within this framework, the transition relation T can be conceptually thought of as an exhaustive tabular listing of the dynamic evolution of the system. The temporal structure $M = (Q, T)$ is therefore a pure behavior under the definition given above.

What is significant here is the exclusive focus on the set of states Q and the transition relation T as the primitive definition of behavior. Various semantic models differ with minor modifications about this central scheme but the basic nature of the state set Q and the transition relation T remains. Indeed, save for one or two rather exotic cases which are exemplary in their originality, all finite-state semantics and their models focus on defining behavior in terms of transitions between states. However, within the general framework of $M = (Q, T)$ there are numerous variations. For example, adding labeled enabling conditions $\alpha \in \Sigma$ on the edges in the transition relation produces $T \subseteq Q \times \Sigma \times Q$ and defines a class of automata which are subject to some formal language-theoretic acceptance condi-

1. Such a model is also interchangeably referred to as a *Kripke structure* [443] in the literature. In the general case, there is a third element $\Phi: Q \rightarrow 2^{AP}$ which is a function from the set of states Q to a subset of some externally specified atomic propositions AP . Without loss of generality, the proposition map Φ can be taken as isomorphic to Q and so are elided for this work.

tion [684]. Interpreting this enabling condition as an input/output pair $(i, o) \in I \times O$ produces the transition relation as $T \subseteq Q \times I \times O \times Q$ which is the familiar transition relation of a finite state machine with inputs and outputs [439]. One issue which should be noted at this point is that M , in its unembellished form is a primitive. In particular, it does not support composition: there is no notion of communication upon which to build such a coordination scheme. In the sequel such an embellishment in the form of inputs and outputs is assumed as necessary.

The survey of semantics and semantic models in Chapter 2 establishes that at a very primitive level the semantic models of finite state systems, identified by whatever reasonable means, are characterized by states and transition relations. There are exceptions to this and thus for completeness two of the more exotic language-model systems are reviewed to illustrate the non-standard approach. The major point at this early stage is that the state transition approach is fundamental and that this view holds independent of any language syntax or semantic model abstraction. The argument is essentially that “this is now nature really works.”

1.3.1 Full Abstraction and Discrete Time

Having established that the transition relation is the fundamental semantic model of finite state systems, the next obvious question is how the property of full abstraction makes its appearance in a transition-based model. The answer can be seen by examining the notion of what constitutes a ‘behavior’ in a fully abstract regime. Leaving unspecified exactly what *defines* a behavior for the moment,¹ a broad observation about behaviors, the notion of full abstractness of semantic models $M = (Q, T)$ and the structure of time can be made. The notion of full abstraction in a semantics refers to the behavioral properties

1. The presentation of a concrete scheme for behavioral property definition is deferred until Section 3.2. For the present development, any of the previously mentioned schemes of model checking, language containment or bisimulation can be understood as defining a behavior in this vague sense.

of denotations $m \in M$ not to the internal structure within any element m itself. Thus the observation is that the kinds of behaviors of interest in formal verification are those which occur across multiple steps in M . That is, the behavioral properties of interest are somehow characterized solely in terms of sequences of states or sequences of transitions. In particular, there is no notion of behavioral properties which are defined in terms of intra-transition relationships. The answer therefore is that while such a fine structure may exist in the transition relation, it is of no use in the definition of behavioral properties.

The vague and informal definition of behavior provided at this point is sufficient to illustrate the exclusive focus on states $q \in Q$ and transitions $t \in T$ necessarily implies that time has a single-level discrete structure, *as seen from the perspective of the behavioral properties*. This one-level model of time is illustrated in the diagram of Figure 1-3. This characterization of the discrete nature of time means that there are two views of time, one view from within the semantic framework and one view from the outside. From the internal perspective, time is measured in units of steps and two events occurring in the same step are said to occur *at the same time*, or equivalently they occur *in zero time*. This is because from the internal perspective, time is measured in units of steps. From an external perspective there may be some observable ordering or measurable temporal distance between the two intra-transition events. This is because from the external perspective, time is measured in units such as nanoseconds, minutes or days. The key distinction is that within the framework of a fully abstract semantics, temporal relationships within a step are entirely unobservable.

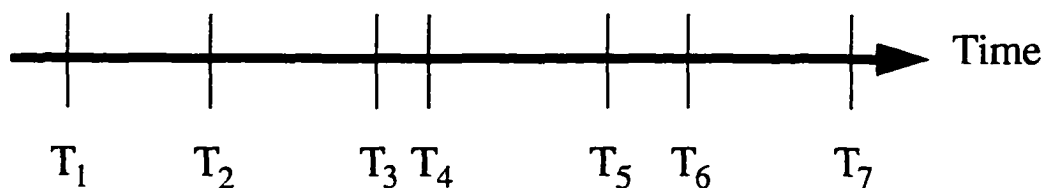


Figure 1-3. The One-Level Time of a Fully Abstract Semantics

1.3.2 Non-Abstractness and δ -Time

If a fully abstract semantics is said to uniquely reflect the operational behavior of the language without containing any extra extraneous information, then a non-abstract semantics is one which denotes the operational behavior in some *non-unique* way. Indeed, there may be multiple denotations in the model that have the same behavior.

The previous section defined a transition relation for a fully abstract model as one where there is no internal structure. There, the transition relation was conceptually an exhaustive listing of current-state/next-state pairs. This implied a discrete notion of time which had a single level and between any two temporal instants nothing could be observed. In contrast a non-abstract transition system is one where there is internal structure. The transitions between states, instead of being atomic, are *paths of state transitions* on some finer level of granularity. The transition relation at this finer level is T_δ and the transitions are called δ -transitions. This finer level of time is called δ -time¹ and the relative scales of δ -time and (macro) time are related only by the embedding of δ -time within a single macro step. This two-level model of time is illustrated in Figure 1-3. That macro time and δ -time are related by embedding only is significant because it means that there may be an arbitrary, but not infinite, number of δ -steps within a macro step. This will be important in the definition of a substitutability property wherein a non-abstract semantics defined on $M_\delta = (Q_\delta, T_\delta)$ can be said to stand in for a fully abstract semantics defined on $M = (Q, T)$ and vice versa.

1.3.3 The Substitutability Condition

Having stated the notions of the primitiveness of the transition-based model, full abstraction defining discrete time and non-abstraction inducing δ -time, the conditions for

1. The symbol δ is used here to as a subscript mark to denote quantities in microtime. The natural character μ already has a use in the μ -Calculus notation as the least fixed-point operator. Too, the term "delta-time" has precedent in the literature from the time model of VHDL [384].



Figure 1-4. The Two-Level Time of a Non-Abstract Semantics

substitutability can now be defined. One can observe that the current interest in formal verification techniques using state-space exploration are driven by the ability to process large sets of states at one time using symbolic representations for the state sets. The key aspect of the substitutability condition is related to this shift from a singleton-oriented to a set-oriented viewpoint. The insight is that Scott's domain-theoretic model¹ of computable functions applies not only to functions acting on a single state (represented in non-symbolic form) but also to functions acting on sets of states (represented in symbolic form) as well.

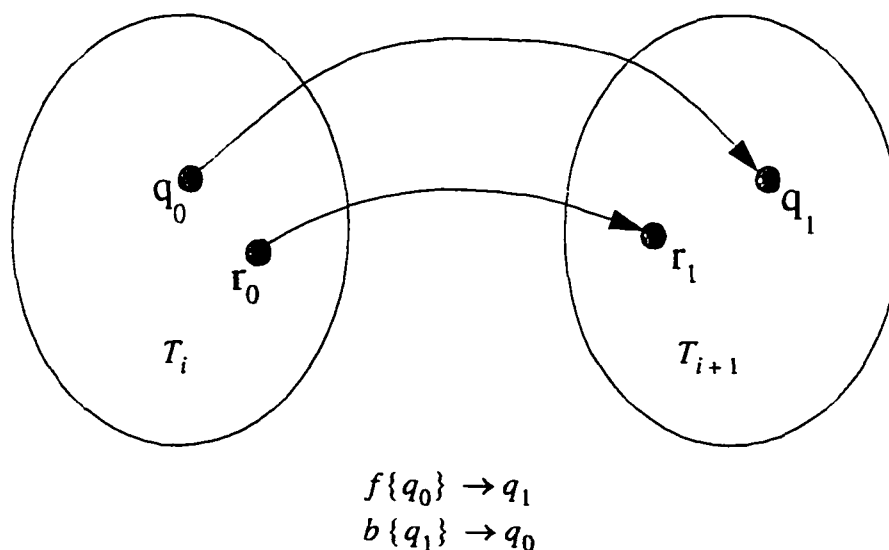


Figure 1-5. The Forward and Backward Simulation Operations

1. The presentation of Scott's theory of semantic domains is deferred until Chapter 3.

In its basic form, the domain-theoretic model represents every computable function as the fixed point of a (different) monotonic and continuous functional on a directed domain. The function which determines the singleton successor state q_1 that is related to the singleton predecessor state q_0 by a transition relation T is computable in a functional form. This function, depicted in Figure 1-5, is denoted by $f\{q_0\}$ and is generally referred to as a forward simulation computation. The function that determines the successor state set Q_1 which is related to the predecessor state set Q_0 by a transition relation T is computable as well. This function, depicted in Figure 1-6, is denoted by $F\{Q_0\}$ and is referred to as the *forward image computation*.¹ According to the theory, there must be an fixed formulation for both the $f\{q\}$ and $F\{Q\}$ cases. Substitutability will be the conditions when such an approximation-based computation, which is necessarily non-abstract, can be seen to be a replacement for a fully-abstract single-step version, and vice versa.

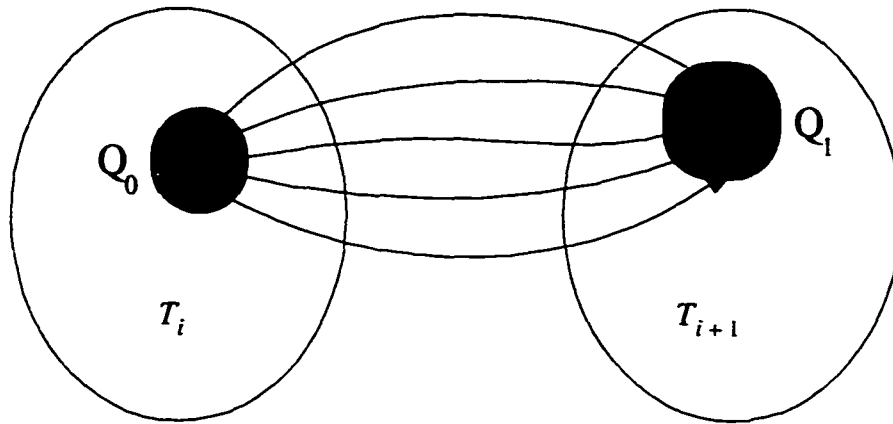
More concretely, in the case of the forward execution computation f shown in Figure 1-5, Scott's theory implies that f can be approximated as \tilde{f} where $q_1 = \tilde{f}\{q_0\}$. Since q_1 is computable from q_0 by \tilde{f} there must exist a functional, call it \mathcal{f}_δ , such that $\tilde{f} = \mathcal{f}_\delta\{\tilde{f}\}$. The functional \tilde{f} is the least function satisfying $\tilde{f} = \mathcal{f}_\delta\{\tilde{f}\}$. Thus q_1 can be computed from q_0 by means of a fixed point computation that produces \tilde{f} through a series of better and better approximations to a well-defined limit. So $\tilde{f} = \mu\mathcal{f}_\delta$ and $q_1 = (\mu\mathcal{f}_\delta)\{q_0\}$.

The key to substitutability is that Scott's theory necessarily applies to computations involving *sets of states* as well as singleton states. In the case of the set-oriented computations, depicted Figure 1-6, the argument is recapitulated in its entirety. Scott's theory

1. Historically [427] the forward direction has been called the *image computation* and the backward direction has been called the *pre-image computation*.

This terminology has been adopted more or less directly in symbolic formal verification arena as well: *c.f.* Coudert *et al.* [209], Touati *et al.* [690] and Burch *et al.* [137].

The terms *forward image* and *backward image* are used in this presentation to emphasize that the operations are indeed symmetric, especially so in the fully abstract case. Additionally the terms *forward* and *backward* motivate the use of $F\{Q\}$ and $B\{Q\}$ as the operator symbols for the respective computations.



$$F\{Q_0\} \rightarrow Q_1$$

$$B\{Q_1\} \rightarrow Q_0$$

Figure 1-6. The Forward and Backward Image Computations

implies that F can be approximated as \bar{F} where $Q_1 = \bar{F}\{Q_0\}$. Since Q_1 is computable from Q_0 by \bar{F} there must exist a functional, call it \mathcal{F}_δ , such that $\bar{F} = \mathcal{F}_\delta\{\bar{F}\}$. The functional \bar{F} is the least function satisfying $\bar{F} = \mathcal{F}_\delta\{\bar{F}\}$. Thus Q_1 can be computed from Q_0 in a means of a fixed point computation that produces \bar{F} through a series of better and better approximations to a well-defined limit. So $\bar{F} = \mu\mathcal{F}_\delta$ and $Q_1 = (\mu\mathcal{F}_\delta)\{Q_0\}$.

This work therefore concentrates exclusively on this latter formulation in terms of sets of elements Q , the image functional \bar{F} and the image approximator functional \mathcal{F}_δ (and of course the corresponding backward image functionals \bar{B} in terms of its image approximator functional \mathcal{B}_δ). This is here called an *image semantics*. Not only does the formulation allow for the convenient treatment of unbounded nondeterministic behavior at the macrostep level but it also has the advantage that it entirely subsumes any construction based on a "point" simulation semantics.

Chapter 3 is dedicated to defining substitutability at the model level, purely in terms of $F\{Q\}$ and $B\{Q\}$, and thus in the absence of any language or semantic considerations. There, substitutability is shown to be the case when the diagram of Figure 1-2 commutes.¹

That is, when there exists a quotient or projection operation Π which takes structures in the model M_δ and projects them down onto the fully abstract model M . This construction produces the intuitive notion of δ -time within discrete (macro) time in a mathematically sound way and hence is independent of any simulator event loop. For every program $p \in L$ with a fully abstract semantics $S[[p]] \in M$ there ought to exist a non-abstract semantics $S_\delta[[p]] \in M_\delta$. This means that there must exist a well-defined projection function $\Pi: M_\delta \rightarrow M$. The existence of such a function for any non-abstract semantics is strongly dependent on the kinds of properties that it must preserve.

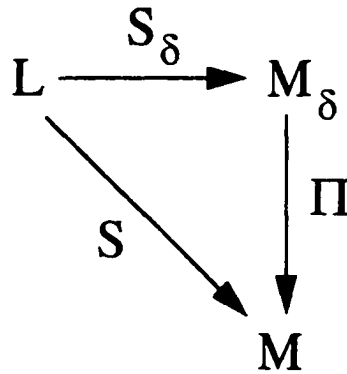


Figure 1-7. A Language, its Fully Abstract and Its Non-Abstract Semantics

1.4 Limits on Semantic Models

Given the potential expressiveness of S_δ with its multi-step δ -paths and the requirement that Π take any such δ -path of M_δ to a (macro) step of M in such a way that $S = S_\delta \circ \Pi$, there must be some limitations on the admissible δ -paths definable by S_δ . Intuitively, the set of admissible δ -paths definable by S_δ are constrained in three ways.

1. The diagram, and indeed the whole approach of this work broadly follows Mulmuley [550]. The focus here however is exclusively with finite-state synchronous systems rather than the more general typed λ -calculus (PCF).

1. S_δ must be able to express the same range of elements in M that are expressible by S . S has a property known as *responsiveness* so S_δ must have this property as well.
2. The δ -paths defined by S_δ must not contain information that Π cannot project down from M_δ onto M . This is a regularity constraint known as *modularity*.
3. Since Π is a projection, any order observable within $m_1, m_2 \in M_\delta$ must also be observable between their projections $\Pi\{m_1\}, \Pi\{m_2\} \in M$. This is a principle known as *causality* which is implicitly carried across from the δ -paths in M_δ .

Thus for substitutability to hold in a general semantics S_δ , the semantics must be responsive, modular and causal. Unfortunately, it can be shown that no semantics can possess all of these properties at one time. This sharply limits substitutability as a general property of an arbitrary semantics. Fortunately however, substitutability can be recovered on a case-by-case basis through various kinds of analyses. First however, a more concrete understanding of the notions of responsiveness, modularity and causality must be developed.

1.4.1 Properties of Semantic Models

The three properties are described informally here with the formal definition deferred until Section 4.1.

Responsiveness

A system is considered responsive if the system's output comes simultaneously with the input that causes it. A semantics is considered to be responsive if it is possible to define a responsive system under the rules of the semantics. For this definition to be meaningful it must be interpreted in the framework of abstract notion of time wherein two events can be considered to occur at the same time. The intuitive notion thus is of two events occurring within the same clock cycle in the case of hardware. For more abstract synchronous software systems there is typically a notion of an event and its reaction which are considered as an atomic unit. In the mathematical sense, the responsiveness criterion can be shown to be equivalent to the distinction between a Moore machine and a Mealy machine [368]. Responsiveness has its basis in the fully abstract model M because clearly Mealy machine behavior can be declared in that model.

Modularity

A semantics is modular if the rule for aggregating components into a whole obeys the property that all parts of the system can be treated symmetrically, inclusive of intra-component communications and component-to-environment communications. Further, every part of the system must have the same view of the instant-to-instant computation. In particular, a semantics is non-modular when information is communicated between components through either the order that outputs are produced or if outputs can adopt more than one value in a step. Modularity is discharged in practice through a broadcast model of inter-component communication with single assignment per step. In an informal sense, the modularity criterion can be thought of as a sort of linearity condition under which the behaviors of the product machine is the intersection of the behaviors of the component machines. Modularity has its basis in the fully abstract model M as a limitation on the amount of information which can be exchanged in a step. In the single-step form outputs can only be assigned once. Further, since there is only one step there can be no order between the output-defining steps; all the outputs must be defined at once in the single step of the semantics.

Causality

A semantics is causal if for any event that is generated in an instant, there is a direct (causal) chain of events leading up to it. Mathematically stated, there must be a consistent ordering among the components' executions when they are aggregated into the whole. This definition is consistent with the standard usage of the term in system theory: a causal system is defined as one which does not anticipate its own future.¹ Causality is thus a practical condition that requires that a system be able to compute its reaction based only on its view of present and past behavior, without having to look ahead in time. Causality

1. Similar definitions can be found in most systems-theory textbooks [568] [399] [735].

has its basis in M_δ from the explicit ordering of the δ -paths which is implicitly carried across into M .

These three properties cover aspects of central importance to the existence of $\Pi: M_\delta \rightarrow M$ in terms of the expressiveness in M_δ and M . Responsiveness, with its basis in M , is a compactness condition driven by the fact that Mealy machines are more compact than Moore machines by a multiplicative factor [439]. Modularity, with its basis in M , is a compositional linearity condition that is fundamental to any formal verification method based on state space exploration. Finally, causality, with its obvious basis in M_δ with its projected effect being visible in M , is a condition on the realizability of the transition structure T in the physical world where time only runs forward. The origin of these properties is depicted in Figure 1-2. Unfortunately, in sum, these properties are incompatible; there is no general semantics S or S_δ which is responsive, modular and causal.

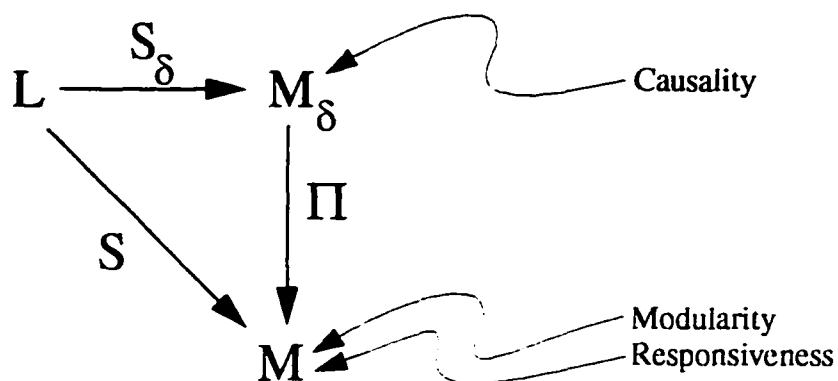


Figure 1-8. The Origins of Structural Properties in Semantics Models

1.4.2 The RMC Barrier

Any finite semantic model $M = (Q, T)$ can be characterized by whether it is responsive or not (R or \bar{R}), modular or not (M or \bar{M}) and causal or not (C or \bar{C}). Such a characterization is of course not unique. What is important however is the theorem, due to Huizing and Gerth [378] which states that no semantics can be R , M and C all at the

same time: There is no *RMC* semantics. This theorem is here called the *RMC Barrier Theorem*.¹ The theorem and its proof are given in Section 4.2 both for completeness and because the details of the proof both illuminates ways to approach the design and analysis of semantics, and also the ways of surpassing the *RMC Barrier* on a case-by-case basis.

1.4.3 Microsemantics

The notion of approximating the transition relation T by a finite series a *microsemantics* under which transitions $t \in T$ are constructed from a sequence of smaller transitions $t_\delta \in T_\delta$. As mentioned, it is typically impractical to give T directly in the monolithic form for any system of reasonable size. A structural method for aggregating smaller sub-models must be defined. In fact this aggregation is the major utility of the semantic composition operator $\circ : M \times M \rightarrow M$. In the abstract, semantic models are distinguished by the rules under which this aggregation is accomplished or even allowed. One of most straightforward possibilities is the “synchronous product” by which a temporal structure is defined in terms of two smaller substructures:²

$$M = (Q, T) = M_1 \times M_2 = (Q_1 \times Q_2, T_1 \times T_2)$$

In the general case one writes $M = \prod M_i$.

There are a wide variety of possibilities for the definition of the synchronous product operator $T_1 \times T_2$. It may be as simple as representing the coordinated single-step transitions as would be natural under the fully abstract transition relation $T_1 \times T_2 = T_1 \wedge T_2$. More likely, and definitely more interesting however, are the cases where $T_1 \times T_2$ depends on some finer structure of Q_δ or T_δ . For example it may be convenient to con-

1. The term *The RMC Barrier* is unique to this presentation.

2. The reader is asked to forego the minor notational sloppiness here that T_1 and T_2 need to be “raised” into the domain of $Q_1 \times Q_2$. Indeed, in the strictest sense, one cannot directly multiply two Kripke structures together because there is no notion of external communication, inputs or outputs, with which to modulate such activity. M is explicitly a static structure. The intent here is to present the flavor of the situation which is implicitly or explicitly in terms of states and transitions between the states. A more formal presentation is deferred until Chapter 3.

sider a state $q \in Q$ as consisting of a state component s and an input/output component o , whereby each q is a pair $q = (s, o)$. Or, it may be convenient to consider transitions $t \in T$ as actually having some internal structure, of having a δ -time which consists of some sort of multi-step chain where each $t = (q_0, \delta_1, \delta_2, \dots, \delta_k, q_1)$.

The identification of such finer intra-model properties within Q_δ and T_δ of the model $M_\delta = (Q_\delta, T_\delta)$ is here called *microsemantic analysis*. This analysis characterizes any semantics in terms of its microsemantic operators and its path-spanning consistency conditions over the δ -transitions $t_\delta \in T_\delta$. This analysis in turn leads to a summarization of the semantic model in terms of R , M , and C . Several important semantic models which have been proposed over the years are analyzed in terms of microsemantics in Section 4.3.

1.4.4 Beyond the RMC Barrier

The RMC Barrier implies that there is no clever language semantics, extant now or yet to be discovered, which provides all the RMC benefits, under all conditions and on a guaranteed basis. Fortunately however, the statement of the RMC Barrier Theorem is phrased in terms of *semantics* which are mathematical rule structures that describe whole classes of systems. The RMC Barrier is silent on the properties of *individual instances* of a semantics. Individual instances are specific programs or particular systems descriptions. This subtle distinction can be seen in the definition of responsiveness which is phrased in terms of the kinds of systems that a semantics allows to be defined.

The subtle difference between semantic models and an instance of a semantic model implies that it is possible to move beyond the RMC Barrier on a case-by-case basis. Five different ways in which the RMC Barrier has been surpassed in existing systems are identified in Section 4.4. Two of these methods are the static and dynamic check for *RMC* on the specific program instance at hand. In such schemes, the semantics S_δ on the model M_δ and the projection operation Π mapping down to the model M are contradictory (as

predicted by the RMC Barrier Theorem). These two methods for surpassing the RMC Barrier take advantage the potential for $S_{\delta} \circ \Pi$ to be consistent on an admissible $m \in M_{\delta}$ but not in the general case. In fact, the definition of the Synchronous Languages such as Esterel [75] [295], Lustre [153] [321] [320], Signal [464] [65] [465] and Argos [501] [502] [503] are phrased in terms of an explicit procedure that verifies that only programs having the RMC property are declared legal. This procedure is usually called *causality checking* because the semantics are already responsive and modular, it is only the causality property that may fail to hold. Causality problems manifest themselves in different ways depending on the particular operators available in the language's semantics.

1.5 Analysis and Design of Semantics, Languages and Models

These four elements, the view that transition systems are the fundamental semantic model, microsemantic analysis, the three properties of responsiveness, modularity and causality and the ways for surpassing the RMC Barrier are treated here as a framework both for classifying existing semantics and languages as well as for proposing new system description languages.

1.5.1 Applied Semantics

Within the framework, a survey of existing semantic models is undertaken in Chapter 5. There, a variety of models are examined. In addition to the models which directly fall into the class of decidable finite-state synchronous systems, two choice examples of undecidable semantic models are examined as well. These two examples are interesting because they lie at the border of undecidability and therefore give some sense of the limits to semantic expressiveness beyond mere state transitions. Their undecidability shows that any of the obvious features which might be added to a state-transition semantics, render it non-finite or otherwise undecidable.

The overall argument of the treatment in Chapter 5 is the claim that the Synchronous

Languages, Esterel, Argos, Signal and Lustre the best choice of semantics in the sense of offering the widest range of language styles while at the same time preserving expressiveness at the fully abstract model level. The Synchronous Languages have an $RM\bar{C}$ semantics and use either a static or dynamic check for causality to surpass the RMC Barrier. This scheme is argued to be optimal in the sense that it is most expressive in the fully abstract model M and allows the widest range of flexibility in the non-abstract model M_δ without compromising the existence of the macrotime projection operator Π . This is roughly seen as: the R property allows Mealy-type behavior, the M property implies that the δ -time level always is projectable down to macro time (where there is single and unordered assignment on outputs). Finally, the loss of the C property in the general case is not of great import because the non-causal descriptions were never really constructible in the real world anyway. The semantic checks for RMC ensure that the described systems are fully expressive (R), have a sound and consistent interpretation in macrotime (M) and can be reasonably implemented in the real world where time only moves forward (C).

1.5.2 System Description Languages

This background then allows for an analysis of system specification languages from a semantic perspective. Such is undertaken in Chapter 6. While it is not feasible or even necessarily productive to classify *all* previously proposed languages in the framework here. In fact many proposed languages were never concretely enough specified to make such possible and indeed many have changed their meaning and intended use over time through subsetting and extension. What can be done however is to sort the languages according to their intended use and the means by which their semantics is given. This is shown to clarify the language design problem to the extent that only the languages designed with a semantics-based analysis of program behavior in mind have been well enough specified to be analyzed in the framework here. This is actually not surprising since simulation-oriented images are generally tuned towards convenience and a particu-

lar simulation algorithm while synthesis-oriented languages are shown to be generally tuned to some key set of data structures which enable the key operations of synthesis. Such classifications are useful for understanding the language design problem relative to a particular application focus. The classification scheme also exposes the inappropriateness of using languages with discrete event semantics such as VHDL [384] [387] and Verilog [570] for system specification.

However despite the overwhelming sense of their inappropriateness, there is an abiding popular interest in using the industry standard simulation-oriented HDLs as system description languages for all the uses: ‘true’ execution via simulation, specification for synthesis and symbolic execution via (formal) verification. The exact reasons for this interest are certainly beyond the scope of this investigation. There are numerous social and economic factors involved in the adoption of such design technologies not the least of which is an immense investment in personnel training and implementation infrastructure for the standard languages. To this end much of Chapter 6 is oriented at analyzing the standard hardware description languages in the framework developed here. In particular, the discrete event paradigm of VHDL and Verilog are shown to have a *three-level* time, the familiar macro-time and δ -time as well as the new η -time. This is markedly different from the two-level structure which is commonly taught for the semantics of discrete-events. As such discrete-event semantics is $R\bar{M}\bar{C}$ and $\bar{R}_\delta M_\delta C_\delta$. This implies that discrete-event semantics fails to have a well defined macrotime projection operator Π and, in turn therefore, there is no fixed point formulation for $F\{Q\}$ and $B\{Q\}$ (or $f\{q\}$ or $b\{q\}$). This means that substitutability fails to hold for discrete event HDLs. In a strong sense this failure makes their use impossible in the formal setting absent some sort of semantics-based subsetting.

To this end a subset of VHDL, Synchronous VHDL [47], which preserves the modularity property is studied within the framework developed here. This subset re-arrives at an

RMC semantics and comes as close as possible to establishing a subset of the discrete event paradigm where the diagram of Figure 1-2 commutes. Unfortunately, this experiment is only partially successful because of some minor, but crucial, aspects of the event-driven nature of VHDL semantics. Among these are that discrete event semantics is based on *events* and not on assignments. Events are changes in value while assignments establish the definition of a variable in its domain. Secondly, the three levels of time bring an intrinsic confusion between state variables and output variables in the microsemantic analysis. This added level of complexity contributes to the failure of substitutability for the semantics of discrete events.

Even if the semantic problems could be completely ironed out satisfactorily, there remains a language style aspect having to do with the flat process model of the discrete event HDLs, namely that structural aggregation can be hierarchical (*e.g.* the instantiation of an entity) but behavioral components must be flat (*e.g.* a process is a single thread of control). This lack was pointed out in the earlier work which defined Synchronous VHDL [47]. With the benefit of more experience there now exists a proposed language extension, the SpecCharts [277], which allows for the compact and elegant expression of hierarchical behavioral specifications. The potential to marry the Synchronous VHDL subset and the SpecCharts language extensions is proposed as a means for acquiescing to the broad-based desire to use an industry-standard HDL at all costs while at the same time addressing the practical need for a language with hierarchical behavioral definitions which built upon a mathematically sound semantic model.

1.6 The Nondeterministic Abstract Machine

The ideas of computational semantics, microsemantic analysis, RMC Barrier theory and the lessons learned from examining both applied semantics and previously-defined system description languages form a guide to the design of languages based on synchronous

semantics. The culmination of this work is the definition of the Nondeterministic Abstract Machine (NDAM). It is an instance of an instruction-oriented operational semantics with the synchronous properties:

- It is a non-abstract image semantics with microsteps defining a “ δ -time” within a step.
- The definition of a step is the fixed point of the microsteps across δ -time.
- The semantics is intrinsically RMC .
- The RMC Barrier is addressed by admitting only descriptions where C holds.

Additionally the representation has several domain-specific features which make it extremely convenient for practical implementation and formal verification. In particular, the NDAM has an operational definition which affords a compact and efficient runtime implementation scheme. Such a runtime system has been constructed. Also of interest is the allowance for the so called “selection nondeterminism,” which is shown to be compatible with synchronous semantics (*i.e.* the definition of concurrency is *not* based on a non-deterministic interleaving of independent threads). Nondeterminism of this sort is necessary in the formal verification context where it is used as a behavioral abstraction mechanism in conjunction with fairness constraints. The central design goal behind the NDAM is its use as an intermediate representation in the compilation of synchronous language with an imperative style. To this end, compilation schemes for Esterel and Synchronous VHDL have been constructed.

1.7 Review

It is the contention here that full abstraction is too great a price to pay in the design of system description languages and their semantics. Further that there exist non-abstract semantics which are substitutable for the fully abstract one but which are simpler and more convenient for use in formal verification computations. That is, the non-abstract semantics are simpler in the sense of it being easy to observe in S_δ the sense of a direct analogy between the language L and an operational view of the denotational model M_δ .

Such cannot be said of the fully abstract semantics S or its denotational model M . A further thread of argument in this work is that the very abstractness of a fully abstract semantics precludes any deeper insight of how system description languages ought to be constructed in the first place. By identifying the conditions when a non-abstract semantics can be substituted for the fully abstract case and vice versa gives a more or less direct prescription for the semantic features that ought to be in a well-designed system description language.

2 Semantics and Models

One might well wonder why a language semantics is needed at all. It is entirely reasonable to adopt the philosophy that a properly designed programming language should be “intuitive” in the sense that there should be no need of extra artifices for specifying its “meaning.” In the realm of algorithmic sequential programming languages where the underlying model of computation is widely understood and reference implementations exist, the issue of semantics is mostly one of recording implementation freedoms and constraints in a common notation.¹ Where the underlying model is part of the design problem as well, there is an overwhelming need to describe not only the model, but also how programs denote structures in that model. So, from one angle the issue of language semantics is an issue of specification: a semantics defines what computation is being described by the language.

From another angle, one can take a mathematical view of the situation. In that light, there are two ways that one can consider any notation, programming languages included. The first way is as textual or pictorial symbols to be manipulated according to a set of grammatical rules. Seen in this way, languages are *free objects*, having no interpretation or associated meaning. The only sensible distinctions that can be made in such a regime are between (grammatically) correct programs and those with syntax errors. The second con-

1. The recent definitions of Scheme [617] and Standard ML [537] can be said to fit this description.

sideration is one in which an interpretation in an underlying mathematical structure. is associated with the symbols. With such a structure, one can make model-theoretic statements about programs such as: two programs *are* the same if they *do* the same thing or the set of behaviors of one program is a superset of the behaviors of another. The *semantics* of a programming language is the means by which the association between the tangible representation and the mathematics is established. The underlying mathematical structure is called the *semantic model*.¹

The semantic model of a programming language is a fundamental part of its design. It lies at the base of all the possible design decisions about the language. This is particularly true in the case of verification where the semantic model's theoretical properties are what is declared by programs in the language and what is manipulated by the verification algorithms. This view of the relationship between programming languages and a given semantic model is illustrated in Figure 2-1.

This chapter focuses on two aspects of programming language design which are directly related to semantic models: the methods for giving semantic definitions and the range of possibilities for semantic models. The purpose of this survey is twofold. The first purpose is of course that of illustrating the diversity which is possible in semantic models. At the same time however, the argument presented here is that this diversity is illusory because despite the range of semantic model formalisms, for the class of synchronous finite state systems, semantic models are fundamentally defined by states and transitions between them. In particular behavior over time is fundamentally defined by a transition relation.

1. Some authors use the two terms *semantics* and *semantic model* interchangeably. Where no confusion may arise, this convention is adopted here as well.

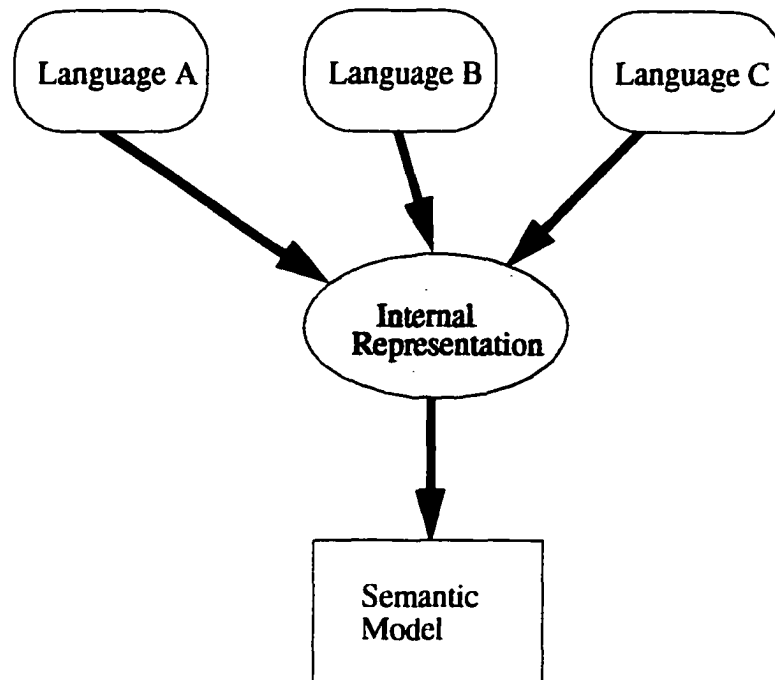


Figure 2-1. The Three-Phase Construction of a Language and its Semantics

2.1 Semantic Specifications

With our understanding to date, the means for specifying the semantics of computing systems can be loosely organized into three complementary approaches: axiomatic semantics, denotational semantics and operational semantics. Each of the methods was developed to address some particular need in specification and this background has colored and to a certain extent, limited the application of the pure forms of the basic specification technique. Recent trends in semantic specifications have addressed this issue by blending the strengths of one or more of the pure approaches in an effort to synthesize a new approach with the strengths of all and the weaknesses of none. The computational semantics approach presented in Chapter 3 of this work can be seen in that light.

2.1.1 Axiomatic Semantics

One of the first methods for specifying the meaning of programs was proposed indepen-

dently by Floyd [263] and Hoare [352]. Later presentations¹ introduced some modifications, however the method has collectively become known as Floyd-Hoare Logic [300]. The essential idea of the method is the association of preconditions P and postconditions Q with each statement or step S in the language. The set of axioms for the language is given in the form $\{P\} S \{Q\}$ for every S . The axioms are accompanied by a set of deduction rules that define how the constructs of a program relate the predicates P and Q . A Floyd-Hoare logic is a first-order system of logic; it allows quantification over elements (but not sets of elements) within a variable's domain. It is considered an exogenous system of logic because program statements S are considered on a par with the formulae P and Q .²

The strength of the axiomatic method is that it focuses exclusively on the issue of proving properties about programs, leaving the finer details of value representation unspecified. The method focuses on relating representations of truth *before* S is executed to truth *after* S is executed. In this sense each axiom can be seen as an abstract representation of a transition relation. In fact, the formal definition of the precondition-statement-postcondition structure $\{P\} S \{Q\}$ is given in relational form as $(\gamma_i, \gamma_{i+1}) \in Op[[S]]$ for the state configuration $\gamma = \langle s, c \rangle$ denoting the pairing of the program variable state predicate s and the continuation c . The structure $Op[[\circ]]$ is called the *operational transition relation*. That the axiomatic behavior is given in terms of a transition relation is a point that is returned to later in this section.

The method has a number of weaknesses, among them being that the representation of

1. For example, Dijkstra's weakest preconditions calculus [237], Pratt's dynamic logic [603] or Boyer and Moore's computational logic [103] among many others. Cousot [213] reviews these as well as numerous others.

2. In an *endogenous* system of logic such as Temporal Logic [599], the program is a completely separate conceptual entity. Reasoning about the program must be done through the artifice of predicates about its internal state. A prototypical example predicate is one that makes a claim about the value of the program counter, e.g. $pc = L(0x1A3F)$.

side effects is problematic in the axiomatic method. While the direct side effects of the statement S are represented explicitly by its axiom there may be other variables affected by the execution of S that aren't explicitly represented in its axiom. Such is the case where S is an abstraction of a larger computation. A prototypical example in this regard are global variables which are modified by a procedure call. This same problem arises when the basic sequential axioms are extended to support nondeterminism or concurrency. A second detriment is that the method focuses on the halting aspects of the computation. It distinguishes total correctness which shows correctness and termination from partial correctness where the correctness is contingent upon termination which unproven.

At a more fundamental level however, the axiomatic method is problematic because it does not relate programs to their denotations in any mathematical model. This lack can be seen in constant focus on the soundness and completeness of the logical theory arising from the axioms and deduction rules. Cousot [213] reviews of the results on the soundness and relative completeness under various domain systems. Interestingly, contrary to viewing this as a flaw, Hoare considered the domain underspecification issue to be a strength rather than a weakness because it allowed for the domains of variables to be made more concrete at some later point [352].

2.1.2 Denotational Semantics

A second approach to language semantics that moved to address the model association deficiency is the denotational semantics method proposed by Scott and Strachey [640] [642] [528] [673]. The meaning of a program in the subject language is identified through an associated with elements of an appropriate space of functions computable in Church's λ -calculus [174]. The appeal of the λ -calculus is that it is said to be universal in the sense of having the power to describe any computable function.

A language semantics is given in terms of a notation that equationally associates syntactic structures in the language with valuations on an underlying domain described by expression in the λ -calculus. An effectively separate body of theory¹ was developed by Scott that proved the existence of a model for the λ -calculus: $\wp\omega$, the power set of the natural numbers.² The original presentation called the notation LAMBDA and its denotations in $\wp\omega$ are actually simple enough that they can be stated in a few lines:³

$$\begin{aligned}
D\llbracket 0 \rrbracket &= \{0\} \\
D\llbracket x+1 \rrbracket &= \{n+1 \mid \forall n \in x\} \\
D\llbracket x-1 \rrbracket &= \{n \mid \forall n+1 \in x\} \\
D\llbracket x \supset y, z \rrbracket &= \{n \in y \mid 0 \in x\} \cup \{n \in z \mid \exists m. m+1 \in x\} \\
D\llbracket x(y) \rrbracket &= \{m \mid \exists e_n \subseteq y. (n, m) \in x\} \\
D\llbracket \lambda x. E \rrbracket &= \{(n, m) \mid m \in E[e_n/x]\}
\end{aligned}$$

Here the denotations of syntactic structures in the LAMBDA language are given in terms of set-theoretic expressions. The conversion rules \Rightarrow_α , \Rightarrow_β , \Rightarrow_η and \Rightarrow_σ in the λ -calculus can then be understood in terms of relating the set denoted of the left-hand side to the set denoted by the right-hand side. For example $(\lambda x.M)N \Rightarrow_\beta [N/x]M$ denotes the relationship $D\llbracket (\lambda x.M)N \rrbracket \Rightarrow_{D\llbracket \beta \rrbracket} D\llbracket [N/x]M \rrbracket$. Computation by functions expressible in the λ -calculus notation is thus related to the manipulation of (infinite) sets through a transition relation.⁴

Though the LAMBDA notation and the underlying model theory are largely orthogonal

-
1. In fact Stoy [673] points out that either body of work can be understood independently of the other.
 2. The domain $\wp\omega$ could also be written 2^ω however that usage is not traditional. Further however, there are various flavors of powerdomain constructors [301]: the upper powerdomain $\wp_u\omega$, the lower powerdomain $\wp_l\omega$ and the convex powerdomain $\wp_c\omega$. These identify computations which may fail, may succeed or which are unique respectively.
 3. From Stoy [673], page 123.
 4. This offers the interesting possibility of analyzing computations in LAMBDA by constructing a data structure that allows for the representation of the sets denoted by λ -terms and the manipulation of them as the transitions denoted by reduction rules such as \Rightarrow_β . In a loose sense, the semantics of 2-adic integers, ${}_2\mathbb{Z}$, and the 2Z language, which is reviewed Section 2.3.2 can be seen as an application of this idea.

the two are intrinsically tied: the model-theory gives justification to the techniques and tricks-of-the-trade used in defining semantic structures (valuations, continuations, stores) in terms of the λ -notation. The semantic structures of interest in the original denotational approach were primarily deterministic sequential computations. Later additions to the theory produced *powerdomains* which model different kinds of bounded nondeterminism and *resumptions* which model concurrent interleaved execution [545] [309] [534].

In the denotational theory, the definition of a function is extremely basic, being simply a set of pairs relating inputs to outputs:¹ $f = \{ (m, n) \mid m = f(n) \}$. One could equivalently write $(m, n) \in f$ thereby expressing f in the style of an infinite transition relation. This is mostly a stylistic observation however because in the theory, all of m , n , the pair (m, n) and even f itself are shown to exist through an isomorphism as elements within $\wp \omega$ itself. Of deeper interest in the treatment here is the characterization of the semantics of programming languages in terms of the *computable functions* which necessarily entails notions of approximation and fixed points. These aspects are returned to in greater detail in Chapter 3.

One can trace a path of subclassification within theoretical semantics through Berry's stable functions [74], Vuillemin or Milner's sequential functions [709] [530] and finally Berry and Curien's sequential algorithms [80]. This last subclass of computable functions is interesting because the sequential algorithm model makes explicit the aspects of control thereby establishing a connection with an operational definition. In fact Berry and Curien point out that the class of semantic maps expressible in the sequential algorithm model can properly be called interpreters for their respective language. These developments produced finer subclassifications of the computable functions and reestablished a connection

1. This expression is a slight abuse of notation. It is substantially correct (*c.f.* Stoy [673], page 118) but glosses over some significant aspects in the definition of functions. This inaccuracy is remedied in the description of the finer properties of $\wp \omega$ in Section 2.2.6. That a function is but a map relating inputs to outputs is a sufficient characterization at this point.

with the operational framework.¹

The whole denotational theory rests on the use of computable functions. This has the unfortunate effect of preventing the use of nondeterminism in conjunction with fairness constraints to combat the state explosion problem. The technical aspects of $\wp \omega$ that prevent from it handling fairness are described in greater detail in Section 3.2. Intuitively however, the problem is that computable functions are defined as the upper bound of a series of finitary approximations and fairness constraints are fundamentally infinitary relationships: fairness is discontinuous in the infinite limit. Despite this fundamental model-theoretic limitation, the denotational theory is rich with techniques for specifying the meaning of languages and many of the techniques developed within it have been adopted in this work. This point is returned to in the next chapter.

The Oxford School

The original presentation of denotational semantics has become known as the “Oxford School.” In that style of presentation, the meta-language LAMBDA is very simple and the λ -notation is used directly. As the λ -notation is extremely low level, the specification of even simple concepts become quite voluminous. The Oxford School is characterized by the use Greek characters for variables denoting well-understood semantic structures such as environments (ρ), continuations (θ), and the like. Layered on top of LAMBDA is a sort of policy or style guide describing how semantic definitions ought to be written. Much of Stoy’s book [673] is devoted to explaining this style. In the realm of automation

1. In a sense one can phrase the need for these subclassifications as follows: not all computable functions can be computed by in a stable or sequential regime. Or: not all computable functions can be computed by the usual practical means.

For example in the case of sequential functions, Milner [530] showed that a model with objects which are not sequentially-definable is necessarily non-abstract. The trivial example cited is the “parallel or” function (*por*) which has the map $por(tt, \perp) = por(\perp, tt) = tt$ and $por(ff, ff) = ff$. The function *por* is not a sequential function even though it is a computable function. Similar examples exist for stable but non-sequential functions and for computable but non-stable functions [74].

of the denotational method, various subsequent investigations gave conditions for the identification of “obvious” semantic concepts such as global variables and stack-based environments [638] [594] in a denotational semantics. Requiring major algorithmic analysis to identify these items represents a serious limitation of the pure approach.

Two-Level Approaches

In response to this limitation, a number of two-level approaches were proposed [558] [594]. These attempted to separate the static semantics, structures that could be found at compile-time, from the dynamic semantics, structures that must exist at runtime. Among these can be distinguished the *Abstract Semantic Algebra* (ASA) approach of Mosses [543] [544]. Central to the ASA approach is the observation that standard denotational semantics descriptions in the λ -notation are too concrete because they intertwine the details of the model of computation with the actual behavior of the program. Thus, implementations resulting from such descriptions tended to look like interpreters for the λ -calculus more than anything else [620] [621] [673].¹ It is interesting to note that even applications of standard denotational semantics techniques to the domain of hardware design have succumbed to this effect; Johnson’s language Daisy [407] and later projects [408] or Leeser’s language HML [466] [467] being most obvious in this regard.

In response, Mosses proposed *Action Semantics* as the notation for abstract semantic algebras. There the atomic actions are interpreted as consuming or producing values or of having side effects on another structure. The details of the underlying model are organized and abstracted onto the (finite) set of actions which then become the sole focus of discourse. The set of actions is chosen to expose the fundamental concepts of the subject language. In a sense the actions can be seen as the rudimentary notion of an abstract instruction.

1. In Stoy [673] the particular comments in Chapter 13, page 321 on how a non-standard semantics is defined operationally on an abstract machine

High Level Semantics

Taking the two-level approach of action semantics one step further, Lee [473] proposed *High-Level Semantics* as a means of preserving the underlying mathematical basis of denotational semantics. His goal was to use the denotational approach while at the same time introducing enough structure that automated code generation techniques could be applied to the ordered elements of the action algebra. The result was the MESS system which demonstrated that it was possible to automatically produce compilers from semantic specifications on a competitive basis with hand-coded versions.

Of interest here is Lee's notion of *semantics engineering* which corresponds loosely to the notion of computational semantics to be defined in Chapter 3. He posited that there was a class of endeavor combining the study, relative to semantic specifications, of engineering, philosophy and implementation. He interpreted engineering in the sense of the software engineering of semantic specifications, philosophy in the sense of understanding what is good, clear and proper in a semantic specification,¹ and of course implementation in terms of the pragmatics of building compilers for languages based on formal specifications. The MESS system was to be interpreted as a preliminary framework for conducting investigations in this field.

Also of interest to this work is Lee's two-level structuring of the semantic specification of languages by a *Principle of Separation* into a macrosemantics and a microsemantics. The macrosemantics described the compile-time relationships while the microsemantics governed the runtime aspects. Lee's significant contribution was that the macro-to-micro

1. It should be noted that from an adoption standpoint, computer languages are a "lifestyle issue" (*c.f.* the introductory comments in Stoustrup [672]). As such, their analysis and criticism is often reduced to extended polemics [279] or theatrics [Coo95]. Reasoned approaches to analysis and criticism have included the extended thematic comparison and contrast essay [348], survey and categorization in a universal framework [635], exposition of design decisions and evolution in historical context [672], information model analysis [304] [305] or the application of complexity measures from information-theoretics [671] or from a socio-engineering perspective [475].

boundary was governed by the signature of the action domain algebra such that different microsemantics could be substituted depending on the implementation domain. The tangible representation of this boundary layer was an internal representation consisting of Prefix Operator Terms (POTs). The use of the prefix-based form in the internal representation set the stage for the use of a Graham-Glanville code generator [289] [290] as one of the possible microsemantics. The scheme is shown in Figure 2-2.¹

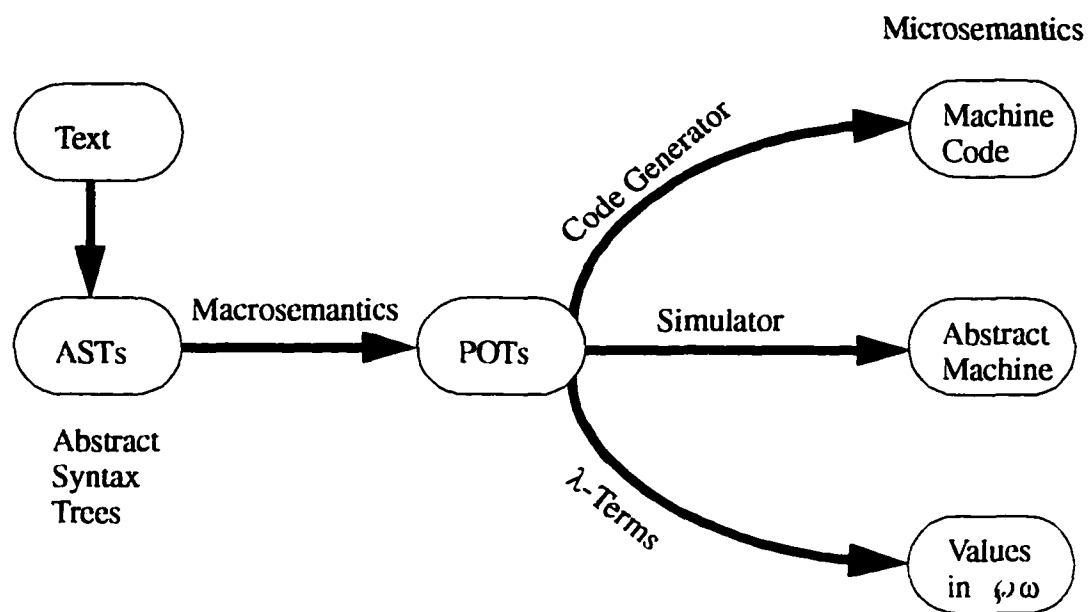


Figure 2-2. The Two-Level Approach with Prefix Operator Terms

2.1.3 Operational Semantics

The operational approach specifies the semantics through a structure and a set of rules for computing on that structure. The finer distinctions are made along the lines of the concreteness of the structure and the rules. On the one extreme is the interpreter approach where the structure is an actual machine and the rules for computing are called instructions. On the other extreme the structure is a system of symbolic terms and computing is

1. Adapted from Lee [473], page 70.

represented by a set of rules that rewrite the terms. Both extremes are in active use though the more abstract term rewriting approach tends to be applied in settings where a mathematical basis is needed (the proofs being given by structural induction over terms).

Operational language definition has been in use since the earliest days of programming. In some sense it is the easiest to understand as the machinations in the source language are defined on a low-level machine that couldn't possibly be misunderstood. Operational definitions have been used as the primary definition mechanisms for a number of "real" programming languages. Oft-cited examples are Algol68 [704] and PL/1 [716].

Interpreter-Based Semantics

The normative use of interpreters is the definition of some sort of standardized abstract architecture that is suited to the underlying goals of the language. Such goals have included portability as is the case with Pascal [723] or more recently Java [676], or pedagogical rigor as is the case with Landin's SECD machine for the λ -calculus [460] or Cousineau *et al.*'s Categorical Abstract Machine (CAM) [217] for ML [675]. At the extreme of the interpreter-based approach are the so called meta-circular interpreters where the definition of the interpreter (defining the meaning of the source language) is given in terms of the source language itself. Examples of this can be seen with Lisp [511] and later Scheme [668].

The major criticism against the exhibition of an interpreter as the semantic basis for a language is that there is little or no connection between the source language and any mathematical model of computation. In defense of the method one can point to Landin's SECD or the more recent CAM as instances where there is a strong connection with an underlying body of theory which has been abstracted and made operational. In fact, Lee's POTs can be seen in this same light, being a sort of halfway point between the source language and an abstract machine that interprets POTs. The "POT machine"¹ is operationally

defined yet is associated with Scott's theory of monotonic continuous functions over $\wp \omega$ by virtue of the microsemantic definition in terms of the λ -notation.

In defense of the interpreter-based approach, interpreter-defined languages have been used as the basis of specifications for hardware synthesis. Johnson [407] used this approach in defining Daisy as a notation for specifying the behavior of systems for synthesis into digital logic. In that work, systems of recursive applicative functions are transformed to suppress the explicit recursion through a process called *lifting*. The lifting transform associates the recursive calls with latches thereby separating the current invocation of the function from the previous invocation; the value from the previous invocation is stored in the latch.

Natural Deduction Semantics

At a more abstract level the operational approach has taken on the aspect of term rewriting. In Kahn's [416] natural deduction semantics of programs, a relation $E \Rightarrow V$ which associates program fragments E with values V in some domain.¹ The operational semantics axiomatizes the relation \Rightarrow by expressing the computation in the form of deductive inference rules. To express that E evolves into E' the inference notation is written as $\frac{E \Rightarrow V}{E' \Rightarrow V}$ meaning "if we wish to infer $E' \Rightarrow V$ then we must first prove $E \Rightarrow V$." Here E is at once the program state and the program continuation as appropriate to the task at hand. This form has become known as the "big-step semantics." It has the advantage that terms E are associated with values V in an underlying domain.

An earlier version of this method proposed by Plotkin [596] had the evolution of a program be given according to pure rewrite rules of the form $E \rightarrow E'$. This form has become

1. Yes, the joke has already been made. *c.f.* Lee [473], page 71.

1. In more complete presentations the relationship is stated as $W \mid E \Rightarrow V$ characterizing the state(s) W in which the rule applies.

known as the “small-step semantics.” The definition of the relation \rightarrow suppresses the association with any underlying value domain. As such some have questioned the mathematical basis of the method. On the other hand, the relation \rightarrow defines how the computation must proceed: a computation is legal if and only if the state transition from E to E' is defined in the relation \rightarrow . Said written differently, the computation is legal if and only if $(E, E') \in \rightarrow$.

Abstract Machines and Internal Representations

Further work in this area has concentrated on automating the operational method. One of the earliest was Despyroux' Typol system [231] which was able to execute the core language of ML given a description of its operational semantics. Hannan and Miller [329] defined conditions under which the big-step rules $\frac{E' \Rightarrow V}{E \Rightarrow V}$ can be transformed into the small-step form $E \rightarrow E'$. In their technique the big-step semantics defines the meaning of the program in a mathematical sense. The small-step form constitutes an operationally-defined target language which is as an instance of Wand's [714] semantics-directed machine architecture. This view has been accepted to the extent that a Plotkin small-step semantics is often referred to as but an elegant and expressive shorthand for an abstract instruction set or Internal Representation (IR) [523]. Extending this thread, Hannan [328] has since provided an automated method for extracting the IR from the small-step operational semantics description. The Non-Deterministic Abstract Machine presented in Chapter 7 can be seen as an instance of such a semantics-directed architecture.

2.1.4 Focus

The three genres of semantic specification on the surface seem to have a very disparate nature. However, as the preceding sections have shown, under the surface, there are substantial similarities. In a sense, this is as it should be since all three address the same fundamental problem. The similarities are reviewed in this section along with two comments

on the lessons that have been learned in developing and using semantic specification techniques.

One of the first similarities that can be seen is the cross-use of definitional mechanisms among the three. For example, within the axiomatic paradigm one speaks of defining the deductive axioms that express computation in terms of an operational transition relation. Within the operational paradigm, one speaks of axiomatizing the relation \Rightarrow thereby expressing operational semantics in terms of axioms! In addition within the denotational method, the semantics of λ -computations are expressed in terms of rewrite axioms on the lambda notation. These rewrite rules can be at once considered the axioms of computation or instructions on a SECD-type interpreter machine.

At a deeper level however, the similarity among the three paradigms reaches even further. At a fundamental level, all three express behavior in terms of a transition relation in some way. An axiomatic semantics is said to define computations in terms of an operational transition relation, a denotational semantics defines computations in terms of functions as a set of pairs relating inputs to outputs and an operational semantics, in the small-step sense, defines computing in terms of a relationship \rightarrow describing how the system state evolves over time steps. This is more than just being a poetic observation or a commentary on the redundancy inherent in semantic formalisms. Relative to the class of systems of particular interest in this work, *i.e.* finite-state synchronous systems, it is significant because there exist powerful methods for representing and manipulating very large transition relations.¹ Indeed significant progress in automated techniques for verification based on the manipulation of transition relations. The application of these techniques is the subject of subsequent chapters.

1. The reference here is to Bryant's OBDDs and implicit methods. Exposition of their use in the verification setting is deferred until Section 3.3.4.

There are also some practical lessons that can be drawn from the range of semantic specifications. One of these is that semantic specifications for languages tend to be complicated (or more properly, they tend to become complicated as they evolve over time). Without some structuring in a software engineering sense, these specification can be so complex that it is hard or impossible for humans to comprehend them, let alone being processed by automated tools. This effect has been shown most forcefully by Lee. Fortunately, he also convincingly demonstrated how to remedy the problem by applying a bi-level specification approach. Thus lesson learned from high-level semantics is that “there should be a middle,” and that the intermediate level can be designed an intended use in mind. Lee’s intent was the application of a specific code generation scheme. The domain here is in the forward and backward image computations $F\{Q\}$ and $B\{Q\}$ used in formal verification by language-containment. The middle in this work is the Non-Deterministic Abstract Machine (NDAM) which is presented in Chapter 7.

The operational paradigm has drawn from the bi-level approach as well in attempting to automate the generation of the intermediate level. In focusing on procedures for deriving IRs from semantic descriptions the IR design problem has been transformed from that of a craft drawing from a body of known tricks and best-practices into a scientific or engineering endeavor. The focus on automatability here is therefore pedantic¹ with the interest being that the IR generation task *can* be automated rather than *how* best to go about that automation. The pedantic view being that a problem can only be said to be well defined when the correctness and optimality conditions are posed concretely enough that it can be subjected to an algorithmic solution.

1. This is to say that the Non-Deterministic Abstract Machine (NDAM) presented in Chapter 7 has the distinction of having been derived by a *manual* analysis of the problem domain. A comparison between the NDAM as presented and one derived from a procedure such as Hannan’s [328] has not been attempted. The NDAM design can however be seen as being justified in terms of matching and optimality criteria relative to the causally-interleaved MASS computation used in image and pre-image computations.

2.2 The Standard Semantic Models

Every scheme for semantic specification consists of two largely separable parts. The first is the part that is properly called the semantics. That is the notation used to identify aspects of the subject language and relate them to the semantic model. The second part is of course the semantic model itself. There are a wide range of semantic models each with its own relative expressiveness and associated proof procedures. Some of the more significant and relevant ones are reviewed in the following sections.

The semantic models can be loosely classified into two broad categories: standard models and nonstandard models.¹ This section and the next respectively review what can be considered the standard models and a few of the many nonstandard models respectively. Examining the similarities and differences across models is a useful exercise for differing proof procedures and implementation methods are implied by each. Such differences are especially true of the non-standard models which were originally proposed to exploit some specific expressiveness result or address some usage which could not satisfactorily addressed in the more traditional framework.

The following sections review five models which have been used as the theoretical basis for various verification theories: trace models, process algebras, Petri nets, Kripke structures and the ω -automata. In addition the universal domain $\wp\omega$ is mentioned for completeness in this context. Scott's domain theory is abstract enough and covers a wide enough range of applications that, for the purposes of this work, the denotational semantics forms more of a recipe for constructing domains with the appropriate internal structure than a direct representational paradigm. Indeed, the computational semantics

1. The term *standard model* is used here in the sense of being in common use within the semantics and formal verification literature. In the larger theoretical sense many of these models are thought of as *non-standard* because they are finite or their definition does not distinguish some interesting theoretical structure (such as linear versus branching time). In the following sections, such overloading is noted where confusion is possible.

presented in Chapter 3 draws directly on elements of this theory as it applies to multi-step transition systems with finite state. Each of the models covered in this survey has been the subject of extensive study in its own right so the focus here is on the comparison and contrast among them rather than an exhaustive enumeration of the theory or results derived from any. Of interest in comparison are the means by which each scheme represents state and state-to-state transitions. The highlight of this comparison is the observation that for practical purposes, all the schemes define state and state-transition systems either implicitly or explicitly. In contrast are the disparate means by which behavioral properties can be specified across multiple macro steps.

2.2.1 Trace Theory

Probably the most straightforward model is the trace theory. This model has the benefit of having a direct relationship with the intuitive operational behavior of the system at hand. Trace theory was first proposed by Brookes, Hoare and Roscoe [354] [118] [119] to provide a model for the Communicating Sequential Processes (CSP) [353] [355]. In that context the meaning of a CSP program is given in terms of the communications that it can perform as it progressed coupled with the set of terminations and failures that the program. Rem. van de Schnepscheut and Udding [619] and Dill [238] have applied trace theory to the modeling and verification of systems of asynchronous logic.

A trace is a set of finite or infinite strings that describe the behaviors of the system (events) as viewed externally. Typically the symbols in the alphabet of the strings are the input, output and synchronization events in which the system can engage. The attraction of a trace-theoretic description of behavior is that if the system is of finite state then the trace sets are guaranteed to be regular.

A trace structure is given by:

$$T = (\Sigma, I, O, S, F)$$

where:

Σ is the alphabet of all possible events,

$I \subseteq \Sigma$ is the set of input events,

$O \subseteq \Sigma$ is the set of output events,

$S \subseteq \Sigma^*$ is the success set describing good behaviors,

$F \subseteq \Sigma^*$ is the failure set describing bad behaviors.

The set of possible events is written as $P = S \cup F$. Trace structures are distinguished in that they identify not only the successful computations but also the failures. Two trace structures are considered equivalent if their success sets are equal *and* so are their failure sets.

A trace structure is called *failure-free* if its $F = \emptyset$. A trace structure can be checked for failure-freeness by inspection if F is given explicitly. The failure set is not explicit when a trace structure is given in terms of the composition of other traces. Composition over trace structures is defined by the intersections of the traces of both systems:

$$\begin{aligned} T_1 \parallel T_2 &= T_1 \cap T_2 \\ &= (\Sigma_1 \cup \Sigma_2, I_1 \cap I_2, O_1 \cup O_2, S_1 \cap S_2, (F_1 \cap P_2) \cup (P_1 \cap F_2)) \end{aligned}$$

The class of trace structures most often used in the modeling of systems are the prefix-closed trace structures. The prefix set $pref(R)$ of a set R is given by

$$pref(R) = \{x \mid \forall y \in \Sigma^* \wedge xy \in R\}$$

When the sets S and P are both prefix closed, a trace structure T can be interpreted as an accurate model of the possible evolutions of a system over time. A prefix-closed set contains not only the finite traces of behaviors of a system at time t_k but also records all the behaviors at instants t_j for $j < k$. With respect to the set P this implies that if there is a $y \in P$ then there must also be a $x \in P$ such that x can be completed to y . Prefix-closed

trace structures are fundamentally sets of finite strings and as such they cannot be used to model fairness or liveness properties. A second kind of trace structure is the so-called complete trace structures [702] which are defined over sets of finite strings and infinite sequences in terms of the mixed-regular sets.¹ For the most part however, trace structures are considered to be prefix-closed and defined over regular sets of finite strings.²

Dill's contribution was a notion of *trace conformance* which related the trace sets of one trace structure to those of another. The trace conformance relation $T_1 \leq_c T_2$ holds if $I_1 = I_2$, $O_1 = O_2$ and for all environments $E[\![\circ]\!]$ it is the case that $E[\![T_2]\!]$ is failure-free implies $E[\![T_1]\!]$ is also failure-free. A decision procedure for checking whether $T_1 \leq_c T_2$ is given by transforming the conformance problem into the problem of determining whether the composition of T_1 with the *mirror* of T_2 is failure-free. The mirror of a trace T^M is defined as:

$$T^M = (\Sigma, I^M, O^M, S^M, F^M)$$

where:

$$I^M = O,$$

$$O^M = I,$$

$$S^M = S,$$

$$F^M = \Sigma^* - P$$

Intuitively the mirror operator T^M transforms T into its own worst-possible environment: T^M accepts the extremes emitted by T and no more and produces the extreme accepted by T and no more. Using the mirror, the conformance relationship can be verified by check-

1. Dill [238] defines a mixed-regular set as the union of a *-regular and an ω -regular set: $\Sigma^\omega = \Sigma^* \cup \Sigma^\omega$. The properties of such structures are essentially those of the ω -regular languages. Kurshan [454] formalizes the obvious argument as to why any treatment handling infinite sequences (the ω -regular languages) necessarily subsumes one handling only finite strings (the *-regular languages).

2. To ensure that an arbitrary trace properly models the execution of something, some other closure properties are required as well: fusion closure, suffix closure and limit closure. *c.f.* Emerson [248], page 1014.

ing that the mirror-composition is failure-free. Failure-freeness in turn can be checked through a sort of reachability analysis.

It is natural to view the relation \leq_c as a sort of language containment. This is not quite an accurate picture because \leq_c is a pre-order not a partial order. This is because the relation \leq_c is not anti-symmetric: $(T_1 \leq_c T_2) \wedge (T_2 \leq_c T_1)$ does not imply that $T_1 = T_2$. Rather a weaker relationship holds, that of *conformation equivalence* which is written as $T_1 \sim_c T_2$. Loosely, conformation is a weaker relationship because trace sets have a lot of information in them that is wholly unrelated to the relationship \leq_c . An intuition of this distinction is that the extra information relates to differences in behavior under failure. The fact that trace structures are specified in terms of the success set *and* the failure set provides an extra level of expressiveness that is not found in other linear-time models. Dill shows how trace structures can distinguish between two branching-time structures that have the same success sets.¹ The two trees are distinguished by having different failure sets. That example is reproduced in Figure 2-3.

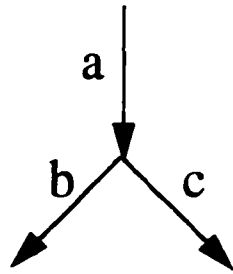
2.2.2 Process Algebras

The fundamental premise of process algebraic models of computation is that a *process* is a concept that is not understood well enough to be concretely characterized. In this view there is no general agreement on the model of what a process is. Instead the approach taken is to explain what a process *can do* and thereby derive some understanding of what a process *can be*. Process algebras are the means by which the operational behaviors of a process are characterized. A process algebra, as such, is a simple abstract programming language which describes the behavior of a process over time in much the same way that the λ -calculus is an abstract programming language. Terms in the process algebra denote processes themselves and the operators of the algebra define behavioral primitives such as

1. *c.f.* Dill [238], page 54.

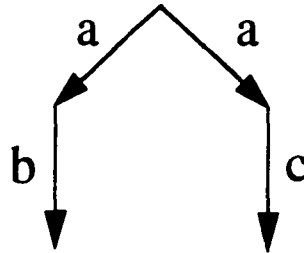
$$S_0 = \{ ab, ac \}$$

$$F_0 = (aa+ b+ c+ (ab+ ac)(a+ b+ c))(a+ b+ c)^*$$



$$S_1 = S_0$$

$$F_1 = F_0$$



$$S_2 = S_0$$

$$F_2 = F_0 + ab + ac$$

Figure 2-3. Branching-Time Structures Distinguished by Their Failure Sets

events, (parallel) composition, sequentality, hiding and renaming. For a process algebra, the existence of a finite set of process variables P_i is assumed as is a finite set of actions $\alpha_i \in Act$. The distinctions between the various process algebras comes in the allowed operators and their meaning. The syntax of a simple process algebra might look as shown in the table of Figure 2-4.¹ The semantics of process algebras is given by the possible rewritings of the terms in a process expression as shown in the table of Figure 2-5.

There are a wide variety of process algebras, each of which is oriented at highlighting some aspect of computation. Of these include:

1. As presented in Milner [534].

Syntax	Description
$\alpha.E$	Prefixing
$E_1 + E_2$	Summation, Choice
$E_1 \parallel E_2$	Parallel Composition
EV_L	Restriction, Hiding
$E[f]$	Relabeling by f
$\mu_j. \{P_i = E_i \mid \forall i \in I\}$	Recursion (return j th)

Figure 2-4. The Syntax of a Simple Process Algebra

- Algebra of Communicating Processes (ACP) [69],
- Algebraic Theory of Processes (ATP) [226] [343],
- Communicating Sequential Processes (CSP) [353] [119] [355],
- Calculus of Communicating Systems (CCS) [532] [533], SCCS and ACCS,
- CIRcuit CALculus (CIRCAL) [526],
- Esterel [79] [81] [295] [228],¹
- LOTOS [114] [397] [701],
- Meije [36] [227],
- The π -Calculus [535] [536].

At a fine-grained level, all these algebras are different enough that a unifying theory has not yet evolved [534]. Some algebras require an interleaving semantics, while others such as Meije or SCCS provide for a synchronous notion of time. Some algebras provide for synchronization among pairs of processes while others allow synchronization among sets of processes. Further distinctions are along the structure of the actions Act . For example in CSP, Act is merely a set of possible actions which occur singly, CIRCAL uses events

1. More recent developments have moved away from the original focus on the process algebra and event derivative interpretation to interpretations based on abstract machines [47] [246] and synchronous circuits [76] [257] [525]. A key issue with these more exotic interpretations is their equivalence with the behavioral semantics of the process algebraic interpretation.

Name	Rule	Meaning
ACT	$\frac{}{\alpha.E \xrightarrow{\alpha} E}$	After α , the system behaves as E
SUM _{j}	$\frac{E_j \xrightarrow{\alpha} E_j}{\sum E_{j \in \Omega} \xrightarrow{\alpha} E_j}$	The j th alternative is selected
COM0	$\frac{E_0 \xrightarrow{\alpha} E_0}{E_0 \parallel E_1 \xrightarrow{\alpha} E_0 \parallel E_1}$	The process on the left E_0 evolves
COM1	$\frac{E_1 \xrightarrow{\alpha} E_1}{E_0 \parallel E_1 \xrightarrow{\alpha} E_0 \parallel E_1}$	The process on the right E_1 evolves
COM2	$\frac{E_0 \xrightarrow{\alpha} E_0 \quad E_1 \xrightarrow{\bar{\alpha}} E_1}{E_0 \parallel E_1 \xrightarrow{\tau} E_0 \parallel E_1}$	Both processes E_0 and E_1 synchronize
RES	$\frac{E \xrightarrow{\alpha} E}{E \setminus L \xrightarrow{\alpha} E \setminus L} \quad (\alpha \notin L \cup \bar{L})$	The restriction of the system hiding L
REL	$\frac{E \xrightarrow{\alpha} E}{E[f] \xrightarrow{f(\alpha)} E[f]}$	Relabeling E by f
REC	$\frac{E_j \{ \mu_j \cdot \bar{p} E / \bar{p} \} \xrightarrow{\alpha} E_j}{\mu_j \cdot \bar{p} E \xrightarrow{\alpha} E_j} \quad (j \in \Omega)$	Recursion over the processes, returning the j th element of the fixed point

Figure 2-5. The Operational Semantics of a Simple Process Algebra

which are sets of actions while Meije's action domain is a commutative monoid¹ where $\circ \times \circ$ denotes synchronization and τ is the unit.

1. From topology [427]:

A *monad* is a structure $(M, \circ \times \circ, \tau)$ over a set M where $\circ \times \circ$ is associative and τ is the unique null element.

A *group* is a structure $(G, \circ \times \circ, \circ^{-1}, e)$ over a set G where $|G| \geq 1$, $\circ \times \circ$ is associative, e is the unique neutral element and $\forall x \in G. x \times x^{-1} = e$; a group is *commutative* or *abelian* when $\circ \times \circ$ is commutative.

Of interest here is the ability to characterize process algebras in terms of labeled transition systems (LTS). For every process algebra the operational rules, such as those given in Figure 2-5, are understood as a labeled transition system:

$$LTS = \left(Q, Act, \{ \xrightarrow{a} , a \in Act \} \right)$$

where:

Q is a set of states,

Act is a set of actions,

$\xrightarrow{a} \subseteq Q \times Q$ is the action relation.

The action relation can also be written in the form of $\rightarrow \subseteq Q \times Act \times Q$ which is the more familiar transition relation. In the general case an *LTS* is not finite because Q grows under \xrightarrow{a} . Conditions when an *LTS* is finite have been developed [708] and this observation forms the basis for the use of symbolic techniques on process algebras [260] [243].

A second point of semantic interest is the notion of equality which is in some sense fundamental to the process algebra approach. The algebraic approach avoids the explicit mention of an underlying model of behavior other than the operational rules of the abstract language. As such, some other means must be used to determine if two processes are equivalent or whether one is more deterministic than the other one. This is accomplished through a notion of *testing equivalence* or *observation equivalence* which relates processes if they can participate in the same set of events *and* subsequently have the same potential for entering deadlock.

The various process algebras distinguish such concept as the necessity and eventuality of termination and deadlock. There are various notions of testing equivalence that can be defined. at the finest level is Milner and Park's *bisimulation equivalence* [580] [531]. A bisimulation B is a relation on the states of two systems P and Q which is the coarsest partitioning of the states of the two systems such that both partitioned systems can be said

to preserve the behavior of the other:

$$B \subseteq 2^P \times 2^Q$$

$$\forall \alpha \in Act$$

$$P \xrightarrow{\alpha} P' \Rightarrow \exists Q'. Q \xrightarrow{\alpha} Q' \wedge (P', Q') \in B$$

$$Q \xrightarrow{\alpha} Q' \Rightarrow \exists P'. P \xrightarrow{\alpha} P' \wedge (P', Q') \in B$$

The attraction of bisimulation is that it offers an automated reduction method whereby a partial view of the global system can be defined. Presumably a property can be characterized by some very simple system Q which can be determined by inspection to test for the property of interest. A verification by bisimulation tests that the system P is bisimilar to the property Q . Verification tools such Aldébaran [258], Argonaute [501], Auto [99], The Concurrency Workbench [198] [199], Xesar [623] and others provide automated means for verification from descriptions given in various process calculi.¹

2.2.3 The Petri Net as a Model

Petri nets [584] [618]² are often touted as an representation of behavior which is orthogonal to that presented in automata theory. This, because of their distributed representation of state and activity. The claim is that this distribution allows them to express the so-called “true” or distributed concurrency as opposed to the “simulated” or interleaved kind. In this sense there is a strong distinction because distributed concurrency is a fundamental property of the net semantics and it is distinct from the derived concurrency induced by nondeterministic interleaving. Complicating matters also is the fact that there are a wide range of subclassifications and analyses which use the Petri net formalism. Only a few of the

1. This is not to suggest that the problem is currently considered to be “solved.” Bisimulation-based verification suffers from state explosion as do all state exploration methods.

2. Petri proposed the net representation as a means of constructing an asynchronous concurrent simulator for a Turing machine [585] [586]. The nets used in the original presentation are now seen as being restricted to safe nets only. As such, the original references are only of historical interest.

most relevant can possibly be related here.¹ Of primary interest is the observation that, Petri nets, when restricted to have finite state, can be seen as an extension of classical automata theory, albeit an interesting extension wherein the state of the automaton is disaggregated and a fine structure on the automaton's transitions can be exploited.

In this development, the Petri net theory is best understood by distinguishing the syntax or uninterpreted structure of the net from the semantics of net execution. The syntax of the net Σ is given as a tuple:

$$\Sigma = (S, T; F, K, M_0, W)$$

where:

S is a set of places,

T is a set of transitions,

$F \subseteq (S \times T) \cup (T \times S)$ is the set of directed edges in the net,

$K: S \rightarrow (\mathbb{N} \cup \{\omega\})$ defines the capacity (in tokens) of each place,

$M_0: S \rightarrow (\mathbb{N} \cup \{\omega\})$ defines the initial marking and $\forall s \in S. M(s) \leq K(s)$,

$W: F \rightarrow (\mathbb{N} - \{0\})$ defines the weight of each edge.

Within this broad definition there are many subclasses of Petri nets, each distinguished by some interesting property. A few of these classes are useful for modeling systems which are constructible in the real world, that is finite state systems.

A net is said to be *safe* when $\forall s \in S. M(s) \leq 1$. A net is said to be *live* when every cycle in the net has at least one marked place. In a large class of cases, the capacity of each place is considered to be unbounded and the edge weighting is irrelevant. Such a restricted net definition is given by the restriction to $\forall s \in S. K(s) = \omega$ and $\forall f \in F. W(f) = 1$. Of interest in the presentation here are such nets. Greibach² has shown that such a net has an equivalent representation as an automaton:

1. The body of Petri net theory is *extremely* voluminous. The classic introductory works in the field, Peterson [584] and Reisig [618] list bibliographies of 300 and 500 entries. A recent compilation [598] lists over 4100 references (reported in Kurshan [454], page 7).

2. From Greibach [303], page 320 as presented in Taubner [682], page 97.

$$N = (S, T, M_0)$$

where:

S is the set of places,

$Act \subseteq 2^{(S \times T) \cup (T \times S)}$ is a set of (aggregate) actions,

$T \subseteq 2^S \times Act \times 2^S$ is a set of (aggregated) transitions.

For a safe net, this definitional form makes it clear that syntactically and operationally a Petri net is a form of automaton¹ and edge elements $a \in Act$ are called *steps* which are said to be made up of unordered *microsteps*. It may be that within a step, two or more microsteps are enabled or that an ordering among the microsteps is required for the net to remain safe. Such situations are called *conflict* and *confusion* respectively.

The net is said to have a *partitioned state* in the sense that its state is defined by the markings of its places, places in turn denoting the holding (or not) of some condition. In the automata-based view, the net is said to have a distributed transition relation which is defined by the aggregation of edges (or firings to use the net parlance) that are involved in some $a \in Act$. Taubner [682] used Petri nets of this form to show transformations from the CCS and TCSP process algebra formalisms. This established the limitations on the representation of programs in those formalisms in terms of automata and nets.

The Petri nets are not really a model *per se*; they are a (network) structure. As such, semantics of Petri net execution must be defined by other means. Primarily the semantics is defined operationally through a *firing rule* that describes when a transition is enabled based on the presence of tokens at its inbound places. Other forms of semantics have also been attempted. Nielsen, Plotkin and Winskel [560] showed a relationship between Scott's topologically-defined domains and event structures on a class of Petri nets called occurrence nets. Mazurkiewicz developed a semantics for Petri nets based on traces [509]

1. The automaton need not be a finite automaton as the number of tokens in a place is unbounded in the general case.

[510].

There is active interest in Petri nets in formal verification where they fall under the rubric of *partial order methods*. This is because events that are not causally related are not associated in the formalism. The partial order is the causal relation among events with $E_1 < E_2$ if and only if E_1 is said to have caused E_2 . The interest in partial orders lies in the possibility of exploiting the independence of concurrent activities under composition to combat the state explosion problem. There is a fairly large body of work on the abstract semantics of partial orders [457] [605] [722]. More recent work has concentrated on practical applications. In one such case a reduced automaton was extracted from a Petri net subclass called a safe P/T net. This automaton was then used in a trace-based language containment [292] decision procedure. In another example, Neilsen *et al.*'s unfolding procedure was extended with stronger truncation conditions and used to verify a distributed mutual exclusion protocol [518]. Both cases exploited the partial order nature of the Petri net to avoid state explosion.

2.2.4 The Kripke Structure

The temporal or Kripke structure [443] was presented in Chapter 1 as $M_K = (Q, T, \Phi)$ with a set of states Q , a transition relation T and a labeling Φ which associates a set of atomic propositions with each state. In the strictest sense, a Kripke structure is not a model because it does not distinguish between two theoretically interesting interpretations of time: linear time and branching time. In linear time each time point contains exactly one successor point. Linear time corresponds to a description of a computation in terms of an infinite string. Under branching time on the other hand, each time point has two or more successors and computations are interpreted as infinite trees.

In the context of verification, and in particular the verification of finite-state systems, a Kripke structure is viewed as the “wrapped up” and finite representation of the actual

model. The Kripke structure can be seen as a *generator* for elements in the temporal model.¹ It also has a convenient and obvious analogy with the constraints on the systems being modeled: finite state reactive systems in hardware and or software. The strong point of interest here however is that proofs carried out on the structure, interpreted as a non-standard finite model, are known to be valid on the infinite model.² Both the linear time and the branching time view have their place. At one point there was debate as to which was better [458] [600] [250] though the current view is that the two views are complementary, each offering its own advantages and disadvantages [188]. The fact that a Kripke structure can be used as a generator for both kinds of models has been an important feature in practical verification schemes [359] [362].

A Kripke model is a so-called “possible worlds” semantics. In such a system, truth is dependent upon the world in which it is evaluated. Further, the truth of a formula φ denoting truth in a world q_i is understood to depend, in a syntax-directed way, on the truth denotations of subassertions in other worlds accessible from q_i . In practical applications the “worlds” are really the states of the system, and each state $q \in Q$ is its own separate world. The transition relation T defines the accessibility relation between different worlds. Distinctions between various classes of Kripke models are then made according to what kinds (or modes) of truth are representable in the model.

2.2.4.1 Linear Time

A linear time model for a Kripke structure $M_K = (Q, T, \Phi)$ is given by:

-
1. The term *Kripke model* is often used interchangeably where the structure of time (linear or branching) has been stipulated.
 2. A survey of the theoretical basis behind this is given in Thomas [684] and Emerson [248].

$$M_{S1S} = (Q, \sigma, \Phi)$$

where:

Q is a set of states,

$\sigma: \mathbb{N} \rightarrow Q$ is an infinite sequence of states,

$\Phi: Q \rightarrow 2^{AP}$ is a labeling of states with atomic propositions AP .

In the linear time framework, time is of course discrete and each point in time has a one-to-one correspondence with some natural number. Each natural number is associated with some state $q \in Q$ and therefore elements in the model are infinite strings of states $\sigma = q_0q_1q_2\dots \in Q^\omega$. At an even more primitive level, a linear time Kripke model is given a model-theoretic interpretation in terms a system,

$$\alpha = \left(\omega, 0, succ, <, (Q_a)_{a \in \Sigma} \right)$$

where, for all $a \in \Sigma$ the set Q_a are the positions containing the character a which is written $Q_a = \{i \in \omega \mid \alpha_i = a\}$.

Linear time is distinguished in that it possesses one sort of modal truth. The sequence of integers α there is only one possible successor: in a ω -word, for any given i , the only possible successor $i + 1$. This corresponds to an interpretation of computations in M_{S1S} in terms of sequences of states $\sigma \in Q^\omega$ which are called *runs*. Any statement about runs σ is implicitly quantified with the sole modal operator \forall , meaning “for all paths in M_{S1S} .”

The interpreted second order theory of one successor (S1S) defines properties in terms of formulas φ .¹ The theory is second order because the quantification is over sets of elements rather than individual elements as is the case in Floyd-Hoare logic. The “one successor” comes from the interpretation of computations as runs σ . The central decision

1. The exact constitution of formulae in S1S is not of direct interest to the development here, the reader is referred to one of the tutorial presentations such as Thomas [684] or Emerson [248]. The major thrust of Büchi’s development was that an automaton $B[\varphi]$ can be used in decision procedures. Thomas [684], page 145, outlines an explicit construction relating $\varphi \in S1S$ to $B[\varphi]$, as does Büchi’s original presentation [131].

problem of the theory is whether α is a model for φ ; this is written as $\alpha \models \varphi$. Büchi [131] showed that any statement in S1S can be rewritten in terms of sequences accepted by a class of ω -automaton. This class has since been dubbed the *Büchi automata*. This result means that formula in S1S can be dispensed with for practical applications: for any $\varphi \in S1S$ one can work exclusively with the implied automata $B[\varphi]$.

A Kripke structure M_K can be used as a generator for elements of M_{S1S} by interpreting M_K as an automaton (say a Büchi automaton or an L-automaton). Then each run σ in the automaton, implies a corresponding sequence of labelings Γ , where $\Gamma = \Phi(q_0) \Phi(q_1) \Phi(q_2) \dots \in (2^{AP})^\omega$. The sequence Γ carries with it the information about the run σ for the atomic properties in AP . The set of all sequences Γ defines the language of the model M_{S1S} and thus the language of the Kripke structure M_K .

A decision procedure for checking that $M_{S1S}, \sigma \models \varphi$ was first suggested by Vardi and Wolper [706]. The idea is to treat the Kripke structure M_K as a generator for elements of the linear time model M_{S1S} and understand that as a Büchi automaton with the language $L(M_{S1S})$. Further, the formula φ , when expressed in Linear Time Temporal Logic, can be transformed into a Büchi automata as well. The formula φ can be said to denote the language of its corresponding Büchi automaton $L(B[\varphi])$. The decision procedure answering whether $M_{S1S}, \sigma \models \varphi$ is then redefined to the problem of answering whether $L(M_K) \subseteq L(B[\varphi])$. This is ω -regular language containment problem. The ω -regular languages are closed under complementation this question in turn can be transformed to the question of whether $L(M_K) \cap \overline{L(B[\varphi])} = \emptyset$. This is directly computable as $L(M_K \times \overline{B[\varphi]}) = \emptyset$ given the “complement automaton” $\overline{B[\varphi]}$. This scheme is called *language containment* and the decision procedure is called the *language emptiness problem*.

Subsequent research has investigated how to approach the automata complementation

problem. This has been shown to be extremely difficult for Büchi automata requiring time in $O(2^{n \log n})$ [633] [252]. Kurshan's contribution [449] [454] was the definition of a class of automata, the L-automata, in which the complementation step is trivial, being a syntactic operation on the acceptance criterion for the automaton. The problem remains PSPACE-hard in the general case as an FSA intersection problem must be solved.

2.2.4.2 Branching time

A branching time model for a Kripke structure $M_K = (Q, T, \Phi)$ is given by:

$$M_{S2S} = (\hat{Q}, \hat{T}, \hat{\Phi})$$

where:

$\hat{Q} = Q \times \aleph$ is an infinite set of states,

$\hat{T} \subseteq \hat{Q} \times \hat{Q}$ is an infinite transition relation,

$\hat{\Phi}: \hat{Q} \rightarrow 2^{AP}$ is the labeling function.

The infinite model M_{S2S} is created from the finite structure M_K by unwinding the structure according to the following rules:

- for the start state $q_0 \in Q$ then $(q_0, 0) \in \hat{Q}$,
- for $(q, n) \in \hat{Q}$ then $\{(q', n+1) \mid \forall q'. (q, q') \in T\} \subseteq \hat{Q}$,
- for $(q, q') \in T$ then $\exists n \in \aleph. ((q, n), (q', n+1)) \in \hat{T}$,
- $\hat{\Phi}((q, n)) = \Phi(q)$.

The unwinding of M_K into an infinite tree M_{S2S} is as shown in Figure 2-6.¹ In the branching time framework, time is discrete as before. The distinction is that time has a branching tree-like structure governed by the relation \hat{T} . Each path in the tree has a one-to-one correspondence with the natural numbers, exactly as is the case with the linear time case. Succinctly, the difference between linear time and branching time is that in the branching case, each point in time has multiple possible futures but only one past.

1. From Emerson [248], page 1013.

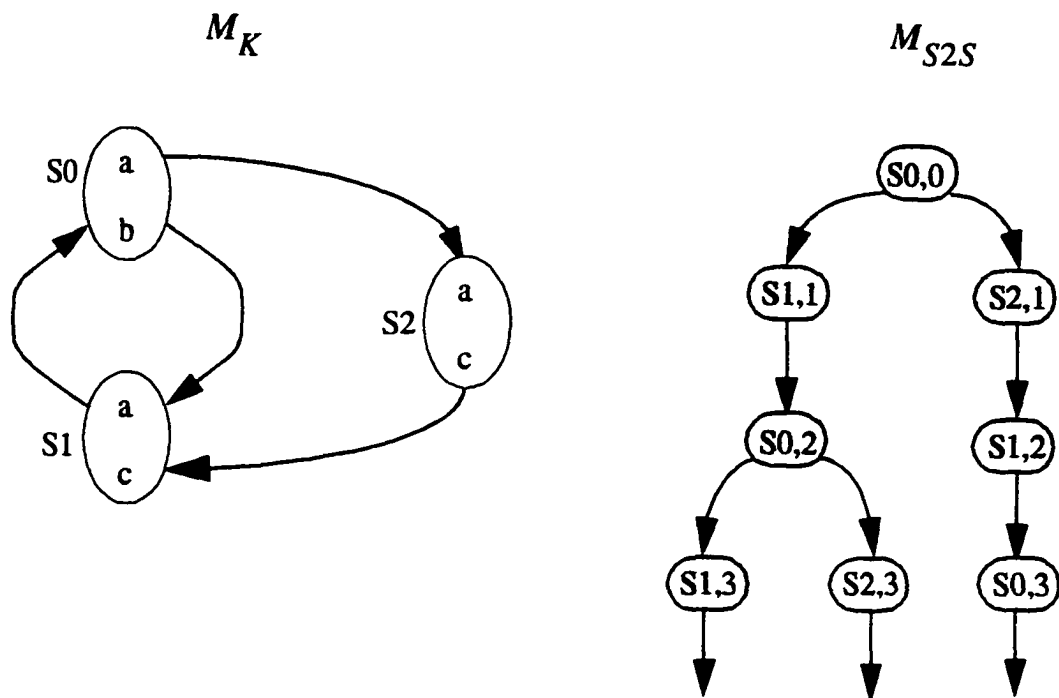


Figure 2-6. The Unraveling of M_K into the Infinite Tree M_{S2S}

At an even more primitive level, a branching time Kripke model is given a model-theoretic interpretation in terms of the infinite binary tree. The tree is understood to be over the alphabet $\Sigma = \{0, 1\}^n$ with each node in the tree being marked with a character $a \in \Sigma$. The bits of the character a are understood to denote whether $p_i \in AP$ holds at that node. The character of a tree node is written $a = t(w)$ for a word $w \in \{0, 1\}^*$ and a bit within the character is written $a_i = t(w)_i$.

Formally the tree structure is given as:

$$t_2 = \left(\{0, 1\}^*, \varepsilon, succ_0, succ_1, <, P_1 \dots P_n \right)$$

where:

$$succ_0(w) = w \cdot 0 \text{ and } succ_1(w) = w \cdot 1,$$

$<$ is the proper-prefix relation over finite words $w \in \{0, 1\}^*$,

the properties $P_1 \dots P_n$ are subsets of $\{0, 1\}^*$ and $w \in P_i$ when $t(w)_i = 1$.

The second-order theory of two successors (S2S) defines properties over such tree structures as t_2 . The primitive binary structure t_2 can be generalized to bounded fanout trees t_n where there are n successor functions $succ_0 \dots succ_{n-1}$. The theory even extends to trees with uncountable fanout t_ω . These extensions give rise to theories S_nS and $S_\omega S$ respectively.

The decidability of S2S is given by the Rabin Tree Theorem [610]. That theorem states that $M_{S2S}, q_0 \models \varphi$ is decidable. The theorem is proven by converting φ to a special kind of tree automaton¹ called a Rabin tree automaton $R_{t_2}[\varphi]$. Thus M_{S2S}, q_0 is a model for φ if and only if that $R_{t_2}[\varphi]$ has an accepting run.² The class of logics where this property holds are said to have the “tree model property.”

The most practical application of branching time has come with the Computation Tree Logic (CTL) of Clarke and Emerson [249] and its many extensions such as CTL* and Fair CTL. The syntactic composition of CTL and CTL* are given in Figure 2-7. The semantics, as defined by the “is-a-model-of” relation $M, x \models \varphi$, is shown in Figure 2-8. The important aspect of CTL is that the path operators E and A are defined in terms of a fixed point computation [184]. As a result, model checking for CTL can be done in time linear

1. A tree automaton has a transition structure $T \subseteq Q \times \Sigma \times (Q \times Q)$.

2. As with S1S, the exact constitution of S2S is not of direct interest to the development here. The tutorials by Emerson [248] and Thomas [684] review the deeper aspects of the theory and the various classes of tree automata.

in the size of the model and of the formula, $O(|M||\varphi|)$. Later, Clarke, Emerson and Sistla [185] showed how model checking for CTL can be performed using breadth-first search and the identification of strongly-connected components. The significance of the fixed point computation and the use of strongly-connected components is that they are straightforward in a second-order μ -calculus framework.

Later additions to the logic added fairness constraints [254] [255] by defining the unique operators A_{Φ_i} and E_{Φ_i} for each fairness constraint Φ_i . The operators only hold on paths where the condition Φ_i holds. Most recently, Burch, Clarke, McMillan, and Dill [136] applied symbolic methods using OBDDs to the model checking problem.

Class	Rule	Inputs	Syntax
<i>SF</i>	S1	$p \in AP$	Each atomic proposition $p \in AP$
<i>SF</i>	S2	$p_1, p_2 \in SF$	$p_1 \wedge p_2, p_1 \vee p_2, \neg p$
<i>SF</i>	S3	$p \in PF$	Ep, Ap
<i>PF</i>	P0	$p_1, p_2 \in SF$	Xp, p_1Up_2
<i>PF</i>	P1	$p \in SF$	All state p formulae are path formulae
<i>PF</i>	P2	$p_1, p_2 \in PF$	$p_1 \wedge p_2, p_1 \vee p_2, \neg p$
<i>PF</i>	P3	$p_1, p_2 \in PF$	Xp, p_1Up_2

AP - atomic propositions
SF - state formulae
PF - path formulae

Figure 2-7. Syntax of CTL and CTL*

The issue of branching time and temporal logic is presented here for completeness.¹ McMillan [518] describes symbolic methods for model checking in branching time CTL in detail.² Of interest here is the definition of branching time semantics and its relationship to the model checking decision procedure $M, q \models \varphi$. McMillan's SMV system is

1. The idea of temporal logic originated with Pnueli [599] [61].
 2. Further related work which applies OBDD methods to verification problems is reviewed in Chapter 7.

Rule	Inputs	Syntax
S1	$p \in AP$	$S \llbracket M, q_0 \models p \rrbracket = p \in \Phi(q_0)$
S2	$p \in PF$	$S \llbracket M, q_0 \models p_1 \wedge p_2 \rrbracket = S \llbracket M, q_0 \models p_1 \rrbracket \wedge S \llbracket M, q_0 \models p_2 \rrbracket$
		$S \llbracket M, q_0 \models p_1 \vee p_2 \rrbracket = S \llbracket M, q_0 \models p_1 \rrbracket \vee S \llbracket M, q_0 \models p_2 \rrbracket$
		$S \llbracket M, q_0 \models \neg p \rrbracket = \neg S \llbracket M, q_0 \models p \rrbracket$
S3	$p \in PF$	$S \llbracket M, q_0 \models Ep \rrbracket = \exists \hat{q} \in Q^{\omega}. P \llbracket M, \hat{q} \models p \rrbracket$
		$S \llbracket M, q_0 \models Ap \rrbracket = \forall \hat{q} \in Q^{\omega}. P \llbracket M, \hat{q} \models p \rrbracket$
P0	$p_1, p_2 \in SF$	$P \llbracket M, \hat{q} \models p_1 \cup p_2 \rrbracket = \exists i \in \omega. (S \llbracket M, q_i \models p_2 \rrbracket \wedge \forall j \in \omega. (i < j \wedge S \llbracket M, q_j \models p_2 \rrbracket))$
		$P \llbracket M, \hat{q} \models Xp \rrbracket = P \llbracket M, \overline{q[1\dots]} \models p \rrbracket$
P1	$p \in SF$	$P \llbracket M, \hat{q} \models p \rrbracket = S \llbracket M, q_0 \models p \rrbracket$
P2	$p_1, p_2 \in PF$	$P \llbracket M, \hat{q} \models p_1 \wedge p_2 \rrbracket = P \llbracket M, \hat{q} \models p_1 \rrbracket \wedge P \llbracket M, \hat{q} \models p_2 \rrbracket$
		$P \llbracket M, \hat{q} \models p_1 \vee p_2 \rrbracket = P \llbracket M, \hat{q} \models p_1 \rrbracket \vee P \llbracket M, \hat{q} \models p_2 \rrbracket$
		$P \llbracket M, \hat{q} \models \neg p \rrbracket = \neg P \llbracket M, \hat{q} \models p \rrbracket$
P3	$p_1, p_2 \in PF$	Same rules as for P0 except that $p_1, p_2 \in PF$ instead of $p_1, p_2 \in SF$

CTL is generated by rules S1, S2, S3, P0, P1, P2
 CTL* is generated by rules S1, S2, S3, P1, P2, P3

Figure 2-8. The Semantics of CTL and CTL*

most illustrative in this regard. In that system, the SMV language is used to describe the system and its fairness constraints while CTL is used to describe the property to be checked. The semantic transformations are illustrated in Figure 2-9. The key point of that figure is that the semantics of SMV programs is defined directly in terms of a Kripke structure. The model checking procedure processes this structure on the formula ϕ in a syntax-directed fashion.¹

2.2.5. The ω -Automata

The ω -automata model directly implies the use of language containment as the property-checking procedure. The language containment proceeds from the premise that any system which accepts inputs or produces outputs has an automata-theoretic interpretation

1. In fact the denotational semantics of SMV constructs a Kripke structure directly. The semantics concentrates directly on constructing the transition relation T from syntactic terms in the language.

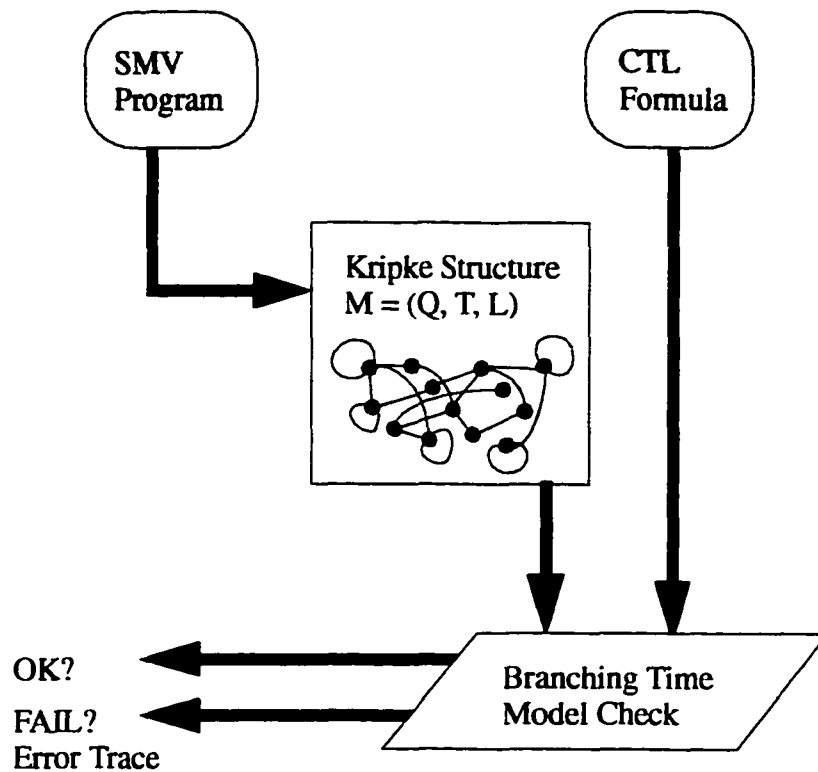


Figure 2-9. The Branching-Time Semantics in SMV

as a generator or recognizer of a language [368]. In the case of a Finite State Machine (FSM), the input/output patterns, under an ordering, can be thought of as the bit-level encoding of a symbol in an alphabet. The alphabet Σ is then the set of bit vectors of length $m + n$ from the set B^{m+n} for an m input n output FSM [439]. An FSM S can be spoken of as having a language as denoted by $L(S)$ which is the set of all possible input/output sequences allowed by the machine. The ω -automata are recognizers for languages consisting not of finite strings, but rather of infinite sequences.

Languages are set-theoretic objects, being sets of strings, or sets of sequences in the infinite case. The concern with ω -automata is exclusively with the infinite case, as reactive systems are considered to run forever, and clearly the finite case is subsumed by the infinite.¹ The ω -regular languages [171] [687], the languages accepted by the ω -autom-

ata, offer the most natural model for the behavior of reactive systems which never halt. Finite or infinite, a language is a set, and as such it is possible to speak of one language being larger, smaller, equal, or being unrelated to another, just as one can for any other kind of set. Such sets are finite so it is possible to speak of the complement of language:

$$\bar{L} = \Sigma^\omega - L$$

The idea of language containment [706] [449] comes from the formalization of a verification problem as the query of whether the language of the system $L(S)$ is contained in the language of some property $L(M)$:

$$L(S) \subseteq L(M)$$

The query asks whether the (generated) behaviors of the system S are smaller than the set of behaviors allowed by the monitor M . It is not possible to decide this directly as both $L(S)$ and $L(M)$ are rather large sets of infinitary objects. A related expression provides the answer via the complement language. The decidable query is:

$$L(S) \cap \overline{L(M)} = \emptyset$$

This quantity can be computed directly so long as the product-machine operator \times is modular. That is, it must preserve the property that the language of the product is the intersection of the languages:

$$L(M_1 \times M_2) = L(M_1) \cap L(M_2)$$

All that is needed is a representation for an automata \bar{M} , that accepts the complement language $\overline{L(M)}$ defined as follows:

$$L(\bar{M}) = \overline{L(M)} = \Sigma^\omega - L(M)$$

The original containment query is then decidable according to the emptiness of the language of the product with the complement automaton:

1. In fact, though this is somewhat obvious, Kurshan [454] spends some effort explaining this.

$$L(S \times \overline{M}) = \emptyset$$

This problem has been traditionally called the *emptiness of complement* problem. The difficulty of the complementation step varies according to the sort of automata that is used. The emptiness of the product is however known to be PSPACE-complete.¹ For language containment to be a useful decision procedure in design verification it remains to identify a scheme under which the (structural) product-machine operator guarantees the language intersection property and where the complementation of the ω -automaton M is in some sense easy.

The L-Automata

Finite automata recognizing finite strings (the **-regular* languages) [368] have but one acceptance structure that defines the language accepted by an automaton. That structure is the familiar set of final states F where a Nondeterministic Finite Automaton (NFA) M is given by:

$$M = (Q, \Sigma, T, I, F)$$

where:

Q is a finite set of states.

Σ is a finite alphabet of input symbols.

$T \subseteq Q \times \Sigma \times Q$ is a transition relation.

$I \subseteq Q$ is a set of initial states.

$F \subseteq Q$ is a set of final states.

A finite string $s \in \Sigma^*$ is said to have an accepting run $\mu(\vec{q})$ in M in the case that the string finishes with the automaton in one of the final states. This run of states is denoted:

$$\mu_M(\vec{q}) = \{q_i \in Q \mid q_0 \in I \wedge (q_i, s_i, q_{i+1}) \in T \wedge q_n \in F\}$$

1. FSA Intersection for the finite case has been shown to be PSPACE-complete: [278] problem AL6. The same complexity is attributed to the infinite case as well: [454] section 8.2.

For the finite case, \bar{M} , the automaton accepting the complement language is derived from M in a straightforward fashion. $\overline{L(M)}$ is simply the set of strings with runs in M that do not finish at a state $q_n \in F$. Thus \bar{M} is defined in terms of M by complementing the set F with respect to the set of possible states:

$$\bar{M} = (Q, \Sigma, T, I, Q - F)$$

The infinite case is not nearly so straightforward. The notion of a run $\mu(\vec{q})$ must be generalized to an infinite vector \vec{q} of states where (clearly) some states must appear infinitely often. The various classes of ω -automata are distinguished by how the accepting states are specified.

The obvious generalization of the finite NFA to the infinite case are called the Büchi automata [131] where the set of runs intersects the set F . Unfortunately such automata are not easily complemented. Kurshan's approach was to identify a class of automata that is easily complemented. The accepting runs of an L-automaton is defined in terms of the disjunction of two items, a set of *cycle sets* and a set of *recur edges*. An L-automaton is defined as:

$$M = (Q, \Sigma, T, I, Z, C)$$

where:

Q is a set of states,

Σ is the input alphabet,

$T \subseteq Q \times \Sigma \times Q$ is a transition relation,

$I \subseteq Q$ is a set of initial states,

$Z \subseteq Q \times Q$ is a set of recur edges,

$C \subseteq 2^Q$ is a set of cycle sets.

An accepting run in M is defined the set of runs that either pass through a recur edge infinitely many times or else eventually cycles wholly within a cycle set C_i forever:

$$\mu_M(\vec{q}) = \left\{ q_i \in Q \mid \left(\begin{array}{l} q_0 \in I \wedge (q_i, s_i, q_{i+1}) \in T \wedge \\ \text{inf}_e(\vec{q}) \cap Z \neq \emptyset \vee \exists C_i \in C. \text{inf}_v(\vec{q}) \subseteq C_i \end{array} \right) \right\}$$

The language of M is defined as the set of runs that either traverse a recur edge infinitely often or which eventually settle within a cycle set:

$$L(M) = \{x \in \Sigma^\omega \mid x \text{ has a run } \mu_M(\vec{q})\}$$

By the Theorem of Kurshan,¹ a deterministic L-automaton has a unique accepting run. Thus its complement language $\overline{L(M)}$ is defined by runs where recur edges are traversed *finitely* often and any infinite behavior is *not* contained in a cycle set:

$$\overline{\mu_M}(\vec{q}) = \left\{ q_i \in Q \mid \left(\begin{array}{l} q_0 \in I \wedge (q_i, s_i, q_{i+1}) \in T \wedge \\ \text{inf}_e(\vec{q}) \cap Z = \emptyset \wedge \neg(\exists C_i \in C. \text{inf}_v(\vec{q}) \subseteq C_i) \end{array} \right) \right\}$$

$$\overline{L(M)} = \{x \in \Sigma^\omega \mid x \text{ has a run } \overline{\mu_M}(\vec{q})\}$$

The elegance here is that the complementation of L-automata is entirely syntactic by virtue of treating the acceptance conditions Z and C in the complementary fashion. A deterministic L-automaton can be complemented in unit time simply by considering its acceptance conditions in a different light.

Practical implementations of formal verification based on language containment using various classes of ω -automata, including L-automata and even more expressive varieties, have been shown by Kurshan [450] [454], and Hojati *et al.* [364] [359] [362] [361].

2.2.6 Denotational Models

The denotational theory of computing starts from the premise that given a structure such as a network or a state transition system, there is an natural operational definition which

1. Variously presented *c.f.* [454], Lemma 6.2.30, page 95.

defines an execution of that structure. This definition is said to be the *operational semantics* of the structure. Such a system exists because it can be laid out as a (finite) set of rules. It is far less obvious however that a given operational semantics is well defined in the sense that its operations preserve any deeper mathematical relationships. The *denotational semantics* of the system establishes just such a connection by relating the syntactic objects of the structure to elements of carefully constructed (possibly infinite) function spaces where such relationships are already known to exist. These function spaces are called *domains* and are constructed to preserve the topological notions of composition, partial ordering, approximation, limits, least upper bounds and fixed points. These abstract topological concepts are in turn used to model the operational concepts of execution, termination and equivalence. Further, once the operational and denotational semantics are proved equivalent then propositions about the execution and manipulation on the syntactic structure can be said to have a meaning in the model.

A denotational semantics is based on the premise that composition at the syntactic level (juxtaposition) corresponds directly to functional composition on a suitably defined domain. So a semantics is a map $S:L \rightarrow M$ from programs in a language L to instances of a model M which preserves composition:

$$S[\![\text{statement}]\!] = \text{model}$$

$$S[\![\text{statement}_1 ; \text{statement}_2]\!] = S[\![\text{statement}_1]\!] \circ S[\![\text{statement}_2]\!]$$

$$S[\![\text{statement}_1 ; \text{statement}_2]\!] = \text{model}_1 \circ \text{model}_2$$

The semantics defines a set of domain equations associating syntactic structures with elements of the underlying models. The semantics is said to be *fully abstract* when two statements which behave the same (relative to some model-theoretic definition of behavior) denote the same model element. With either a fully abstract or a non-abstract denotational semantics in hand, the execution of the program under the operational rules can truly be said to “solve” the domain equations for the final model element. As such, the central con-

struction in a denotational semantics is a domain which is suitably rich so that it is certain to contain all of the required initial, intermediate and final functions. In particular the model composition function $\circ : M \rightarrow M$ must always exist and be well-defined. Failing its existence, there might exist programs which, through composition, attempt to denote non-existent elements in the underlying model. Such programs, if they existed, would necessarily be nonsensical and by implication their operational result would necessarily be undefined.

Fortunately, Scott showed that there exists a universal domain which is rich enough to contain any computable function and into which any computably-based domain can be embedded (conversely, out of which any such domain can be retracted). That universal domain is $\wp \omega$, the powerset of the natural numbers. This is a surprisingly general result because it applies both in case of the finite computations as well as infinite ones. Its significance is that *any behavior* which can be described in terms of a computable function can, in principle, be described through a denotational semantics that constructs the appropriate domains. The generality of the theory allows for its application in a wide variety of situations pertaining to both finite (terminating) and infinite (non-terminating) computations.

Thus in this small survey, denotational semantics offers essentially a recipe for constructing a representation of the model in terms of ideas from algebraic topology. As such, a denotational model of a system with state and state transitions, no matter how these elements are represented in model, will necessarily take advantage of topological concepts such as composition, partial ordering, approximation, limits, least upper bounds and fixed points. For the purposes here it is sufficient to observe that a singleton state, a set of states and even the transition relation itself are isomorphic to their respective characteristic function which is necessarily a computable function [324]. As such, the basic operations of formal verification, symbolic execution, can be phrased in terms of such topological concepts and in fact can take advantage of them in the design of the language semantics itself.

In fact, the exposition of these principles and their effect on the model of time within a step, on the existence and utility of a well-defined notion of δ -time, is the central focus of the computational semantics proposed in Chapter 3.

2.2.7 Focus

The previous sections highlighted five of the standard semantic models. In each case the formalism consisted of some explicit or latent definition of state coupled with a monolithic or disaggregated definition of a transition relation. The exact form of the state and transition relation representation varied and in that variance enabled the definition of various notions of behavior. The common thread throughout was the notion of state, latent or explicit, monolithic or disaggregated, and the concomitant transition relation which defined the (macro) step of the formalism. Within each formalism distinctions could be observed in the fine structures in the representation of state or the transition relation itself. Of course, among all five of the standard models the definition of behavior was subtly different though relationships have been established among the notions of trace conformance, language containment, model checking and bisimilarity.¹ Finally the domain theory of Scott and the universal domain $\wp\omega$ were introduced, not as a further semantic model *per se*, but rather as the theoretical basis for a semantics based on approximations and fixed points in a non-abstract δ -time to be presented Chapter 3.

2.3 Some Non-Standard Models

Having reviewed the standard models, their operational properties and behavioral specifications, it is useful to contrast them with some non-standard models which were designed with a more specific application language in mind. There are two non-standard models which can be highlighted for their elegant association to their respective language.

1. Gupta's overview of formal verification methods [310] surveys such results where they are known.

They are distinguished in that they each form the semantic basis for a particular language which is then able to exploit some particularly interesting property of the model. The two models are the Non-Deterministic Event Sequence model which is the semantic model for the UDL/I language and the domain ${}_2Z$, the 2-adic integers, which is the semantic model for the 2Z language. These models and their languages are summarized in this section. The major focus is the semantic model, as opposed to the language *per se*, and especially focusing on the simulation, synthesis or verification analysis that is enabled by the semantic model. The treatment here is again that of a survey, presenting the major features and highlightings to these languages. A simple program fragment is provided for each language to give some sense of its flavor.

2.3.1 Non-Deterministic Event Sequences (NES)

The Nondeterministic Event Sequence (NES) model [731] [395] forms the semantic basis for the UDL/I language [732] [404]. The high-level language is UDL/I which is oriented at gate-level and register-transfer-level descriptions of hardware. The language was originally designed for standardization on a par with VHDL but with an eye towards being useful as a specification vehicle for automated synthesis procedures [423]. The syntax of the language is based on an earlier language HSL-FX [561] which was developed at Nippon Telephone & Telegraph. An example of a program fragment from the literature is shown in Figure 2-10.¹

The Nondeterministic Event Sequence is general enough to support the notions of time used in both asynchronous and synchronous circuit design methodologies. There is even a simple intermediate “core language” that provides a sort of half-way point between the HDL at the high level and the operationally-defined event sequence machine at the low level. The language-to-semantics linkage is shown in Figure 2-11. The only aspect miss-

1. This is a partial presentation of the example in Karatsu [418], page 55.

```

automaton: ifet(ift): .rset: .clk;
  fetch: begin
    memout := mem<pc>;
    pc := inc(pc);
    op1 := memout;
    if memout<0> then
      "test MSB of OP1"
      "1 = 2-word instruction"
      "0 = 1-word instruction"
      -> op2fet;
    else
      "task transfer"
      --> exec**ext;
      -> fetch;
    end_if;
  end;
  op2fet: begin
    memout := mem<pc>;
    op2 := memout;
    pc := inc(pc);
    "task transfer"
    -->exec**ext;
  end;
end_auto;

```

Figure 2-10. A Finite State Machine in the UDL/I Language

ing are effective algorithmic procedures for synthesis, simulation and verification that exploit the properties of the model.¹

An understanding of UDL/I and the purpose of its NES model is much more obvious if one understands that the language was designed for the description of mostly gate-level descriptions with a second level of synchrony defined by clock-activated latches. As such, the “design center” of the model is asynchronous logic where delays are measured in unit-based time coupled with a secondary level of synchronous time defined by an externally-supplied clock. The major questions to be asked of descriptions in the language are those

1. This is not to say that commercial UDL/I simulators do not exist. The point here is that these simulators do not exploit the all-possible-scenarios semantics that is intrinsic to the NES model. Rather they support only a deterministic subset using the accepted event-driven or compiled-code schemes. As such they merely provide an alternative to simulators accepting VHDL or Verilog. Some attempts have been made to apply symbolic methods to the full NES model [155] [393].

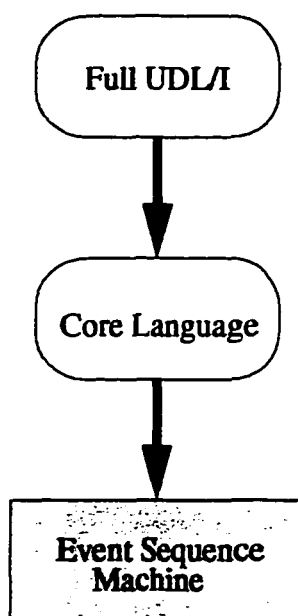


Figure 2-11. The Structure of the Semantics of UDL/I

concerning the presence or absence of glitches or indeterminacies at the asynchronous level.¹ The ability to pose questions about the behaviors at the synchronous level are at best secondary.

Uncertainty is fundamental to the gate-level problem space. Uncertainties consist of unknown values such as the unknown delay of a gate or so-called “don’t care” values as found in incompletely specified boolean functions. Both of these types of uncertainty are addressed in the NES model through the use of nondeterminism. A system is modeled as a sequence of events $E = (p, v)$ which are understood as a value v occurring at a place p . Places are wires or latches and it is said that with the occurrence of E , the place p becomes v . Events have zero duration and events are totally ordered. As such there are no simultaneous events. Causal relations between any two events E_1 and E_2 is given by a

1. There are some interesting and nonobvious results in this area such as *Monotone Speedup Failure* where a combinational circuit may take longer to stabilize as the gates on the critical path are sped up [516]; the circuit as a whole may slow down when individual gates are sped up.

partial order between the two events $E_1 < E_2$. This partial order is said to define a set of all event traces that satisfy the partial order. An example of a partial order defining a set is shown in Figure 2-12. The model-theoretic basis of the NES model is therefore that of trace sets.

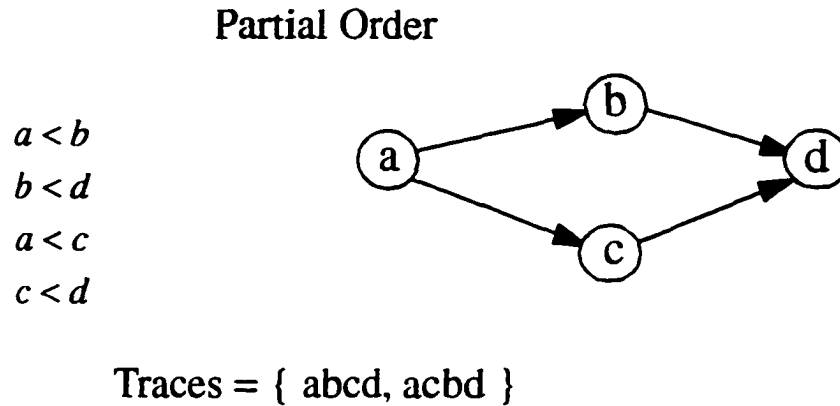


Figure 2-12. A Partial Order on NES Events Defines a Trace Set

The NES model has a multiform notion of time. That is, time is treated as a special sort of event $E = (@T, unit)$ and there can be many kinds of time. Examples of units might be nano-seconds or clock ticks.

Operationally the NES model is defined over an abstract machine which is somewhat like a Turing machine. The NES machine reads a single tape from left to right and computes based on the events that it reads on the tape. Processing an event consists of reading the event from the tape and producing a new output sequence of events which are inserted back onto the tape just before the read pointer. This is shown in Figure 2-13.

Formally, a NES machine is defined as:

$$M = (Q, E, I, \delta)$$

where:

Q is a set of states,

E is the alphabet of events that can be seen on the tape,

$I \subseteq Q$ is a set of initial states,

$\delta \subseteq (Q \times E) \times (Q \times E^*)$ is a transition relation describing how a state-event pair moves to a new state, writing a sequence (not just a single event) onto the tape.

A run in M is given by:

$$\xi: Q \times E \rightarrow 2^{E^*}$$

where:

$$\xi(q, \varepsilon) = \{\varepsilon\}$$

$$\xi(q, e \cdot x) = \{e \cdot y \mid ((q, e), (q', \sigma)) \in \delta, y \in \xi(q', \sigma \cdot x)\}$$

The NES machine specifies behavior as the set of runs through the machine. This makes the machine a transformer of traces to traces defined formally as:

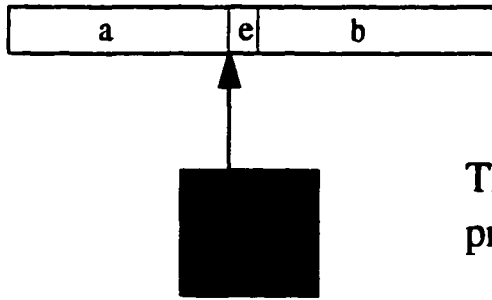
$$\Xi: 2^{E^*} \rightarrow 2^{E^*}$$

where:

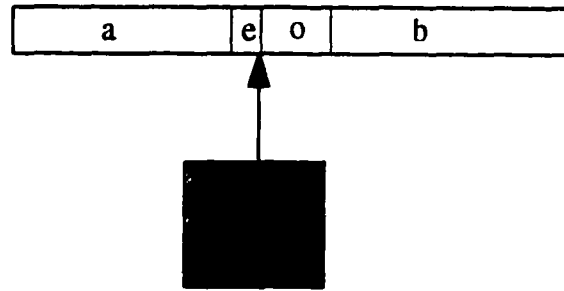
$$\Xi = \{ (x, y) \mid \forall x \in E^*, s \in I, y \in E^*. y \in \xi(s, x) \}$$

There is a parallel composition operator that allows for two machines to be structurally composed together to form one larger machine. The operator $M_1 \parallel M_2$ is defined using the “shuffle set” of two strings defined as

$$shuffle(\sigma_1, \sigma_2) = \{ \sigma \mid proj_1(\sigma) = \sigma_1, proj_2(\sigma) = \sigma_2 \}$$



The machine before processing the event e



The machine after reading the event e having written output o

Figure 2-13. The Execution of an NES Machine

The parallel composition of two NES machines is then given by:

$$M_1 \parallel M_2 = (Q_1 \times Q_2, E_1 \cup E_2, I_1 \times I_2, \delta_{M_1 \parallel M_2})$$

where:

$$\delta_{M_1 \parallel M_2}((q_1, q_2), e) = \left\{ \begin{array}{l} ((q_1, e), (q'_1, \sigma_1)) \in \delta_1 \\ ((q'_1, q'_2), \sigma) \mid ((q_2, e), (q'_2, \sigma_2)) \in \delta_2 \\ \sigma \in \text{shuffle}(\sigma_1, \sigma_2) \end{array} \right\}$$

A system is modeled at the NES machine level by treating all gates and latches in the system description as individual machines. The parallel composition operator is used to express the execution of all of these machines in unison. Conceptually, a simulator for the NES machine could then compute the set of possible traces allowed by the design.

A UDL/I description is converted to a NES machine description by first compiling it

into the core language. The definition of constructs in the core language is then defined in terms of individual abstract machines and the parallel composition operator. The correctness of an implementation with traces Ξ_I relative to a specification with traces Ξ_S is defined relative to a trace conformance condition:

$$\forall x \in E^*. \Xi_I(x) \subseteq \Xi_S(x)$$

Any simulation procedure that “executes” an NES model must intrinsically be complex as the semantics denotes all possible behaviors be computed and represented in a data structure. Ishiura *et al.* [393] admit that the semantic model is horrendously detailed and probably too complex for use on a design-wide scale. In response they posit the existence of a range of accuracies under the NES model such that different simulation algorithms could be used for different accuracy levels. Such an accuracy framework however has not materialized so the NES model has not seen use in practical applications.

2.3.2 The 2-adic Integers (${}_2Z$)

A second but more practical non-standard model is the so-called 2-adic integers (${}_2Z$) which provides a semantic model for the 2Z language [100]. 2Z is used to describe computations on systolic array processors and so-called Programmable Active Memories [86] in particular. An example of a program fragment from the literature is shown in Figure 2-14.¹

Vuillemin [710] established the connection between Hensel’s p -adic numbers² for $p = 2$, several classes of computable functions and synchronous circuits. The 2Z language semantics is an interpretation of expressions in ${}_2Z$ as synchronous circuits defined from the composition of four basis elements: the constants $0 = {}_20\dots$ and $-1 = {}_21\dots$,

1. Vuillemin [710], page 877.

2. Hensel’s work is attributed variously as being “around 1900” or 1913 [346]. A more modern presentation can be found in Knuth [438].


```

device Mulmat
  n = 4;           // Vector size
  b = 24;         // Serial bits per sample
  input   x:[n];
  output  y:[n];
  var     C:[n,n];

  // Read the coefficients
  C = Read("FileOfCoeffs");
  // Synthesis for the reset signal
  r = 2**(b-1)/(1-2**b);

  reset r do
    for i < n do
      Y[0,i] = 0;
      for j < n do
        Y[j+1,i] = Y[j,i] + C[i,j] * x[j];
      end for;
      y[i] = Y[n,i];
    end for;
  end reset
end device;

```

Figure 2-14. A Matrix Multiplier in the 2Z Language

the register and the multiplexor (mux). The structure of the language and its semantics is shown in Figure 2-11.

The p -adic numbers are the integers $Z = \{-\omega, \dots, -2, -1, 0, 1, 2, \dots, \omega\}$ modulo a prime p which are presented least significant digit first. A p -adic number is written according to the rule $B_p(n) = (n \bmod p) B_p(n \div p)$. The representation of a p -adic number is by definition infinite and as such the terms generated by $B_p(n)$ must eventually cycle. For example, for $p = 2$ and using parenthesis to denote repetition, zero is represented by ${}_2(0)$, one by ${}_21(0)$, two by ${}_201(0)$ and so on. Negative numbers are represented in their infinite 2s-complement form: minus one is ${}_2(1)$, minus two is ${}_20(1)$, minus three is ${}_210(1)$ and so on. The rationals with an odd denominator tend to have non-trivial repetitions; such as $-22/7$ which is represented as ${}_20101(110)$. For $p = 2$ there is the obvious association of $n \bmod p$ with bits and of $B_2(n)$ with serial computation.

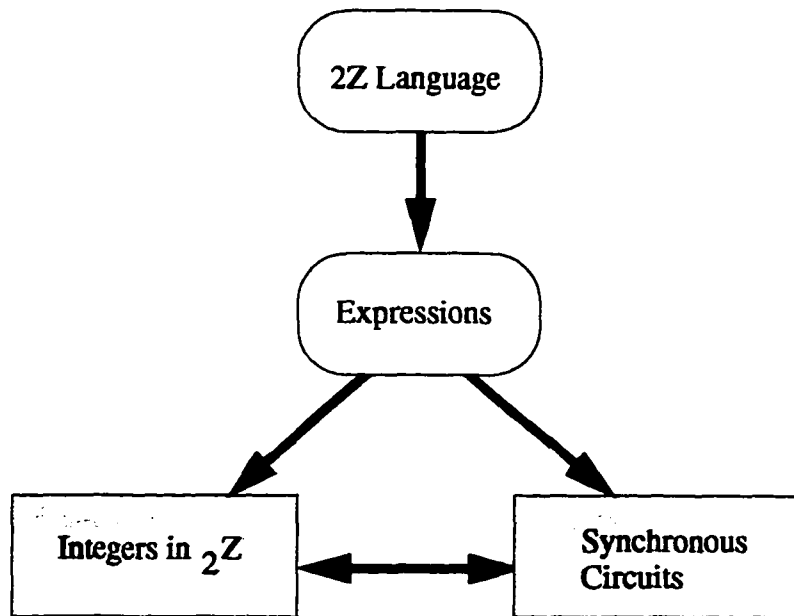


Figure 2-15. The Structure of the Semantics of $2Z$

What is interesting about ${}_2Z$ is that it has the properties of *both* the ring over the integers,¹ $(Z, \circ + \circ, \circ - \circ, 0, \circ \times \circ, 1)$, and of the boolean algebra over the set of natural numbers² $(N, \circ \cap \circ, \circ \cup \circ, -\circ, \emptyset, 2^N)$. What is missing is (truncative) integer division, and exact division by an even number. This latter loss means that ${}_2Z$ is not a field as there is no multiplicative inverse for every element of the structure: $\forall b \in {}_2Z. \exists i. i \times b = 1$ fails to hold. The structure ${}_2Z$ is *almost* a field in the sense that the so-called odd inverse is

1. From topology [427]:

A *monad* is a structure $(M, \circ \cdot \circ, \varepsilon)$ over a set M where $\circ \cdot \circ$ is associative and ε is the unique null element.

A *group* is a structure $(G, \circ + \circ, \circ^{-1}, e)$ over a set G where $|G| \geq 1$, $\circ + \circ$ is associative, e is the unique neutral element and $\forall x \in G. x + x^{-1} = e$; a group is *commutative* or *abelian* when $\circ + \circ$ is commutative.

A *ring* is a structure $(R, \circ + \circ, \circ - \circ, 0, \circ \times \circ, 1)$ where $(R, \circ + \circ, \circ - \circ, 0)$ is an abelian group and $\circ \times \circ$ is associative and distributes over $\circ + \circ$.

A *field* is a structure $(F, \circ + \circ, \circ - \circ, 0, \circ \times \circ, 1)$ which is a ring and $|F| \geq 2$ and $(F - \{0\}, \circ \times \circ, \circ^{-1}, 1)$ is an abelian group.

From algebra [325]:

A *boolean algebra* is a structure $(B, \circ \wedge \circ, \circ \vee \circ, \neg \circ, 0, 1)$ over a set B with a distinguished universal element 1, its complement 0, $\circ \wedge \circ$ and $\circ \vee \circ$ are commutative and distributive, 1 is the unique neutral element with respect to $\circ \wedge \circ$ and 0 the unique neutral element with respect to $\circ \vee \circ$.

2. This is just $\varphi\omega$ from Scott's theory.

allowed. Also missing is integer comparison as any function that performs a comparison between two numbers must by definition see the infinite whole of the number. Since a number's representation is by definition of infinite size, such a function is not continuous and so cannot be realized in a physical object.

Interpretation as Circuits

In the theory, a synchronous circuit is defined as an expression in terms of the four basis elements: the constants ${}_2(0)$ and ${}_2(1)$, the mux $?:{}_2Z^3 \rightarrow {}_2Z$ and the register $2x:{}_2Z^2 \rightarrow {}_2Z$. The two non-constant elements are defined by

$$?(c, a, b) = \begin{cases} b_t & \text{if } c_t = 1 \\ a_t & \text{if } c_t = 0 \end{cases} \quad 2x(i_t) = \begin{cases} 0 & \text{if } t = 0 \\ i_{t-1} & \text{if } t > 0 \end{cases}$$

Using the constants, the mux gate $?(c, a, b)$ is universal with respect to the boolean algebra; it can implement any of the boolean operators $\circ \wedge \circ$, $\circ \vee \circ$ and $\neg \circ$.

A circuit expression is composed using a set of variables $V(C) = I \cup M \cup R$ that represent the connections in the circuit. The set $I = \{i_1, i_2, \dots, i_k\}$ is the set of input variables, $M = \{m_1, m_2, \dots, m_l\}$ the set of mux output variables and $R = \{r_1, r_2, \dots, r_m\}$ the set of register output variables. The set of output variables O is just some subset of the overall variables: $O \subseteq V(C)$ and there may be no combinational cycles unbroken by a register.

Intuitively, every 2-adic number can be produced by an output-only synchronous circuit that produces the bits of the number one at a time starting at the least significant bit. Functions over the 2-adic numbers correspond to synchronous circuits that accept representations of 2-adic numbers and produce their results least significant bit first. Every synchronous circuit $C[[f]]$ with k inputs can be thought of as a function $f:{}_2Z^k \rightarrow {}_2Z$. Any expression $E[[f]]$ over terms in ${}_2Z$ can be thought of either in its synchronous circuit

interpretation or as a function. Vuillemin distinguishes three classes of functions, each a superset of the previous: bitwise mappings, online functions and continuous functions. As each class subsumes the previous, it is convenient refer to an individual function as belonging to the smallest possible class. With this in mind, the bitwise functions are straightforward. They use the boolean algebra aspect of ${}_2Z$ and correspond to a serial version of combinational functions. The two other classes of functions correspond to synchronous circuits.

The online functions are dependent at time $t = k$ only on the inputs that have been presented from time $t = 0$ up to $t = k$. The continuous functions on the other hand are more general. They are dependent at time $t = k$ on inputs that have (or will be) presented from $t = 0$ up to some time $t = m(k)$. Clearly every online function is also continuous. The class of continuous functions¹ can be viewed from a systems-theoretic perspective as non-causal systems; they are systems that anticipate future inputs. They are implemented by transforming them into causal systems through the introduction of delay. This delaying function is phrased as an “output enable” in the theory and amounts to an extra signal indicating to the user that the early junk produced by the circuit should be ignored.

It is a theorem that any online function can be implemented by some finite synchronous circuit expression $C[[f]]$ if a finite representation for its Synchronous Decision Diagram (SDD) can be computed. The SDD construction procedure may fail to terminate if the function is not online (thereby creating an infinite SDD). The SDD is a canonical form for synchronous circuits, and so form the basis for a physical synthesis procedure analogous to those using Bryant’s OBDDs for combinational circuits [496]. Further the retiming and resynthesis framework of Leiserson and Saxe [474] is directly applicable to such circuits,

1. The terminology and results from Scott’s theory about continuous functions is directly applicable here. The continuous functions correspond to the computable functions. A function which is not continuous is not computable

thereby connecting the bit-serial forms to the more conventional bit-parallel representations.

2.3.3 Focus

The preceding sections presented two examples of non-standard semantic models. These models, the Non-deterministic Event Sequence model and ${}_2Z$, expressions over the 2-adic integers, were not based (directly) on transition systems but rather on some more specific and detailed mathematical structure which highlighted a property of interest. In the first case that structure was a Turing machine-like model which operated upon nondeterministically shuffled event sequences. Instead of being a transformer of states-to-states as is the case in a state-transition system, the NES Machine is a transformer of sequences-to-sequences. The distinguishing characteristic being that the output of a transition can be a sequence of events which then forms the new input. In the second case that structure was a definition of integers, the 2-adic integers, which has been shown to have a direct analogy to combinational logic and serial sequential circuits. Unfortunately, while these non-standard models are tremendously interesting from an intellectual perspective, there is much less depth to them in the potentially application areas: simulation, synthesis and formal verification. They are featured here because these semantic models in particular are part of two very elegant examples of language definitions where the rigorously defined semantics contributes to the proposed use of the language. This contrasts markedly with the situations where the language semantics merely forms a neutral representation language for describing compiler design constraints.

2.4 Review

The preceding sections have surveyed various means for specifying programming language semantics and illustrated the tremendous breadth possible in semantic models. The theme throughout the presentation was that despite this breadth, for the case of finite state

synchronous systems, language semantics and semantic models ultimately boil down to a notion of transition relations. That is, the behavior of systems is ultimately defined in terms of states and the transitions between those states. In a formal sense, a program in a language is therefore a convenient and cogent representation of a transition relation acting between program states. This conclusion was arrived at through a chain of observations. The first observation established that a semantics is in fact necessary. Simply stated, this is because checking that a program obeys some behavioral property requires a formal notion of what the program computes. The definition of a computation is intrinsically some mathematical structure and the association of the program with a mathematical construct expressing its computation is the very definition of its semantics.

The second set of observations concerned the paradigms used to define language semantics. These were reviewed in Section 2.1 with the specific eye towards drawing parallels between the axiomatic, denotational and operational methods of semantic specification. While all three methods are on the surface quite disparate, providing differing and complementary views of semantics, when one looks deeper one always finds a notion of state and a program is viewed as a form of representation for transitions between those states. This was seen as an observation on the nature of computation within the constraints of finite-state. From that observation a series of semantic models were presented in Section 2.2. Each of those presentations focused on the essential aspects of the model but in doing so showed that underneath the distinguishing features there was always some notion, however latent, of state and state transitions. In contrast, two non-standard semantic models were shown in Section 2.3. These models did not have the state and state-transition aspects of the standard models. In contrast however the two non-standard models, the NES model and ${}_2Z$, the 2-adic integers, formed an elegant semantic basis for their respective languages UDL/I and 2Z.

At a fundamental level therefore abstract behavior for finite state systems is in some

fashion, defined in terms of states and state transitions. Given this definition for abstract behavioral components, the next question to be addressed is the specification of pure behaviors and the structural aggregation of them through coordination between components. The subsequent question to be addressed is whether there are in fact any limits on the amount and kinds of structure which can be introduced into a finite state semantic model. These two issues are the subject of the next two chapters.

3 Computational Semantics

A semantics is a formal means by which a mathematical object is associated with the textual representation of a program. The examples of the previous chapter established, in the form of an overview presentation, that for the finite-state case, semantic models ultimately boil down to a means for specifying states and the transitions between them and that there are a variety of such methods. That presentation established the base of the diagram of Figure 3-1, namely that the model $M = (Q, T)$ will at some level of detail be formulated in terms of a set of states Q and a transition relation T . Building up from that proposition, the next element of concern is the semantic map $S:L \rightarrow M$ and its internal properties relative to desirable properties in L and M . Of particular interest here are the aspects of S which aid or hinder any practical ability to manipulate model elements in symbolic form. These internal properties of S are the focus of computational semantics by which is meant the use of the semantic map directly as the basis for symbolic analysis of program properties on the state-based model $M = (Q, T)$.

Within the study of semantics itself, the major issues are whether full abstraction and full expressiveness are afforded by the combination of the semantic model and semantic map. In a theoretical setting these two issues are the major preoccupation in the design and analysis of language semantics. In a more pragmatic setting however our interest is not only in these two aspects but also the practical aspects of symbolic manipulation of

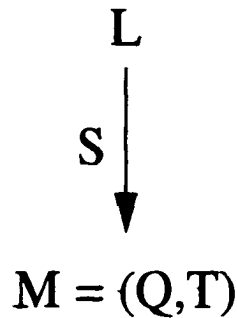


Figure 3-1. A Language, its Abstract Semantics and the Model

model element. In particular the interest here is in the competing constraints imposed on any semantics both from below and above. From below, that is from M upwards, there are pragmatic constraints on the feasibility of creating and manipulating certain kinds of symbolic representations. These constraints are shown here to be related to the desire for a fully abstract model. From above, that is from L downwards, there are constraints imposed by the desire to write cogent programs. Programs are written with a certain economy of exposition and a language which offers a more compact representation for the combinational elements of a design is typically viewed in higher regard than one which requires each potential alternative to be laid out in detail.¹ These constraints are shown here to be related to the need for full expressiveness but which are subject to the pragmatic desire for compact representation at the language level.

More concretely, the issue at hand is one of a tension between the ability to denote elements in M by a method of incremental approximation and the ability to ignore the concomitant mathematical baggage carried along by such an approach. This brings out the structure of a practical semantics as shown in the diagram of Figure 3-1.² In that view a

1. A more detailed series of observations on system description languages themselves must be deferred until Chapter 6 at which point the opportunities, limitations and alternatives in semantics will have been explored. There, a more detailed series of observations on language design can be made within the framework for semantic analysis that developed here and Chapter 4.

practical semantics S_{\circ} is a map which is non-abstract and identifies elements in a model M_{\circ} in a way which is somehow convenient relative to the language L . The condition of full abstraction is rederived through a projection $\Pi: M_{\circ} \rightarrow M$ which provides a way to ignore the extra baggage of M_{\circ} . The conditions for full expressiveness through this chain requires some analysis and in fact it is shown in Chapter 4 that there are certain fundamental limitations on the sorts of implementation details that the projection Π can hide.

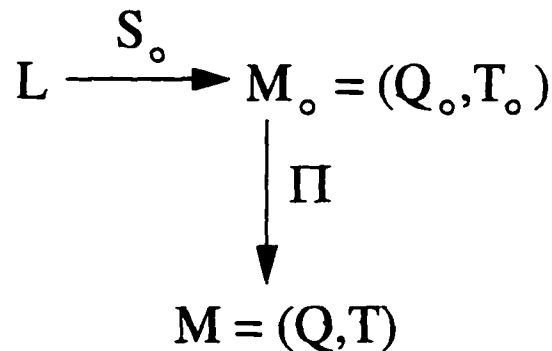


Figure 3-2. A Language, its Non-Abstract Semantics and the Model

The theory proposed here for computational semantics is based on the notion of approximating computations on a space of functions. Approximation in this context is measured by the closeness to completion of a step in M . Each \circ -step taken in M_{\circ} is a better approximation to a step in M in the sense that some metric of completion is maximized at the limit of the \circ -steps. The idea behind computational semantics is that it is best, conceptually and pragmatically, to express any single step in M by first defining it in terms of a series of much smaller steps in M_{\circ} . The projection operation Π is then used to formally abstract away these intermediate \circ -steps.

2. Where necessary the symbol " \circ " is used as a placeholder for the symbolic name of the microsemantics: e.g. \circ -time, \circ -step or X_{\circ} . The placeholder " \circ " becomes " δ " and " σ " in later sections. This use of " \circ " as a placeholder is distinct from its use as the composition operator: e.g. $f \circ g$ which is the function $\lambda x.g(f(x))$. The reader is warned that both uses of " \circ " appear in the following. The intended meaning is unambiguous in all cases.

The claim being made here is that a fully abstract semantics, *i.e.* the direct map $S:L \rightarrow M$ shown in Figure 3-1, is too great a leap to make in a practical sense. It is too difficult to construct a general S which establishes a direct link between an arbitrary $p \in L$ and its corresponding $m \in M$. In addition, producing and manipulating symbolic representations of the transition relation in such a direct regime has been shown to be problematic. What is needed to remedy this situation is a two-phase approach as is depicted in Figure 3-1. There, the first phase is the semantics S_0 which has a more or less intuitive correspondence to the operational execution of programs in L . Once this has been established in terms of \circ -step paths the second phase is the projection Π which extracts out the fully abstract single-step element in M .

In order to construct a system such as the one in Figure 3-1, several elements must be assembled. First, a theory of computation that incorporates a notion of approximation over function spaces must be found. Fortunately such a theory already exists in Scott's domain theory¹ and continuous functions between them. This theory allows the multi-step semantics (S_0) to be related to the single-step version (S) in terms of a fixed point on an underlying \circ -step approximation series. Secondly, this theory must then be incorporated into the relational transition semantics that was argued to be fundamental in Chapter 2. This incorporation allows the forward and backward image computations to be explained in terms of the least and greatest fixed points over the \circ -step series. These elements then constitute a theory of multi-level time which identifies the conditions which must prevail for the multi-step \circ -time to be substitutable in place of the single step macro time. This is the condition when $S = S_0 \circ \Pi$ holds and is the condition of substitutability.

1. Scott's original publications [640] [641] [642] [673] [643] contain the more theoretical treatment. The development here follows that of Gunter and Scott [309].

3.1 Semantic Domain Theory

Scott's domain theory is implicitly based on the idea of continuity from algebraic topology. Concretely this takes the form of establishing that every computable function is the fixed point of a series of better and better approximations to it. Under certain conditions this series is known to converge and one can claim at that point to have an exact characterization of the function. This is a familiar idea in the realm of numerical analysis where approximation, limit series and continuity allow for computational solutions to systems of differential equations. In this case, however the underlying domains are not of the real numbers but are function spaces which must be constructed so that the familiar notion of approximation, limit series and continuity exist.

Scott showed that the domain $\wp\omega$, the power set of the natural numbers, under a broadly-defined isomorphism construction, is large enough to hold all the computable functions while preserving the required relationships. Thus neither whether function spaces with the appropriate properties can be constructed nor the particularities of the isomorphism embedding of functions into $\wp\omega$ is in question here. Questions in that regard have been amply dealt with elsewhere.¹ What is of interest here is the particular use of domain theory in the definition of the multi-step model M_ω and the projection Π down onto the single-step model M .

The following sections provide the basic definitions and properties of Scott's domain theory. The issue of a limitation to computable functions in the definition of macro-step behavioral properties is returned to in Section 3.2.

3.1.1 Primitive Domains

Domains are countable sets endowed with an internal ordering. The ordering forms a complete partial order² which, when the basis elements are countable and the internal

1. *c.f.* Stoy [673], the technical reports cited therein and Scott's subsequent publications.

structure is algebraic, becomes a domain. The definitions presented below constitute the basic domain theory. They are presented here in fully general form, allowing for the sets to be both finite and infinite, in order to establish the non-utility of domain theory in defining fair macrostep behaviors. This is accomplished in Section 3.2. Subsequent sections show that certain of the careful limit constructions which are needed for the infinite case become superfluous in the finite case. In fact for the finite case, computability falls out almost directly. It is only in the infinite case that computability is endangered so some care must be taken.

The bases of domain theory are sets, functions and a relationship \sqsubseteq between elements.

The relationship $x \sqsubseteq y$ is understood to mean that “ y is more defined than x .”

A *partially ordered set* (poset) is a set D with an ordering \sqsubseteq which must be:

reflexive:	$x \sqsubseteq x$,
anti-symmetric	$(y \sqsubseteq x) \wedge (x \sqsubseteq y) \Rightarrow (x = y)$,
transitive	$(x \sqsubseteq y) \wedge (y \sqsubseteq z) \Rightarrow (x \sqsubseteq z)$.

The y in $x \sqsubseteq y$ is called an *upper bound* of x and may be written as $x \sqcup y = y$.

Of course $x \sqcup y = y \sqcup x$ and $x \sqcup x = x$.

As well the x in $x \sqsubseteq y$ is called a *lower bound* of y and may be written as $x \sqcap y = x$.

Of course $x \sqcap y = y \sqcap x$ and $x \sqcap x = x$.

A *directed subset* $M \subseteq D$ has the property that every finite $X \subseteq M$ has an upper bound within M : $\sqcup X \in M$. Therefore every pair of elements $x, y \in X$ has an upper bound $z \in M$ given by $z = x \sqcup y$ and the set $X = \{x_0, x_1, \dots, x_k\}$ of an increasing chain $x_0 \sqsubseteq x_1 \sqsubseteq \dots \sqsubseteq x_k$ is a special kind of directed set.

A directed subset M is *interesting* when it does not contain its upper bound: $\sqcup M \notin M$.

No finite directed set is interesting in this sense.¹ An element $z \in D$ is called a *limit point* when $z = \sqcup M$ for an interesting directed set M .

A *complete partial order* (cpo) is a poset D where:

there is a unique element $\perp \in D$ called *bottom*² such that $\forall x \in D. \perp \sqsubseteq x$ and, every increasing chain $x_0 \sqsubseteq x_1 \sqsubseteq \dots \sqsubseteq x_k$ has a *least upper bound* $\sqcup x_i \in D$.

2. The original derivation was based on complete lattices. More recent formulations of the domain theory are based on complete partial orders (cpo) which are similar to complete lattices only they need not have a \top element. The reasons for this switch have to do with certain technical aspects of the powerdomain construction. This is discussed in Scott [642]. This presentation follows the cpo approach.

1. Though some are thought to be *amusing* in an existentialist sense.

There exist *compact* (equivalently *finite*) elements e in the cpo such that for a directed subset M where $z = \bigsqcup M$ and $e \sqsubseteq z$ then there is also a $y \in M$ such that $e \sqsubseteq y$. Call the set of all compact elements $K(D)$.

A cpo is *algebraic* when every element y is the least upper bound of its compact elements. That is, when $M_x = \{e_i | e_i \sqsubseteq x\}$ then $x = \bigsqcup M_x$.

An algebraic cpo requires that the information which defines x come exclusively from the compact elements and not, for example, from the order in which the e_i are assembled. Further an algebraic cpo requires that a limit point x to be the upper bound of only finite elements (though infinitely many of such elements e_i may be required). From these definitions, a domain is defined as:

A *domain* D is an cpo which is algebraic and where the set $K(D)$ is countable.

These definitions are highly abstract and of note is that they apply equally to the finite case as to the infinite case. The only restriction is that the number of compact elements in the set must be countable. Two examples of *flat domains* are shown in Figure 3-3. In particular, any set can be made into a flat domain treating the set as the basis elements and adding \perp below every element of the set. Domains in this style are said to have a *discrete ordering* among the proper elements meaning that $x \sqsubseteq y$ if and only if $x = y$. They are simple in the sense that they have no directed subsets.

In contrast, a non-flat domain is one in which there are internal relationships among the elements. In this case the domain elements consist not only of the basis elements but also of elements which are the upper bounds of other domain elements. Such domains have directed subsets. An example of this kind of domain is the power set 2^S of a countable set $S = \{s_1, s_2, \dots, s_n\}$. The empty set \emptyset corresponds to the element \perp , the compact ele-

2. Barendregt [54] also refers to \perp as *unspecified* (page 325). Either sense refers to \perp as the unique representation of "no information." Also, \perp is often called the *improper element* of the cpo since it need not have a material representation. In an implementation setting \perp may or may not require a tangible form subject to certain conditions. Kahn's Dynamic Process Network model [414] [415] gives the conditions preventing explicit computation with \perp in a stream-based communicating process model.

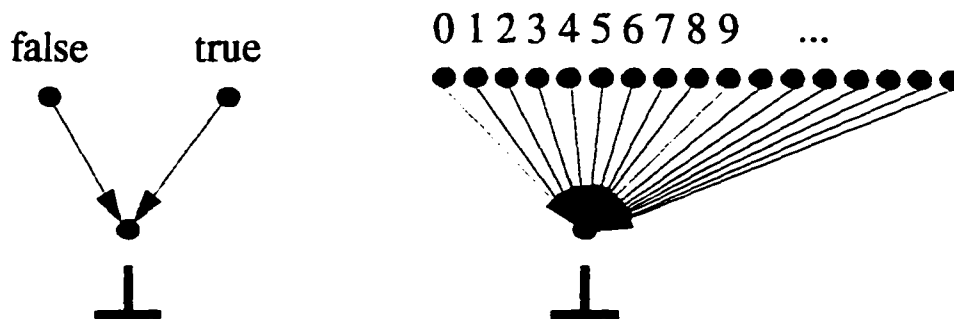


Figure 3-3. Examples of Flat Domains on Countable Sets

ments are the non-empty subsets $U \in 2^S$ and the ordering \sqsubseteq is merely subset inclusion \subseteq . Such a domain for the basis set $S = \{a, b, c, d\}$ is depicted in Figure 3-4. The same construction can be applied to the countably infinite case as well. One example in that case is $\wp \omega$, the power set of the natural numbers ω . Here the compact elements would be the set of finite subsets of $\wp \omega$ and the ordering \sqsubseteq is again subset inclusion \subseteq . A second example, one which is more commonly used, is the set of infinite sequences from an alphabet: $w \in \Sigma^\omega$. The empty sequence ε corresponds to \perp , the finite strings $s \in \Sigma^*$ are the compact elements of the domain and the ordering $x \sqsubseteq y$ means that x is a prefix of y (i.e. $\exists z \in \Sigma^*.xz = y$).

The paradigm in domain theory is the use of domain equations to define the result of a computation. The execution of a program is then said to “solve” these domain equations. A crucial concern therefore is the existence of a (non-trivial) solution to an arbitrary set of domain equations. Fortunately, there exists a universal domain into which any other domain can be embedded.¹ Conversely, one can also retract any domain out of this universal domain by defining a function which is the identity on the elements of interest and

1. The encoding and embedding construction is called the *Inverse Limit Construction*, an outline of which can be found in Stoy [673].

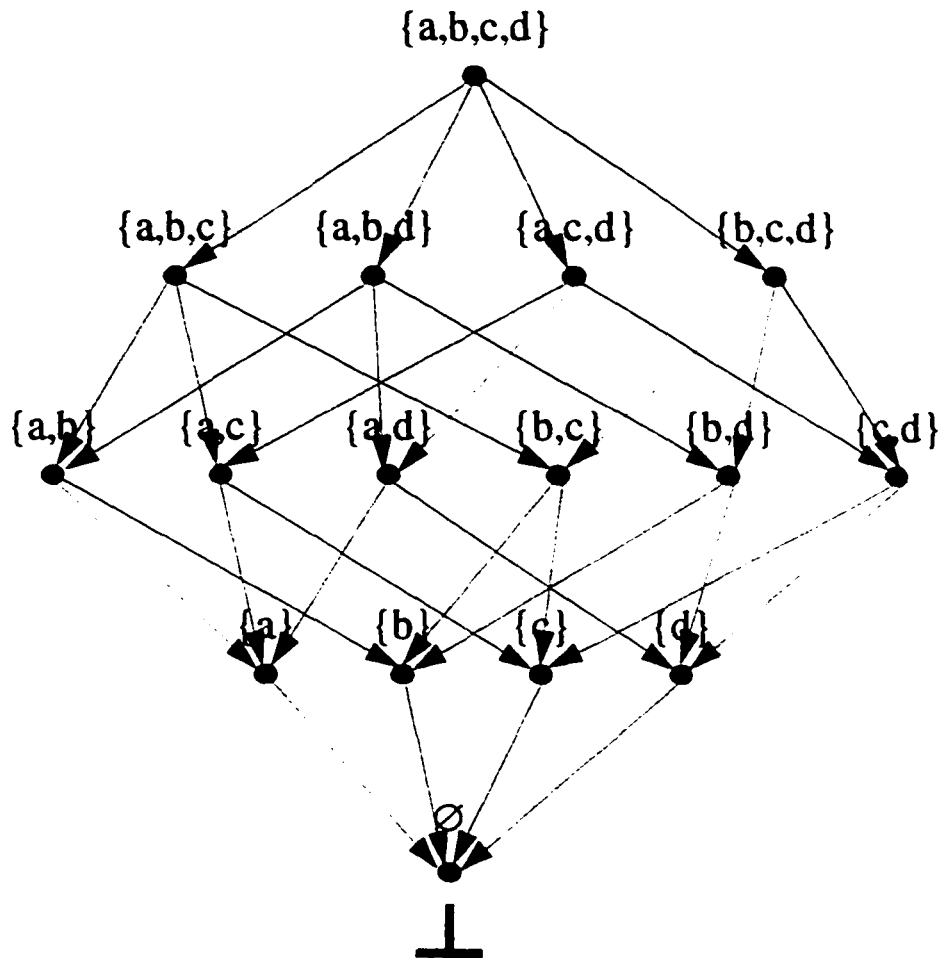


Figure 3-4. The Domain on the Power Set of $S = \{ a, b, c, d \}$

sends all other elements to \perp . The universal domain is $\wp \omega$, however it is only significant for this presentation in the sense that its existence ensures that nontrivial solutions to domain equations exist. On a practical level it means that one can posit domain equations, subject to some broadly defined limits, which produce whatever internal structure suits the task at hand with the confidence that there do exist domains which satisfy the equations. The ability to embed an arbitrary domain within $\wp \omega$ or to retract any domain of interest out of $\wp \omega$ is the strong result which allows the topological aspects of domain theory to be separated from the engineering aspects of designing a semantics in terms of domain

equations. By way of (an almost direct) analogy, the topological aspects of real number theory is effectively separable from the engineering aspects of designing computational methods to solve systems of differential equations.

3.1.2 Functions on Domains

Functions can be naively understood in completely set-theoretic terms as a set with a special kind of structure [427]. Since the sets outlined above can be potentially countably-infinite some restriction must be made to ensure that the class of functions definable in this manner is not too large to be of use. An obviously reasonable requirement, given the algebraic property of a domain, is that a function be such that for any input value, an enumeration of the compact elements of the output can be produced by an enumeration of the compact elements generating the given input. A further requirement that finite approximations in the input correspond to finite approximations of the output. Thus any infinite result, a limit point, can be approximated arbitrarily accurately through an enumeration and assembly of arbitrarily many finite estimates. Any element for which no such finite enumeration and approximation scheme is possible is necessarily *incomputable*. The restriction that allows for this decomposition and enumeration is called *continuity* and is given by the following definitions:

A *function* is a map from one domain to another. Such a map is merely the set of pairs for which the function is defined: $f = \{ (d, e) \mid d \in D, e \in E, e = f(d) \}$. The notation $f: D \rightarrow E$ is used to denote the function f which maps elements in the domain D to elements in the domain E .

A function f is *monotonic* when $x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$; it preserves or increases the relative order of x and y .

A *continuous* function f is a monotonic function where $f(\bigsqcup x_i) = \bigsqcup f(x_i)$ holds. Any function f on an element y made up of compact elements e_i as $y = \bigsqcup e_i$ can be determined by computing the function f on the compact elements and taking the least upper bound of the result.

It is worth again observing that no finite directed subset M of a domain D is interesting and that in every uninteresting directed subset, it is the case that for a finite subset $X \subseteq M$

where $X = \{e_i | e_i \in K(D)\}$ there exists a $y \in M$ such that $y = \bigsqcup e_i$. Further, every such element y under a function f has the property that $f(y) = f(\bigsqcup e_i)$. When f is continuous then $f(\bigsqcup e_i) = \bigsqcup f(e_i)$. But for the finite domain D there are no interesting directed subsets M , so it is never the case that $y \in M$ but $y \notin X$. For finite domains it is always the case that $y = \bigsqcup e_i$ and $y \in X$; the monotonicity conditions guarantee that $f(y) = \bigsqcup f(e_i)$. Thus for finite domains every monotonic function is also continuous. It is only in the infinite case, at the limit points, where continuity says something extra about f . There, continuity says that f at the (infinite) limit point is completely determined by the (infinite) limit of finite approximations.

3.1.3 Fixed Points of Functions

Fixed points of functions bring the approximation theory of the previous section up to the level of functions. A fixed point is a point, $x \in D$ where a function $f: D \rightarrow D$ has the property that $x = f(x)$. An arbitrary function f on an arbitrary domain D may have zero, one or a multiplicity of fixed points. The utility of fixed points in the approximation theory is that a monotonic function f coupled with the ordering relation \sqsubseteq give some measure how well any given estimate \hat{x} approximates a given x which is a fixed point of f . Intuitively if \hat{x} is said to approximate x then it must be the case that $\hat{x} \sqsubseteq x$ (or $x \sqsubseteq \hat{x}$, but consider the former case first). Then f gives a way of arriving at a better approximation, namely $f(\hat{x})$. Because of monotonicity if $\hat{x} \sqsubseteq x$ then $f(\hat{x}) \sqsubseteq f(x)$, however because x is a fixed point of f then $f(x) = x$ and $\hat{x} \sqsubseteq f(\hat{x}) \sqsubseteq x$ holds. The same chain of reasoning applies for the $x \sqsubseteq \hat{x}$ case.

Tarski [681] showed that when f is monotonic then there is a unique *least fixed point* μf that is the upper bound of all the estimates of it starting from the unique improper element \perp (starting from no information at all). In addition when there exists a unique improper element \top (the “inconsistent” or “overdefined” element) then there is also a *greatest fixed point* νf as well. It is the lower bound of all the estimates of it start-

ing from \perp . These are defined as:

$$\mu f = \bigsqcup_{i=0}^{\infty} f^{(i)} (\perp) \quad (\text{Eq 3-1})$$

$$\nu f = \bigsqcap_{i=0}^{\infty} f^{(i)} (\top) \quad (\text{Eq 3-2})$$

One can additionally speak of the *relative* least and greatest fixed points. These are just the least and greatest fixed points from Eq 3-1 and Eq 3-2 except the starting point for the series is some other previously-agreed upon point p . To highlight this subtle difference one writes $\mu_p f$ and $\nu_p f$:

$$\mu_p f = \bigsqcup_{i=0}^{\infty} f^{(i)} (p) \quad (\text{Eq 3-3})$$

$$\nu_p f = \bigsqcap_{i=0}^{\infty} f^{(i)} (p) \quad (\text{Eq 3-4})$$

The respective domain elements \perp and \top have the unique advantages that they are minimal (respectively maximal) relative to \sqsubseteq and so the fixed points $\mu_{\perp} f$ and $\nu_{\top} f$ approximated from them are truly the smallest (respectively the largest). Further, since they are the extreme elements in the domain, the series Eq 3-1 and Eq 3-2 is guaranteed to find a fixed point when it exists. The danger with Eq 3-3 and Eq 3-4 computing $\mu_p f$ and $\nu_p f$ is that necessarily $\perp \sqsubseteq p$ (respectively $p \sqsubseteq \top$). It may be the case that the only fixed points of f are elements q_i and every $q_i \sqsubseteq p$ (respectively every $p \sqsubseteq q_i$). Thus Eq 3-3 and Eq 3-4 are only valid when it is known *a priori* that there exists at least one fixed point directionally beyond p .

In a practical setting where domain equations are being analyzed symbolically, the use of some other well known landmark point p may be appropriate because such an *a priori* analysis of the domain space can be accomplished before choosing a particular p . As well the use of Eq 3-3 and Eq 3-4 obviates the need to have an explicit representation for \perp or \top in the symbolic form. The series approximates just as well from p though the fixed

points found will not be the least (respectively greatest) in the absolute sense. They are especially convenient when there is no \top element defined for the domain as is the case for domains based on *cpos*!

3.1.4 Functions as Fixed Points

Fixed points of functions defined on the primitive domains were introduced in the previous section. This allowed elements in a primitive domain to be computed by a series of approximations. The monotonic function f increased the accuracy of the approximations until the fixed point was reached. The least and greatest fixed points relative to a landmark point were defined. Scott showed that the continuous functions are a domain as well, so conceptually at least, a function can be computed by successive approximation, given an appropriate notion of \sqsubseteq , just as certain elements of the primitive domains can be computed. This sort of fixed point is a solution to the functional¹ equation $f = F \{f\}$ for the function f and the monotonic functional F .

Such an equation implicitly depends on the existence of an ordering relation $f \sqsubseteq g$ which identifies how well the function f can be said to approximate g . Scott showed, among other things, that such a relation \sqsubseteq exists and is well defined. Thus the set of continuous functions between two domains is itself a domain. The natural ordering relation for continuous functions is derived directly from the ordering relation on the function's domain elements:

$$f, g \in D_1 \rightarrow D_2 \quad f \sqsubseteq g \equiv \forall x \in D_1. f(x) \sqsubseteq g(x) \quad (\text{Eq 3-5})$$

1. A *functional* is a function taking a function as an argument. It is a higher-order function. Notationally functionals are written here as $F \{f\}$ instead of $f(x)$. This is purely a matter of taste and is used to give the reader some mild hint about the type of the elements.

Thus an equation such as $f = \mu_{\perp} F$ over $f: D \rightarrow D$ and $F: (D \rightarrow D) \rightarrow (D \rightarrow D)$ is said to have the meaning:

$$f = \mu_{\perp} F = \bigsqcup_{i=0}^{\infty} F^{(i)} \{\perp\}$$

When F has a fixed point, then f can be said to be the smallest function satisfying the equation $f = F \{f\}$. As well, if $Id: D \rightarrow D$ is the identity function then an equation such as $f = \mu_{Id} F$ is said to have the meaning:

$$f = \mu_{Id} F = \bigsqcup_{i=0}^{\infty} F^{(i)} \{Id\}$$

When F has a fixed point greater than Id , then f can be said to be the smallest function relative to Id which satisfies the equation $f = F \{f\}$.

The greatest fixed point works similarly relative to \top (when it exists) and Id .

3.1.5 Constructed Domains

The final aspect of the domain theory of interest here are non-primitive or constructed domains. The primitive domains were introduced in Section 3.1.1 as special sorts of sets which had an ordering relation \sqsubseteq on them. They were called primitive domains because the existence of the set was postulated at the same time as its ordering \sqsubseteq was defined. It is also possible to create new domains out of combinations of already-existing domains. This is done with the aid of *domain constructors* and *projectors* which respectively create elements of the new domain from one or more elements of the component domains and which identify the component elements in the new the aggregate domain. There are a wide variety of non-primitive domains, each modeling some necessary aspect of computation.¹ Whereas the primitive domains came with their own externally-supplied ordering relation \sqsubseteq , constructed domains must define their ordering relation subject to the order-

1. Mosses [545] and Gunter and Scott [301] together present a more complete set of domain constructors and their properties. There are a number of possibilities and alternatives. Only the domain constructors which are required for the later sections are presented here.

ing of their constituent domains. The constructed domains used in the subsequent development are:

$D = D_1 \times D_2 \times \dots \times D_n$ constructs a new Cartesian product domain of n -tuples.

The constructor function is $(d_1, d_2, \dots, d_n) : D_1 \times D_2 \times \dots \times D_n \rightarrow D$, and for every D_i there exist unique projection functions $\pi_i : D \rightarrow D_i$.

$(x_1, x_2, \dots, x_n) \sqsubseteq_D (y_1, y_2, \dots, y_n)$ just when $\forall 1 \leq i \leq n. x_i \sqsubseteq_{D_i} y_i$.

$D = D_1 \otimes D_2 \otimes \dots \otimes D_n$ is the so called “smash” product.

This is exactly like the Cartesian product except that n -tuples with any elements $\perp \in D_i$ are identified with $\perp \in D$. The smash product preserves the flatness of domains.

$D = D_1 + D_2 + \dots + D_n$ constructs a new “separated” sum domain.

Elements of D are replicas of all elements from D_i and there is a new $\perp_D \in D$. $x \sqsubseteq_D y$ just when $x \sqsubseteq_{D_i} y$ and elements which originating from different summands are incomparable. The constructor functions are the injection mappings $\varphi_{D_i \rightarrow D} : D_i \rightarrow D$; it may be elided without ambiguity. The projector functions $\pi_{D \rightarrow D_i} : D \rightarrow D_i$ are well defined; they are the identity when the argument originated in D_i and return \perp_{D_i} otherwise.

$D = D_1 \oplus D_2 \oplus \dots \oplus D_n$ constructs a new “coalesced” sum domain.

It is exactly like the separated sum except that elements of D are replicas of all elements from $D_i - \{\perp\}$ and the $\perp \in D_i$ are all identified with $\perp_D \in D$. The coalesced sum preserves the flatness of domains.

$D = D_1 \rightarrow D_2$ is the domain of continuous functions between D_1 and D_2 .

The functional constructor is λ -abstraction which is written $\lambda x \in D_1. formula$, and the functional accessor is application written as $f\{x\}$ for $f \in D$ and $x \in D_1$.

As before: $f \sqsubseteq_D g \equiv \forall x \in D_1. f(x) \sqsubseteq_{D_2} g(x)$.

$D = \wp D_1$ constructs the natural powerdomain¹ over D_1 .

Elements in domain are sets of elements of D_1 representing the *potential* for the computation to be. Powerdomains are used in to represent nondeterminism, and by extension concurrency.

The definition of $u \sqsubseteq_D v$ is crucial to the powerdomain construction. It is related not only to the subset relation on the power set of D_1 but is also subject to the ordering $x \sqsubseteq_{D_1} y$. The natural order $u \sqsubseteq v$ holds when every element in u has a refinement in v and every element in v is a refinement of some element of u :

$$u \sqsubseteq_D v \equiv \left(\forall x \in u. \exists y \in v. x \sqsubseteq_{D_1} y \right) \wedge \left(\forall y \in v. \exists x \in u. x \sqsubseteq_{D_1} y \right)$$

1. Also called the convex or Plotkin powerdomain. It distinguishes nonterminating computations (*i.e.* elements of D containing \perp_{D_1}) whereas the upper and lower powerdomains either ignore such elements or identify them with \perp_D respectively.

These domains are known to exist because they can be retracted out of the universal domain. As well, each of the domain-specific constructor and accessor functions are known to exist. The key questions about domain constructors for computational semantics are how \perp is treated in the new domain relative to the old and whether the constructor defines a new class of isolated elements.

Isolated Elements

A feature of central importance in the computational analysis of domain equations are the existence of *isolated points*. These are points in a domain which cannot be approximated. They are considered isolated because there are no nearby points under \sqsubseteq other than \perp , which is of little use computationally. In the flat domain examples shown in Figure 3-3, all of the domain elements are isolated because the relation \sqsubseteq treats each value as a distinct and unrelated quantity (e.g. in \aleph_{\perp} of that figure, the value 3 is entirely unrelated to the value 4; that $3 \leq 4$ also holds is irrelevant because \leq is not \sqsubseteq). In contrast, in the non-flat domain example shown in Figure 3-4, the singleton sets $\{a\}$, $\{b\}$, $\{c\}$ and $\{d\}$ are isolated while the aggregate sets are not (e.g. value $\{a, c\}$ can reasonably be said to be an approximation of $\{a, c, d\}$ under the relation \sqsubseteq).

Isolated elements are identified through a relationship $x \ll y$ which is understood to mean that “ x is an essential component of y .” Formally $x \ll y$ holds when for $y \sqsubseteq \bigsqcup Z$ then there is a $z_i \in Z$ such that $x \sqsubseteq z_i$. An isolated point is a point $a \in D$ such that $a \ll a$; it is its own essential component.

The existence of isolated points becomes relevant in the computational analysis setting because their existence necessarily means that there is no way to approximate them: they must be computed explicitly and completely or not at all. Whereas a non-flat domain has some isolated points among many others, a flat domain necessarily consists only of isolated points. This situation can potentially become even more extreme when domain con-

constructors are considered. Whether these constructors create a flat space with isolated points or the induce some internal structure is an important design decision at the semantic level. It relates directly to the applicability of approximations and limit series in the symbolic analysis of domain equations.

3.1.6 Focus

The previous sections introduced the basic aspects of Scott's domain theory as it is used here. A denotational theory using domains is given in terms of a certain number of primitive domains, each of which comes complete with its own \sqsubseteq relation and minimal element \perp . There are domain constructors which create new aggregate domains out of previously-existing ones. An aggregate domains defines its own \sqsubseteq and \perp in a way which is consistent with its component domains. The information ordering \sqsubseteq gives rise to a notion of approximation wherein one domain element can be said to be an estimate of another. Under the direction of a monotonic functional, an estimate can be improved relative to \sqsubseteq until a fixed point is reached. The fixed point represents the completed computation of an exact representation of the domain element.

The domain theory, as presented in this section, was necessarily abstract. It remains for subsequent sections to apply these ideas. In particular, two constructions are shown. The first is a behavioral domain of bisimilar behaviors. It is defined in the following section. That example illustrates why domain theory cannot used to define fair behaviors: fairness in the limit is discontinuous. The second example, which is the subject of Section 3.3, is the exposition of a domain which supports a fixed point formulation of the forward and backward image computations in δ -time. This domain construction is then generalized in Section 3.4 to allow for a theory of time under which various classes of micro-time step paths can be aggregated into a single macrotime step.

3.2 Behavioral Domains

Domain theory is intrinsically limited by its ability to express only computable functions. The direct implication is that any denotational definition of behavior is necessarily restricted to ones which can be approximated as the infinite limit of finite observations [595] [597]. This is of particular interest because the domain theory is shown in this section fails to be a useful and practical tool for the definition of behavioral properties at the macro-step level. This is because the definition of macro-step behaviors requires that the abstraction mechanism of (unbounded) non-determinism in conjunction with fairness constraints be available.¹ Practical examples of the use of fairness in property specifications were illustrated in Section 2.2 in the case of fair model checking and of course, fairness constraints are intrinsic to language containment between ω -automata.

Fairness constraints are necessary at the macro-step level because nondeterminism, while a wonderful abstraction tool, typically abstracts too much. Some sort of *post hoc* constraint must be imposed to ignore (or conversely to require) certain infinite behaviors. Fairness constraints are therefore constraints on infinite behavior that necessarily do not exist in any finite approximation. This is true even in the finite state case because even finite state systems manipulate an infinite quantity: time. Time is a (countably) infinite sequence and fair behaviors are properties defined across the whole of that ω -path. To see the inability of a domain-based semantics to express infinite fair behavior one can examine behavioral domains on sequential machines. Gordon [298] outlines a denotational semantics for sequential machines which is fully abstract with respect to bisimulation equivalence and which can be used effectively to illustrate this limitation. This construction is useful as well because the paradigm of approximation used in this section is

1. Strong fairness requires that a process which is enabled infinitely often executes infinitely often. Complete treatments of fairness can be found in Francez [266], or more the accessible overviews of Emerson [248], relative to computational logics, and Kurshan [454], relative to the classes of ω -automata.

repeated in the ω -time approximations of microsemantics in Section 3.3.

3.2.1 The Behavioral Domain $B = IN \rightarrow (OUT \times B)$

A sequential machine is a structure M with behaviors defined over two finite domains IN and OUT . Intuitively, the domains IN and OUT are the machine's input and output respectively. They may, in general, be flat domains as no internal structure within them is required. The structure M is therefore defined as:

$$M = (S, O, N, s_0)$$

where:

S is a flat domain of states,

$O: IN \times S \rightarrow OUT$ is a function producing outputs given an input and a state,

$N: IN \times S \rightarrow S$ is a function determining the next state given an input and a state,

$s_0 \in S$ is the initial state.

A behavior of such a machine is an element of the domain satisfying the equation $B = IN \rightarrow (OUT \times B)$. Thus a behavior is a function which, given an input, produces both an output and a further behavior. Full behaviors are infinitely long chains of these input-to-output-and-successor functions and approximations of full behaviors are therefore finite chains of such functions.

The elegance of the denotational method comes in the realization that an element $b \in B$ is an infinite object yet it can be derived by a series of approximations according to the series:

$$b_0 = \perp$$

$$b_{n+1} = \lambda i. (\pi_1(b_n, i), \pi_2(b_n, i))$$

and:

$$b = \bigsqcup b_i$$

It is shown¹ that for a particular machine M that there exists the particular function $f_M: S \rightarrow B$ which transforms a state machine into a behavior as:

$$f_M = \mu_{\perp} F \quad F = \lambda f. \lambda s. \lambda i. (O(i, s), f\{N(i, s)\})$$

This representation which, while notationally complex, is merely the definition of the simulation relation given in recursive functional form. Gordon shows that this is equivalent to the relational form which was presented in Section 2.2.2. The key insight behind the functional formulation of the bisimulation relation is that the material representations of the function f_M can be constructed iteratively from the representations of O and N which came with M . Further, given two machines M and M' , their behavior functions f_M and $f_{M'}$ can be constructed iteratively and their bisimilarity proved by induction on n which is the length of paths from each machine's respective start state.

3.2.2 The Problems with B

The previous section showed a conceptually simple formulation of behavior which uses successively more accurate finite approximations to characterize the temporal behavior of M over time. This definition of behavior is elegant, implementable and has the benefit of full abstraction. Unfortunately however, this formulation of behavior is fundamentally limited in four ways. First the definition takes into account only deterministic sequential machines. It does not take into account nondeterminism either in the output function O or in the next state function N . Such nondeterminism has been shown to be extremely useful in abstracting complex deterministic machines into much simpler nondeterministic machines.² Secondly, it formulated in terms of a single monolithic machine and offers no obvious means of defining the coordinated concurrent behavior of two or more machines.

1. The proofs are given in Gordon [298]. They are straightforward and their mechanics are not of particular interest here.

2. Deterministic finite automata may be exponentially larger than their deterministic counterparts. The Rabin-Scott subset construction can be used to determinize nondeterministic finite automaton [368] (a language recognizer). Some care must be taken when reformulating this construction for the transducer case (*i.e.* for Mealy machines) because determinization is not in general possible in that case [439].

Both of these problems could be repaired by reformulating the problem in terms of an appropriate powerdomain construction,¹ albeit at the cost of much greater notational burden. Thirdly, and of direct interest to formal verification, is that the formulation of ‘a behavior’ as a domain $B = IN \rightarrow (OUT \times B)$ fundamentally disallows the use of fairness constraints to simplify the representation of elements $b \in B$. This is because fair executions are discontinuous functions and thus cannot be approximated by any finite limit series.²

The first three limitations deal with the *ability to express* behavior in a denotational model so in a sense they are fundamental limitations to the denotational method of specifying behavior. Denotational specifications are necessarily restricted to elements which can be defined in terms of computable functions. Pragmatically this means a limitation to definitions of behavior that can be described cogently which in turn precludes describing (bounded) nondeterminism and concurrency. Thus the denotational definition of behavior is limited both in the kinds of behaviors that can be prescribed as well as in the compactness of those descriptions.

This intrinsic limitation has led the formal verification researchers to concentrate on

1. Mosses summary of the denotational method [545] illustrates the techniques used for adding (bounded) nondeterminism and interleaving-based concurrency to the denotational semantics of a simple imperative programming language. These techniques could be applied directly to this formulation although powerdomains are notoriously difficult to manipulate notationally. Additionally there are substantial impediments to full abstraction when resumptions are used as the concurrency operator (*c.f.* Mosses specific comments, page 625, on the ease with which Plotkin’s Structural Operational Semantics extends to the nondeterministic and concurrent case).

2. Owicki and Lamport [572] explain that denotational models have historically not been used as the basis of formal verification schemes, their proof lattice method included, precisely because the denotation of a fair scheduler is a function which is discontinuous in the limit.

In turn, Hojati and Brayton’s [358] denotational semantics for the Combinational/Sequential model has a syntax (of “tables”) denoting transition and initial state relations. These relations in turn are said to denote some ω -language. The limit construction, if given, would have followed Kahn’s ω -sequence construction. It is left to the reader. A subset of the denoted ω -language is called *accepted* just when it matches the externally supplied set of fairness constraints.

The key is that fairness is necessarily a constraint supplied independently of the denotations constructed by the semantics.

non-denotational methods of specifying behavior though possibly taking advantage of the finite approximation theory of denotational semantics for defining intra-behavioral computations. That is, the use of denotational models is restricted to the definition of the computations which constitute a step; these computations are necessarily finite and therefore computable. In contrast, the specification of behavior at the trans-step level is accomplished in some other manner (*i.e.* one which uses fairness constraints).

The focus on the intra-step computations introduces the fourth limitation which deals with *feasibility of representation*, namely the feasibility of representing $O:IN \times S \rightarrow OUT$ and $N:IN \times S \rightarrow S$ independent of whether a behavior is defined in terms of bisimilarity or by some non-continuous means. In practice the mere representation of O and N have been problematic because of the state explosion problem. That is, in practice, one is not given a sequential machine M directly but rather a set of coordinating components M_i which *implicitly* form a product machine $M = \prod M_i$. Thus without any further information about any intra-step structure, the denotations of O and N are fully abstract and their representation in a material data structure is problematic.

3.2.3 Focus

Domain theory and denotational semantics works off the premise that any computable function can be approximated to an arbitrarily accurate level and that the exact result can be obtained by taking the least upper bound over all such approximations. This is a reasonable restriction when the denoted functions are to be *executed*. Instead, if the functions are to be *analyzed*, then it is more reasonable to want incomputable functions as the denotations of behavior, subject to certain constraints. The example of this section illustrated the need for incomputable functions in the context of a behavioral domain B . A second observation made in the definition of the behavioral domain B was that it was dependent on the existence of the uninterpreted functions O and N . These functions were expected to exist after the product machine M was constructed, yet the denotational semantics

gave no hint about how they might be assembled from the O_i and N_i of a group of component machines M_i .

These two elements point towards an application of domain theory to the computations *within* a step wherein a (macro) step is actually approximated by a series of smaller steps. The following section shows that the macro step is adequately approximated by a series of smaller steps which exactly equal to a macro step when a certain fixed point is reached. This view is new. It is significant because it forms the basis for a theory of time where there is a mathematically-based definition of \circ -time. In particular this sort of \circ -time is derived from the sound principles of algebraic topology of the domain theory and is defined independently of any simulator event loop algorithm.

3.3 Microsemantic Domains

The microsemantic domains of \circ -time can now be stated. These domains allow for a macrostep to be approximated by a series of \circ -steps with the equivalence between the exact and the approximate forms occurring at the fixed point of an approximating functional. The construction first establishes a functional “wrapper” for the non-directed transition relation which, in Chapter 2, was argued to be the basic component of any finite state semantics. This functional connection induces a direction onto the transition relation in the form of the forward and backward image computations, $F\{Q\}$ and $B\{Q\}$, and establishes the notion of full abstraction relative to these computations alone. By defining full abstraction relative to $F\{Q\}$ and $B\{Q\}$ the problem of the previous section, the definition of trans-macrostep behavior in terms of computable function is neatly side-stepped. By limiting the semantic analysis to the image computations alone, trans-macrostep behavior can be defined in whatever way is convenient and appropriate. This explicitly makes a place for behaviors defined with the aid of incomputable concepts such as fairness. The focus here is exclusively within a step.

With the definitions of the fully abstract image computations in hand it is then possible to reestablish their definition by approximation as $\tilde{F}\{Q\}$ and $\tilde{B}\{Q\}$ in terms of the fixed point of their respective *approximator functionals*. These are $\mathcal{F}_\circ\{\tilde{F}\}$ and $\mathcal{B}_\circ\{\tilde{B}\}$ which produce successively better approximations to \tilde{F} and \tilde{B} respectively. Certain properties of the microsemantic domains are shown to be required for these approximator functionals to be well defined and to have a least (respectively greatest) fixed point. The steps of the approximating series that they generate are shown to be a topologically rigorous definition of \circ -time. It then remains to show that the respective fixed points $\tilde{F} = \mu\mathcal{F}_\circ$ and $\tilde{B} = \nu\mathcal{B}_\circ$ of the approximator functionals are comparable to F and B of the fully abstract case: namely that $\tilde{F}\circ\Pi = F$ and $\tilde{B}\circ\Pi = B$. Thus the computation of the fixed point of an approximator functional is substitutable for the fully abstract single step image computation when its projection onto the fully abstract space is the same functional as the fully abstract single step one. This is the condition of substitutability.

3.3.1 Functional Domains in a Relational World

In order to set the stage for the construction of the microsemantic domains, an apparent contradiction must first be addressed. The domain theory is based exclusively on function spaces, albeit spaces of computable functions, but spaces of functions none the less. In particular, domains are not spaces of *relations*. However, in Chapter 2 it was argued that the fundamental denotation throughout all formal methods was the transition relation. In fact, the bulk of that chapter was dedicated to a survey of formal methods which highlighted the relational aspects of each. Thus, some explanation is required.

The basis for the concentration on the transition relation was its unique status as the existential definition of pure behavior in a step. Indeed, the transition relation is conceptually but an exhaustive enumeration of the allowable moves in the system. It is fundamentally a static entity. This is both a strength and a weakness. The strength lies in the fact that a relation has no preferred direction. It describes transitions in a forward direction as well

as in the backward direction. The strength is that the ability to compute the set of states which preceded a certain set of states is crucial in the symbolic analysis of finite state systems. The weakness in the relational approach relates directly to the task at hand: domain theory is a theory based on continuous function spaces which is a concept that does not readily transfer over to relations directly. Thus some means of wrapping a transition relation inside a directional functional interpretation must be given. This will explain away the seeming disparate nature of the transition relation approach argued in Chapter 2 and the functional approach needed here for the microsemantic domain construction. That functional wrapper is exactly the functional of the forward and backward image computation: $F\{Q\}$ and $B\{Q\}$.

Earlier in Section 3.1.5 the domain of computable functions $D = D_1 \rightarrow D_2$ was defined. Its constructor was the λ -abstraction $\lambda x \in D_1. formula$ which created a new function in D . At that point the definition of a *formula* was left vaguely defined. A more concrete definition is given now to establish the direct connection between the (static) transition relation approach of Chapter 2 and the (dynamic) functional approach required for the domain theory.

A *formula* is a sentence in the μ -Calculus. This representation gives the direct connection between the relational and the functional approach since relations in the μ -Calculus are denoted by their characteristic function. The *formula* in the calculus takes the transition relation and recasts it directionally thereby defining either the forward or the backward image computation. The exact definition of these sentences is deferred until Section 3.3.2. The indirection step, encapsulating the transition relation within *formula*, has the benefit that different functionals based on the same transition relation can be used for the two different execution directions. It has the additional benefit that the static transition relation becomes symbolically operational because sentences in the μ -Calculus can be easily translated into computational recipes using OBDDs.¹

3.3.2 The Relational μ -Calculus

Strictly speaking the μ -Calculus¹ is a logic and thus has associated notions of complexity, expressiveness and decidability.² The interest here is not in those relative notions at all but rather in an absolute sense, in the use of the μ -Calculus as a notation for describing the image computations in finite-state systems. The constituent elements of the logic are given directly.

The μ -Calculus consists of sentences over two variable sets RV and BV where:

RV is a set of relational variables,

BV is a set of individual (or Boolean) variables.

A formula is a sentence classified either as Boolean formulae or relational formulae according to two sets of rules.

The Boolean formulae:

- *true* and *false* are Boolean formulae,
- if $x \in BV$ then x is a Boolean formula,
- if p and q are Boolean formulae then so are $p \vee q$ and $\neg q$; for notational convenience, $p \wedge q$, $p \Rightarrow q$, $p \equiv q$ and $q \oplus q$ are also Boolean formulae with the traditional interpretation,
- if p is a Boolean formula and $x \in BV$ then $\exists x.p$ is a Boolean formula; for notational convenience, $\forall x.p$ is also a Boolean formula with the traditional interpretation,
- if $R \in RV$ and $x_1, \dots, x_n \in BV$ then $R(x_1, \dots, x_n)$ is a Boolean formula.

1. Following McMillan [518] who illustrated several applications of a general μ -Calculus model checker including a decision procedure for language containment using Büchi automata.

1. The μ -Calculus is generally attributed to Park [578] [579] with later presentations by Pratt [604] and Kozen [441]. The notation here substantially follows McMillan's presentation [518].

2. Kozen and Tiuryn's survey [442] outlines relationships with other logics.

The relational formulae:

- if $R \in RV$ then R is a relational formula,
- if $p \in BV$ then $\lambda x_1 \dots x_n. p$ is a relational formula,
- if $R \in RV$ and F is a relational formula that is formally monotonic¹ in R then $\mu R. F$ is a relational formula standing for the least fixed point of $\lambda R. F$,
- if $R \in RV$ and F is a relational formula that is formally monotonic in R then $\nu R. F$ is a relational formula standing for the greatest fixed point of $\lambda R. F$.

The sentences are given an interpretation through a structure $I = (S, \phi, \psi)$ where:

S is a set of states,

$\phi: BV \rightarrow S$ associates an individual variable in BV to a state in S ,

$\psi: RV \rightarrow S^n$ associates every relational variable with an n -tuple of states from S .

Thus any sentence R can be associated with a set of states by $\psi(R)$. Typically the definition of I is omitted because the association between formulae and the underlying set of states S is direct. In the sequel, the formula R and the set of states that it denotes are used interchangeably.

3.3.3 The Microsemantic Model

Microsemantic analysis is based on an extremely general model which has a finite set of states and a transition relation as justified by the argument of Chapter 2. The model that of a module M which is defined as:

$$M = (Q, I, O, T, Q_0)$$

where:

$Q = \{q_1, q_2, \dots, q_n\}$ is a finite set of states,

$I = \{i_0, i_1, \dots, i_k\}$ is a finite set of inputs,

$O = \{o_0, o_1, \dots, o_l\}$ is a finite set of outputs,

$T \subseteq Q \times 2^I \times 2^O \times Q$ is a (possibly nondeterministic) transition relation,

$Q_0 \subseteq Q$ is a set of initial states, and possibly $|Q_0| > 1$.

1. The term *formally monotonic* means that R appears under an even number of negations. A function must be monotonic if it is formally monotonic but the converse is not the case.

For the most part, Q_0 plays little part in microsemantic analysis. It is included in the definition of M for completeness: everything must have a beginning.

The major interest in M is the study of Q and 2^O as domains with an internal information relation \sqsubseteq . For example, they may be flat domains, non-flat or have unobservable elements depending on the concurrent coordination properties of the microsemantics. As for 2^I , in a concurrent combination $M_1 \parallel M_2$ the input domain 2^I of M_1 is synonymous with the outputs of M_2 and vice versa. Thus it is sufficient to give domain constructions for 2^O alone while treating the i_j as mere formal placeholders for as-yet unconnected outputs.

In some cases there will be no domains 2^I or 2^O *per se*. In those cases, inter-machine communication is accomplished solely through distinguished dimensions of Q which are constructed from I and O . In such cases $T \subseteq Q \times Q$ and Q is a specially-constructed domain.

3.3.4 Symbolic Representation of Microsemantic Models

Semantics, as treated in this work, is based on a notion of the image computation which establishes the set of states that a system will be in the next or previous macrostep. Traditionally a semantics describes computation in terms of functions mapping singleton elements in a domain. This is the “point simulation” case from Figure 1-5. A semantics of singleton domain elements also has an associated semantics of powerdomain elements. This latter is called here *the image semantics*. The image semantics is significant because denotations in the image semantics can be represented in symbolic form as sentences of the μ -Calculus. In turn, functions computable in the logic necessarily imply the computation of the corresponding subsets of domain elements.¹

1. Symbolic techniques are not new: *c.f.* Darringer [222] and more recently Bose and Fisher [95], Coudert *et al.* [209], Burch *et al.* [137] and McMillan [518]. What is novel here is the focus on the primacy of the image semantics with the point semantics being the derived notion.

The following truth makes it possible to speak at once about the point semantics and the image semantics. If $f:D_1 \rightarrow D_2$ is a continuous function on a finite domain then the *extension of f* is the function $F: \wp D_1 \rightarrow \wp D_2$ over the respective powerdomains. This result applies even when f injects into a powerdomain $f:D_1 \rightarrow \wp D_2$. It is a theorem¹ that the extension of a continuous function on a finite domain is also continuous. This gives rise to the following implicit relationships which are used throughout this chapter:

$$Q = \cup Q_i = \cup \{q_i\} \quad (\text{Eq 3-6})$$

$$F\{Q\} = \cup F\{Q_i\} = \cup f\{q_i\} \quad (\text{Eq 3-7})$$

$$B\{Q\} = \cup B\{Q_i\} = \cup b\{q_i\} \quad (\text{Eq 3-8})$$

These equations state in sum that:

- A finite domain Q is merely the union of the partitions Q_i of itself. In the limit where the Q_i are singletons, the set Q is merely the union of the singleton elements, treated as sets.
- A forward image step on a set Q is the union of the forward image steps of partitions Q_i . In the limit where Q_i is a singleton, the forward image is the union of the “point simulations” of the singletons q_i .
- A backward image step is similarly defined.

Thus it is possible to describe microsemantics in terms of image computations in the realization that this view completely subsumes the “point” case. To reacquire the “point” semantics, one has but to restrict $|Q| = 1$ and either require determinacy or make non-deterministic choices arbitrarily. This view of semantics in terms of image steps is fundamental to microsemantics and to the symbolic manipulation of them.

Symbolic Representations

The attraction of symbolic methods is that the size of symbolic representations is often entirely unrelated to the number states being processed. This makes it possible to process representations of huge sets of states thereby avoiding the state explosion problem. The

1. *c.f.* Gunter and Scott [309], page 637. The proof is straightforward and is omitted. It is based on the fact that \cup is monotonic and every monotonic function on a finite domain is also continuous.

current level of interest in symbolic methods is driven by the compactness afforded by Bryant's Reduced Ordered Binary Decision Diagram (OBDD) [125]. For large classes of Boolean functions the size of its representation as an OBDD is very much smaller than its size in other representations; *e.g.* its sum-of-products form [108]. Additionally, the OBDD representation has the advantage of being canonical; that is, when two functions are equal, their representations are the same. Unfortunately OBDDs are not always compact and they are always strongly influenced by the linear ordering which must be given to the variables in any formula.¹ Further, it has been shown that the size of the OBDD for any function is bounded by a double exponential in the "reverse communication complexity" of a cross section of the circuit computing that function:²

$$|OBDD[f]| \in O\left(n \cdot 2^{w_f} \cdot 2^{w_r}\right) \quad (\text{Eq 3-9})$$

Here n is the number of inputs to the circuit, w_f is number of wires crossing a cut of the circuit in the forward direction and w_r is the number of wires crossing a cut of the circuit in the reverse direction. This bound and the problem of obtaining good variable orders in the general case has formed a serious impediment to the general application of symbolic methods.

Since Bryant's original publication, there have been a number of modifications, enhancements and reformulations of the basic data structure. Some of these are listed in Figure 3-5 and indeed the proposition of modifications to the basic OBDD to address problems of variable ordering, canonicity and exponential blow-up remains an ongoing research interest. For the purposes here, there is no attempt to anoint any particular one of

-
1. Variable ordering heuristics remain an active research topic: [270] [75] [144] [406] [273] [522] [271].
 2. This particular result is due to McMillan [518]. Other upper bounds on OBDD sizes for more specialized classes of circuits (boolean functions) are given by Bryant [126] and Devadas [233]. McMillan's reverse communication complexity result has been tightened by Aziz *et al.* [42] [43]. In summary, OBDDs may be compact when a good ordering exists but there exist many common functions for which no good ordering exists.

those representations as “the winner.” The focus here is the interface between the semantics and symbolic techniques in the general case. For that purpose, what is necessary is a rudimentary concept of the computational complexity of OBDD operations. These are listed in Figure 3-6 for the case of vanilla OBDDs, assuming using some common implementation techniques (e.g. Brace *et al.* [105] or Aziz *et al.* [37]).¹

Abbreviation	Name	References
BDD	Binary Decision Diagram	[468] [11]
OBDD, ROBDD	(Reduced) Ordered BDD	[125] [127] [88]
ADD	Algebraic Decision Diagram	[46]
BMD	Binary Moment Diagram	[128]
BDD Trees	BDD Trees	[519]
EVBDD	Edge-Valued BDD	[455]
ITE DAG	If-Then-Else DAG	[422] [423] [665]
LIF	Linearly-Inductive Functions	[313]
MDD	Multi-way BDD	[417] [667]
MTBDD	Multi-Terminal BDD	[196]
OPBDD	Ordered Partial BDD	[626] [627] [522]
RBDD	Residue BDD	[432]
TDD	A Signature-Cube encoding of a BDD	[73]
XBDD	Extended BDD	[406]
ZBDD	Zero-Suppressed BDD	[538]

Figure 3-5. Some of the Many Variants of the Binary Decision Diagram

For many of the operations listed in Figure 3-6, especially the “and-smooth” case, the indicated upper bounds are not often met in practice. Unfortunately however, the bound of Eq 3-9 is met often, especially in representations of the transition relation. Microsemantics can be seen as an attempt to avoid that bound by approaching the problem of computing forward and backward images with smaller steps using simpler transition relations. The minimal requirement therefore is a symbolic representation where Eq 3-6, Eq 3-7 and

1. These are the commonly-cited upper bounds [125] [105] [518].

Eq 3-8 hold. This statement is made more precise by classifying a symbolic technique as either an implementation issue, an algebraic optimization or a semantic model. The concern here, of course, is exclusively with the last of these three.

Name	Formula	Cost
not	$\neg f$	$O(1)$
miscellaneous boolean ops.	$f \bullet g$ for $\subseteq, \supseteq, \wedge, \vee, \oplus, =$	$O(f \times g)$
equivalence	$f \equiv g$	$O(1)$
and-smooth, and-exists	$\exists x.f(x) \wedge g(x)$	$O(f \times g \times 2^{2 x })$
substitute y for x in f	$f[y/x]$	$O(f)$

Figure 3-6. The Asymptotic Cost of OBDD-Based Computations

An *implementation issue* is a technique used to enhance and optimize the performance of a particular symbolic representation itself. Among these can be named:

- variable ordering heuristics [270] [75] [144] [406] [273] [522] [271],
- attributed edges [539] [105],
- reference-count garbage collection [105],
- breadth-first algorithms [433] [564] [27],
- global sharing of function graphs [393] [105],
- dynamic variable reordering [394] [257] [632].

As well, the variety of different representations listed in Figure 3-5 fall into the category of implementation issues. Implementation issues, while absolutely essential to the practical implementation of symbolic methods, are wholly independent of semantics.

An *algebraic optimization* is an optimization pertaining to a particular class of expressions when represented in a particular sort of symbolic representation. Among these can be named:

- auxiliary variables [148],
- function vectors [209] [95],
- functional dependencies [371],
- generalized cofactor [208] [209],
- iterative squaring [136][505],
- implicitly conjoined sets [372] [373],
- minimization [150] [166] [655],

- partial product heuristic [690],
- recursive domain and range decomposition [208],
- structural conjunctive decomposition [147] [148],
- symmetries [391] [392].

Algebraic optimizations, while quite powerful, are highly idiosyncratic to the form of the expression and the particular cost of operations on that form. They are largely unrelated to the semantics, though whether the optimization is even relevant may implicitly depend upon the micro-structure of the semantics

A *semantic method* is one which directly and explicitly relates the denoted transition relation of the model to the forward and backward image computations. Such methods necessarily apply only in the case of an image semantics, where Eq 3-6, Eq 3-7 and Eq 3-8 hold. Such an image semantics is said to be *direct* when the macrostep image computations are stipulated. It is also said to be *extensional* or *fully abstract*. A microsemantics is *computational* when approximator functionals exist and the macrostep image computations is defined in terms of their fixed points. The successive approximations generated by these functionals defines a finer granularity of time in a topologically rigorous way. When a microsemantics is not computational, then there is no such limiting approximation. This failure to have a limit may be for either of two reasons: either the semantics is fully extensional meaning that there is no approximator functional or an approximator-like functional exists but has no fixed points. In this latter case the approximating series computed by the functional has no infinite limit. An example where such is the case is the image semantics of VHDL (respectively Verilog) which is presented in Section 6.3.

A strong distinction is made here between these three classes because neither implementation issues nor algebraic optimizations contribute much to any understanding of how languages ought to be designed or how their semantics ought to relate to formal methods.

3.4 Microsemantic Analysis

Microsemantic analysis addresses two questions: how a given semantics defines the concurrent combination of one or more models $M_1 \parallel M_2 \parallel \dots \parallel M_k$, and how the domains of each M_i contributes to the definition of a macrostep. Specifically, a microsemantic analysis explains the ordering and completion conditions inside a macrostep. That explanation is aided by the domain theory, and in particular by the information ordering relation \sqsubseteq on elements within a domain and monotonicity of microsteps relative to \sqsubseteq . The functional which performs this incremental microstep computation is called the approximator functional. In turn, the \sqsubseteq relation, monotonicity and the guaranteed existence of a fixed point is used to define a rigorous notion of approximation within a step: a microstep approximation series is complete when a fixed point of the approximator functional is reached. Thus of particular interest is the internal structure of the domains Q and 2^O , and how domain elements are elevated by the successive application of the approximator functional. Various microsemantics are distinguished by their internal domain structure and the conditions at which their approximator functional has reached a fixed point.

3.4.1 Structure of the Analysis

Microsemantic analysis follows the eight steps listed in Figure 3-7. The examples of the next three sections follow this outline directly. Of course, for the fully abstract case of Section 3.4.2, steps 5, 6 and 7 are skipped because they describe how a given non-abstract microsemantics uses approximation to define the image computations.

These eight steps are justified by the following explanations:

Temporal Analysis

Time in a microsemantics necessarily has some internal structure. Time is a discrete quantity, but there is room for different granularities of time below the macro level. The

1. Temporal Analysis
2. Domain Analysis
3. The Primitive Transition Relation T_0
4. The Primitive Image Functionals F_0 and B_0
5. The Approximator Functionals \mathcal{F}_0 and \mathcal{B}_0
6. The Approximated Image Functionals \tilde{F} and \tilde{B}
7. The Projection Π to Full Abstraction
8. Observations

Figure 3-7. The Eight Steps of Microsemantic Analysis

temporal analysis describes these levels of time and the relationships between the levels. This analysis expresses how time *is intended to behave* in the microsemantics. This is an intuitive specification, deferring the more precise statement until Step 5.

Domain Analysis

To arrive at the desired temporal structure certain monotonicity relationships must hold on the elemental domains of the system. These \sqsubseteq relationships are defined at this stage. Also defined are the domain constructors that compose the elemental domains (*e.g.* individual outputs or component's state) into the domain of the whole system. Whether and how these domain constructors preserve the flatness of elemental domains or introduce non-flatness into the system's domain are the important considerations. These properties directly affect the fixed point of the approximator functionals in steps 5 and 6.

The Primitive Transition Relation T_0

The primitive transition relation is stipulated to exist. It is stipulated in the sense that this transition relation must be feasibly constructible for the microsemantics to be computational. These transitions are the "small steps" that are chained together in the approximator functional of Step 5. They form a macrostep at the approximator functional's fixed point which is defined in Step 6.

The Primitive Image Functionals F_0 and B_0

These forward and backward image computations are stipulated to exist. They are based on the primitive transition relation T_0 and, like it, are stipulated in the sense that they must be feasible for microsemantics to be computational.

The Approximator Functionals \mathcal{F}_0 and \mathcal{B}_0

An approximator functional is a conceptual control scheme which is used to construct successively better approximations of the respective fully abstract image functional F or B . They amount to control schemes or schedules for the primitive image functionals F_0 or B_0 about which precise statements about completion and exactness in the limit can be made.

The Approximated Image Functionals \bar{F} and \bar{B}

In practice, the approximator functionals are never constructed in their own right. Instead, they are evaluated until they reach a fixed point: $\bar{F} = \mathcal{F}_0\{\bar{F}\}$ or $\bar{B} = \mathcal{B}_0\{\bar{B}\}$. At that point \mathcal{F}_0 or \mathcal{B}_0 is said to have computed the best approximation, \bar{F} or \bar{B} , of the respective fully abstract image functional F or B . The approximated image functionals are the least and greatest fixed points of their respective approximator functionals: $\bar{F} = \mu\mathcal{F}_0$ and $\bar{B} = \nu\mathcal{B}_0$.

The Projection Π to Full Abstraction

An approximated image functional \bar{F} and \bar{B} is said to be a “best” approximation of F or B in the sense that there is no better one: further refinements using \mathcal{F}_0 or \mathcal{B}_0 merely produce the same results. The approximated image functional is said to be exact when there exists a projection $\Pi: M_0 \rightarrow M$ which makes $F = \bar{F} \circ \Pi$ and $B = \bar{B} \circ \Pi$.

Observations

From the seven previous steps, some observations can be drawn about the microseman-

tics:

The first observation is that different domains deserve different of sorts of information content ordering \sqsubseteq over \circ -time. Outputs ought to be in a non-flat domain that orders outputs by “is defined.” An output which has been defined is larger than one which is as-yet undefined. Further, the output domain constructor should preserve this non-flatness. States on the other hand ought to be in flat domains where no state is better than any other. This is especially important for \circ -states. These observations are made more precise in the particular examples.

The second observation is that there is a trade-off between the coordination that is “compiled away” within a fully abstract unit versus the complexity which is left latent in the iteration of the approximator functional. This plays out in the fully abstract transition relation being too complex to be feasibly constructed, yet its associated state space is clean and devoid of implementation-dependent “junk” like unobservable \circ -states. In contrast, a non-abstract transition relation can be very regular and compact enough to be constructed trivially. Its associated state space however is filled with implementation details like unobservable \circ -states and the remnants of the outputs used for intra-step coordination. This can be stated in a mathematically precise way.

3.4.2 The Fully Abstract Semantics

A semantics is fully abstract when the denotations of two language elements are equal whenever those constructs behave the same in all contexts and vice versa.¹ Intuitively full abstraction is a statement that the abstract semantics exactly reflects observable behavior and no more. The fully abstract semantics has no implementation details of any sort. This can be precisely stated using the information content ordering \sqsubseteq for an abstract seman-

1. This definition follows Mulmuley [550]. The requirements and properties of a fully abstract model for the typed λ -calculus (PCF) are described in Milner [530] and Mulmuley [550].

tics S_A and a non-abstract semantics S_O (an operational semantics). Let C be an arbitrary “context functional.” Then full abstraction is the condition that:

$$\forall s, t, C \in L. S_A \llbracket s \rrbracket \subseteq S_A \llbracket t \rrbracket \Leftrightarrow S_O \llbracket C \{s\} \rrbracket \subseteq S_O \llbracket C \{t\} \rrbracket$$

Exploiting the fact that $(x \subseteq y) \wedge (y \subseteq x) \Rightarrow (x = y)$ one arrives at the statement that full abstraction is the condition that two language elements denote the same element (the abstract side) when they behave the same in all contexts (the operational side).

Full abstraction therefore implicitly rests the definition of a “behavior” and a “denotation” since the denotations must be the same when the behaviors are the same. The argument presented in Chapter 2 was that a semantics is fundamentally about mapping language elements to transition relations, except in some very exotic circumstances. Earlier in this section the forward and backward image computations were presented thereby giving the static transition relation a directional orientation in time. Thus the denotations of language elements are transition relations. As for behaviors, the behavioral domain constructed in Section 3.2 showed that it is not possible to use an approximation-based definition scheme to define infinite properties based on fairness because fairness is discontinuous in the infinite limit. The lesson there was that a reasonable approach is to use an approximation-based definition for the behavior within a step and allow for some other (incomputable) method for specifying the trans macrostep behaviors. Thus a behavior, in the sense used here, is the extensional characterization of a macrostep which is just a transition relation. Hence, for the purposes of full abstraction in this context, both a “behavior” and a “denotation” are a transition relation.

In fact, full abstraction is a precise way of stating when a semantics is a canonical representation for the behavior of its corresponding program element. Two transition relations operate the same way in a macrostep just when they are equal. To use the definitions given in Section 3.4.1 a fully abstract semantics is not computational, it is extensional and must

be stipulated. In particular a fully abstract transition semantics uses the transition relation in monolithic form and offers no decomposition of it.

3.4.2.1 Temporal Analysis

In this analysis, time is discrete and has a single level. No behavior can be observed between time steps and there is no internal structure to a step. A step is atomic and is said to occur “instantaneously” with respect to the model of time. An external observer may be able to observe an irregular pattern to the steps, but within the model the instants are isomorphic to the natural numbers. A fully abstract time line is illustrated in Figure 3-8. It is distinguished by its simplicity.

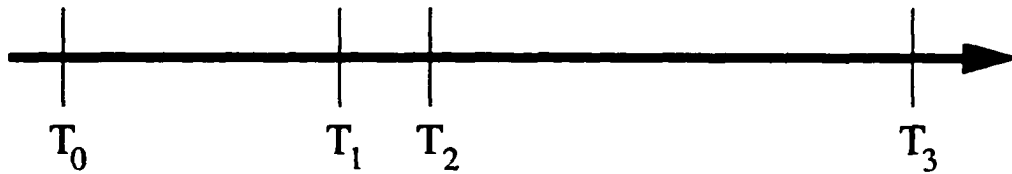


Figure 3-8. The Single Level of a Fully Abstract Time

3.4.2.2 Domain Analysis

Two flat domains were illustrated in Figure 3-3. A flat domain conveys the minimal amount of information about its elements in the sense that none of the proper elements contain any more information than any other. A flat domain D has the property that:

$$\forall d_\alpha, d_\beta \in D - \{\perp\} . d_\alpha \sqsubseteq d_\beta \Leftrightarrow d_\beta \sqsubseteq d_\alpha \Leftrightarrow d_\alpha \equiv d_\beta \quad (\text{Eq 3-10})$$

For a fully abstract transition semantics, the domains must convey as little information as possible. Therefore:

- S is a flat domain of states,
- 2^O is a flat domain of outputs.

The first statement is justified by the fact that no state of M is any different than any other state. The second statement is justified by the fact that inter-component coordination is

essentially “compiled away” by the concurrent combination operation. This is shown in the following, however intuitively this must be so because a fully abstract semantics hides every feature which is not extensional.

$$S = \{s_1, s_2, \dots, s_n\} \text{ is a flat domain} \quad (\text{Eq 3-11})$$

$$2^O = \bigotimes_{i=0}^l O_i \quad (\text{Eq 3-12})$$

$$Q = S, \text{ and is independent of } 2^O \quad (\text{Eq 3-13})$$

3.4.2.3 The Transition Relation $T(c, n)$

The fully abstract transition relation is a monolithic quantity defined in terms of the conjunction of the transitions allowed by each machine:

$$T(c, n) = \exists o. T_{P_1}(c_{P_1}, o_{P_1}, o_{P_1}, n_{P_1}) \wedge T_{P_2}(c_{P_2}, o_{P_2}, o_{P_2}, n_{P_2}) \wedge \dots \wedge T_{P_k}(c_{P_k}, o_{P_k}, o_{P_k}, n_{P_k})$$

Each component transition relation can be thought of conceptually as consisting of two parts, a “pure” transition relation which determines the successor state(s) and the output relation which determines the outputs produced by the component in a step:

$$T_P(c_P, i_P, o_P, n_P) = X_P(c_P, i_P, n_P) \wedge O_P(c_P, i_P, o_P) \quad (\text{Eq 3-14})$$

The general inter-process communication structure of a fully abstract semantics is depicted in Figure 3-9. There the outputs of any machine are available to all and every machine makes its decisions based on its internal state and the global output. The emission of the outputs and the occurrence of the individual components’ transitions occurs atomically:

$$T(c, n) = \exists o. \prod_{i=1}^k T_{P_i}(c_{P_i}, o_{P_i}, o_{P_i}, n_{P_i}) \quad (\text{Eq 3-15})$$

A fully abstract semantics is not computational because there is no way to approximate the transition relation of Eq 3-15.

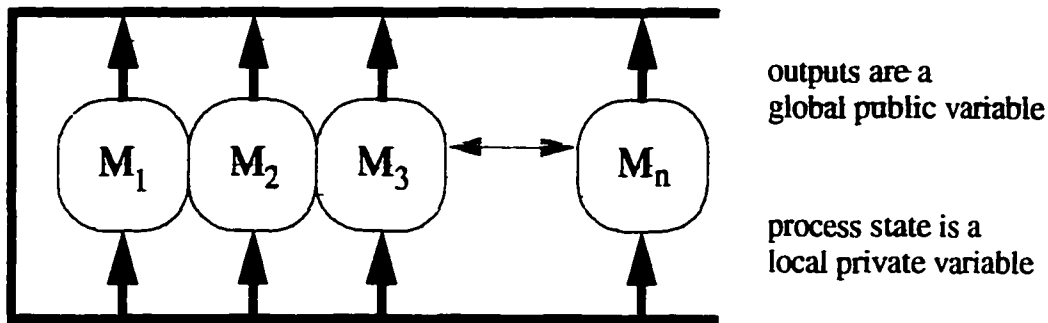


Figure 3-9. Communication in a Fully Abstract Semantics

3.4.2.4 The Image Functionals $F\{Q\}$ and $B\{Q\}$

With the monolithic transition relation being given, the image functionals are:

$$F = \lambda Q. (\exists c. Q(c) \wedge T(c, n)) [n/c] \quad (\text{Eq 3-16})$$

$$B = \lambda Q. (\exists n. T(c, n) \wedge Q(n)) [c/n] \quad (\text{Eq 3-17})$$

In application form, the two functionals compute the step image directly:

$$Q_{i+1} = F\{Q_i\} \quad (\text{Eq 3-18})$$

$$Q_{i-1} = B\{Q_i\} \quad (\text{Eq 3-19})$$

3.4.2.5 Observations

The key observation to make on the fully abstract semantics is that the smoothing operation (the $\exists o$ in Eq 3-15) has the effect of “compiling away” the internal coordination within the system. What is left is a purely extensional representation of the behavior in a step. In a practical implementation the construction of the monolithic transition relation is a major bottleneck.¹ This gives impetus to the design of classes of computational semantics which are necessarily non-abstract.

1. This bottleneck is a direct effect of the state explosion problem. Various algebraic approaches have been used to address this problem [209] [136] [690].

3.4.3 A Non-Abstract δ -Time Semantics

If the problem is too much abstraction in a step and there is no internal structure within a step then a semantics-based approach to the problem is to introduce internal structure into a step. This gives rise to a notion of δ -time which is similar in spirit to the kind found in an event-driven simulator.¹ The definition of δ -time semantics here is decidedly denotational and additionally is a (non-abstract) computational semantics. The communication model is a generalization of that shown in Figure 3-9. The generalization is that each machine takes multiple steps (δ -steps) in a macrostep. This introduces a restricted notion of interleaving wherein the execution of the machines are interleaved subject to the inputs which they require becoming defined. The implicit restriction on the interleavings is that no machine executes across a δ -step which depends on an input which has not yet arrived.²

3.4.3.1 Temporal Analysis

In this model, time has a two-level structure as depicted in Figure 3-8. Within each macrostep there are an arbitrary number of δ -steps. The number of δ -steps can vary based on the internal state of M or on the input given to M in a step or some combination of the two. A particular chain of δ -steps is said to form a δ -path. Any δ -path may split, converge and reconverge with another δ -path in the δ -time level. The only restriction is that there be a bounded number of δ -steps in any individual δ -path. The spaghetti-like nature of δ -time is illustrated in Figure 3-11 in the context of the image computations.

1. The parallel is not exact. In fact, the δ -time of a discrete-event simulator is markedly different in significant ways. An analysis of discrete-event semantics is deferred until Section 6.2 since that analysis is best undertaken in the framework of the RMC Barrier Theorem presented in Chapter 4. Particular comments on the *three-level* model of time used in a discrete-event semantics are found in Section 6.2.3.

2. It is decidedly not clear that such an interleaving necessarily exists. The conditions on when it does exist is substantially explained by the RMC Barrier Theorem. That theorem is presented in Chapter 4 so for the moment the reader is asked to continue in the belief that such an interleaving can exist for the classes of system having this microsemantics.

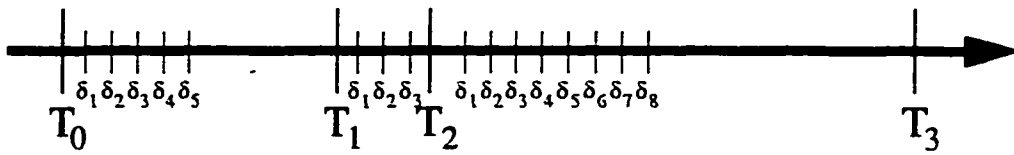


Figure 3-10. The Two Levels of a Non-Abstract δ -Time

The significant aspect of the finer-granularity δ -time is that it is entirely unobservable outside of M . Again the philosophical position is that the only “real” measure of time is macrotime. On the macrotime line events either take place *instantly*, meaning within a macrostep, or they take *some* time, meaning they take one or more macrosteps. Thus the only externally observable effect of δ -time, is to induce an *causal ordering* relationship among events that occur instantly. The causal ordering is observable at the module interface between certain inputs consumed and outputs produced by the model. In particular this means that δ -time is immeasurable (in units of macrosteps) and each δ_i is said to take “zero time” (in units of macrosteps).

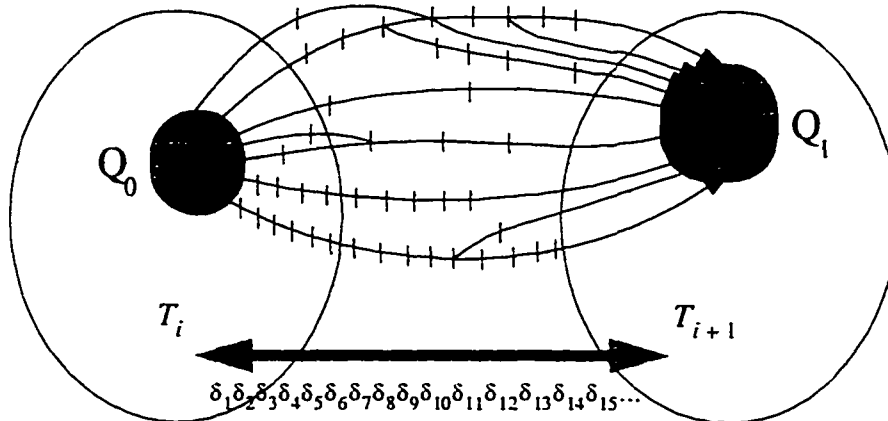


Figure 3-11. The Number of δ -Steps is State- and Input-Dependent

3.4.3.2 Domain Definitions

The two salient characteristics of the δ -time semantics are the unobservability of the “ δ -states” and the well-definedness of a δ -path’s completion. The domains of M are the

explicit support for these two characteristics. Therefore:

- S is a non-flat domain of states with a particular terrace-like structure,
- 2^O is a non-flat domain of outputs with a Cartesian structure.

The internal structure of the δ -time domains depend on the particular relationships and interpretations of these two domain structures. Therefore some explanation of them is required.

The Domain S

The domain S has the interesting property that it must contain both macrostep states and δ -step states. Therefore, it must be a special kind of sum domain between macrostep states S_T and δ -step states S_δ :

$$S_T = \{s_{T_1}, s_{T_2}, \dots, s_{T_{n_T}}\} \quad (\text{Eq 3-20})$$

$$S_\delta = \{s_{\delta_1}, s_{\delta_2}, \dots, s_{\delta_{n_\delta}}\} \quad (\text{Eq 3-21})$$

Intuitively no macrostep state $s_T \in S_T$ is better than any other and only one macrostate is chosen per macrostep. Thus S_T is a flat domain. On the other hand, there are many δ -step states selected per macrostep so there must be a non-trivial order within S_δ so that the state selected in δ_i can be less than the state selected at δ_{i+1} . It is not necessary to require that any particular order hold within S_δ other than that \subseteq_{S_δ} define a (complete) partial order.¹

In either the forward or backward direction a δ -path must terminate in a macrostep state s_T . In the forward direction this means the “end” of the δ -path and in the backward direction it means the “start” of the δ -path. Thus the following relationships must hold:

1. The practical aspects of constructing S_δ so that a partial order is guaranteed to exist are covered in Chapter 7 in the setting of an imperative language. In a data-flow setting a partial order is guaranteed to exist when there is a topological order on the network elements executed within a macrostep: this statement is usually phrased as “combinational cycles must be broken by latches.”

$$\text{forward: } \forall s_\delta \in S_\delta, s_T \in S_T \cdot s_\delta \sqsubseteq s_T \quad (\text{Eq 3-22f})$$

$$\text{backward: } \forall s_\delta \in S_\delta, s_T \in S_T \cdot s_T \sqsubseteq s_\delta \quad (\text{Eq 3-22b})$$

Both relationships cannot hold at the same time. When constructing the domain equations for the forward direction it is sufficient for Eq 3-22f to hold. When constructing the domain equations for the backward direction it is sufficient for Eq 3-22b to hold.

To formalize this relationship a new sort of domain constructor is introduced:

$D = D_1 \tilde{\oplus} D_2 \tilde{\oplus} \dots \tilde{\oplus} D_n$ constructs a “terraced coalesced” sum domain.

It is exactly like the coalesced sum with the addition that every $y \in D_i$ is more valuable than any $x \in D_{i+1}$. The terrace structure is defined by the new relationships $\forall y \in D_i, \forall x \in D_{i+1} \cdot x \sqsubseteq_D y$.

The domain S is defined relative to Eq 3-22f and Eq 3-22 by:

$$\text{forward: } S = S_T \tilde{\oplus} S_\delta \quad (\text{Eq 3-23f})$$

$$\text{backward: } S = S_\delta \tilde{\oplus} S_T \quad (\text{Eq 3-23b})$$

The ordering \sqsubseteq_S implicitly refers to either Eq 3-22f or Eq 3-22b. When there is ambiguity, the following notation is used in the sequel to refer to the direction-specific \sqsubseteq_S :

- \sqsubseteq_f refers to \sqsubseteq_S via Eq 3-22f,
- \sqsubseteq_b refers to \sqsubseteq_S via Eq 3-22b.

The Domain 2^O

The output domain for M must represent the outputs which have been defined so far in the δ -path. For a single output $o_i \in O$, the output may be *undefined*, *present* or *absent* from a domain looking like the one shown in Figure 3-12. This domain is of course isomorphic to the boolean domain shown in Figure 3-3, however it is useful to think of output values as either having been assigned or unassigned in a step.

When considering two or more outputs, the relative defined-ness of the individual components must be taken into consideration. This is accomplished with the Cartesian product domain constructor. Its effect is illustrated in Figure 3-13 for the simple case of a two-out-

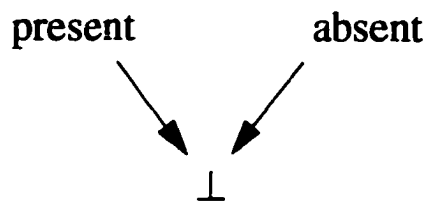
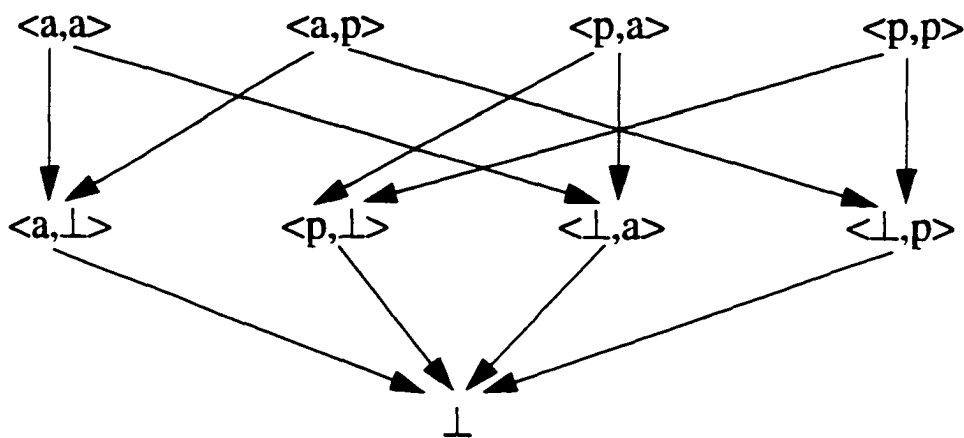


Figure 3-12. The Primitive Domain of a Single Output Variable
 put system. The output domain 2^O is just the Cartesian product of the domains of the indi-
 vidual output variables:

$$2^O = \prod_{i=1}^l O_i \quad (\text{Eq 3-24})$$



$\perp = \text{undefined}$, $a = \text{absent}$, $p = \text{present}$

Figure 3-13. The Constructed Domain of an Output Variable Pair

The Domain Q

Together the states and the outputs are non-flat:

$$Q = S \times 2^O \quad (\text{Eq 3-25})$$

What is interesting about the domain of Q is how it is designed to behave, relative to \sqsubseteq , across δ -steps of an image computation. In particular the domain Q has the following interesting properties:

- The pre-existing partial order on δ -states means that any path across the δ -state space is non-decreasing;
- Paths across the δ -state space can be arbitrarily long but may not have loops in S ;
- Any macrostate is greater than every δ -state, so when a δ -path reaches a macrostate, it is maximal and cannot be extended;
- Defining an output increases, and this increase is maximal when all outputs are defined.

Taken together these properties on the domain Q give δ -paths a monotonic trajectory under forward and backward image computations. This is depicted in Figure 3-14.

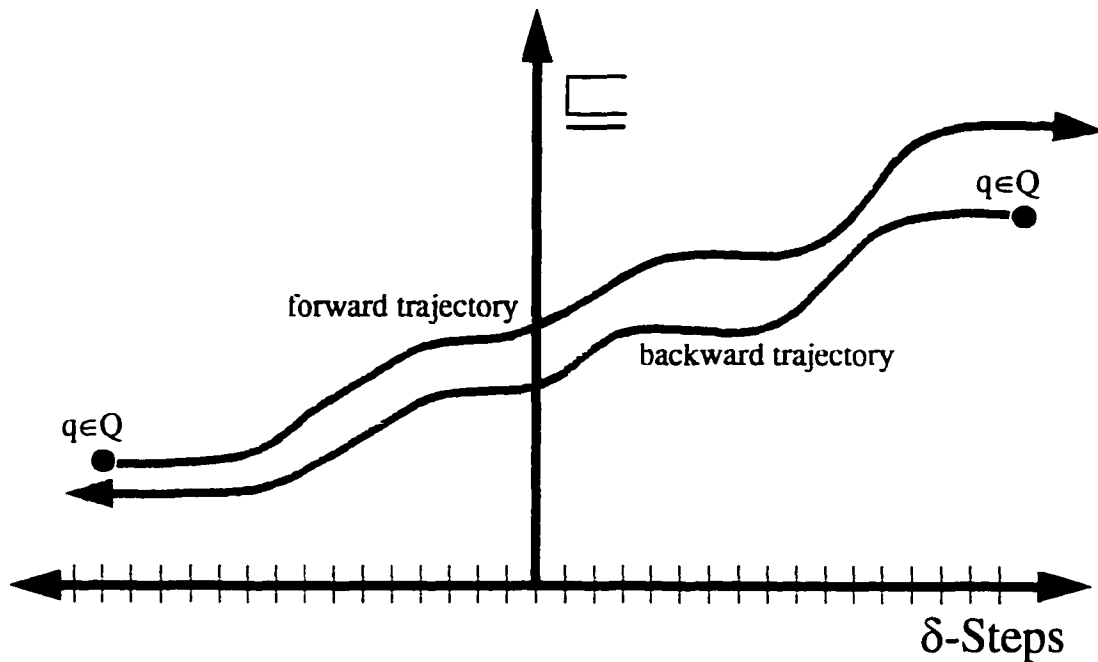


Figure 3-14. The Forward and Backward Trajectories in the Domain M

3.4.3.3 The Primitive Transition Relation $T_\delta(c, n)$

Each δ -step is a transition and so there must be a transition relation which describes it. As a shorthand these transition relations can be called *instructions*. In particular, the mon-

iker is apt because in a δ -step there are a set of variables that change and a set of variables which stay the same. This is exactly what occurs inside an instruction set processor when it executes an instruction. An instruction's transition relation therefore consists of two parts, the *op* part and the *preserve* part:

$$T_I^{(op)}(c_p, n_I) = \text{a relation over } c_I \text{ and } n_I \text{ that is specific to } I \quad (\text{Eq 3-26})$$

The *preserve* part merely relates the universe U of other variables that don't change:

$$T_I^{(preserve)}(c_U, n_U) = \prod_{i=1}^I (c_{U_i} \equiv n_{U_i}) \quad (\text{Eq 3-27})$$

The whole instruction is the conjunction of what changes and what remains the same.¹

$$T_I(c, n) = T_{I_i}^{(op)}(c_{I_i}, n_{I_i}) \wedge T_{I_i}^{(preserve)}(c_{\bar{I}_i}, n_{\bar{I}_i}) \quad (\text{Eq 3-28})$$

The appeal of such a representation is that it may be possible to represent $T_I^{(preserve)}$ differently than $T_I^{(op)}$. In the familiar (non-symbolic) "point" execution world $T_I^{(preserve)}$ is vacuously represented since memory naturally holds its value unless rewritten.

Very few restrictions need be placed on the instruction transition relation of Eq 3-28. There is one extremely important requirement however which will be crucial to the definition of the approximator functionals' fixed point in Section 3.4.3.6. Every δ -path is required to have an *absorbing end condition* so that all the δ -paths of various lengths can be stretched out and aligned. The absorbing end condition is a self-loop transition on a macrostate that is specific to a particular direction. The idea is that when the directional δ -step series reaches the final macrostate of the δ -path, it gets "stuck." This has the obvious effect of making every δ -path an infinite path, which in turn makes all the δ -paths the same (infinite) length and thereby elegantly sets the stage for δ -time to be imbedded

1. This constraint is variously referred to as *the stability property* [370] or as *the frame problem* [162]. The property is quite simple: the states which don't change remain the same. The reason why this is a problem is that the exposition of the lack of change must be made explicitly for each I . This representation can become quite large both in absolute terms and in relation to that of I .

within macrotime in a well-defined way.¹ The absorbing end conditions for the forward and backward case are depicted in Figure 3-15. In practice the absorbing end conditions obligation is discharged by defining the transition relation of a “halt” instruction as a self-loop and in turn requiring that every δ -path end in such a halt instruction.

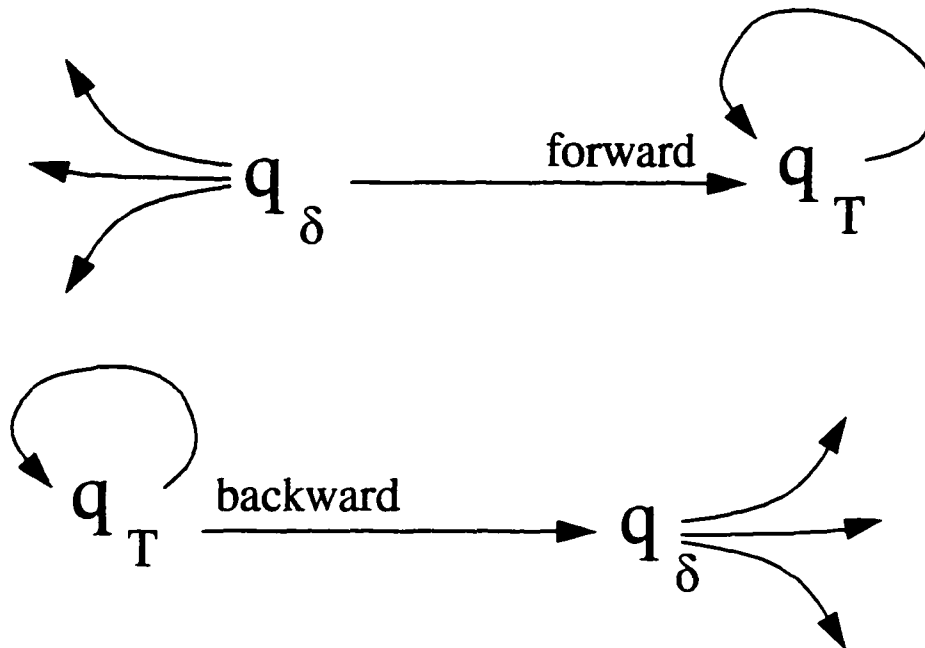


Figure 3-15. Absorbing End Conditions for Forward and Backward δ -Paths

The transition relation of M , as a whole, is the disjunction of all the instructions that it can execute:

$$T_{\delta}(c, n) = \sum_{i=1}^k T_{I_i}(c, n) \quad (\text{Eq 3-29})$$

The first point to observe about Eq 3-29 is that disjunction distributes over smoothing (the $\exists c$ and $\exists n$ steps of the fully abstract image computations). This means that in a practical

1. The reader should understand at this point that the set of δ -step paths, both finite and infinite is a domain. The set of δ -step paths of finite length are the compact elements of the domain and the δ -step paths of infinite extent are the limit points. As defined for this special case, the limit series of approximations defined by δ -step paths of finite length is continuous in the infinite limit.

implementation, the disjunction need never be explicitly computed! The T_{I_i} can be kept disaggregated and the disjunction can be taken after the image computation as follows:

$$\begin{aligned} F\left\{\sum_i Q_i\right\} &= \left(\exists c.\left(\sum_i Q_i(c)\right) \wedge T(c, n)\right) [n/c] \\ &= \sum_i (\exists c.Q_i(c) \wedge T(c, n)) [n/c] \end{aligned}$$

The second observation to make about Eq 3-29 is that every term $T_{I_i}(c, n)$ depends on all the variables c and n since the frame axioms (the information that stays the same) must be explicitly represented and processed at each step.

3.4.3.4 The Primitive δ -Image Functionals $F_\delta\{Q\}$ and $B_\delta\{Q\}$

The primitive δ -image are extensional and are exact replicas of the fully abstract image functionals:

$$F_\delta = \lambda Q. (\exists c.Q(c) \wedge T_\delta(c, n)) [n/c] \quad (\text{Eq 3-30})$$

$$B_\delta = \lambda Q. (\exists n.T_\delta(c, n) \wedge Q(n)) [c/n] \quad (\text{Eq 3-31})$$

And in application form the forward and backward image computations are:

$$Q_{\delta_{i+1}} = F_\delta\{Q_{\delta_i}\} \quad (\text{Eq 3-32})$$

$$Q_{\delta_{i-1}} = B_\delta\{Q_{\delta_i}\} \quad (\text{Eq 3-33})$$

By convention, forward δ -paths are numbered increasing from 0 while backward δ -paths are numbered decreasing from 0. This allows for the association between the old macrostate Q_i and the “zeroth” δ -step, $Q_i = Q_{\delta_0}$, to be unambiguous for the forward and backward directions. The new macrostate is Q_{i+1} in the forward direction and Q_{i-1} in the backward direction.

The following is crucial to the development, though these facts will not be used again

until the observations of Section 3.4.3.6. By construction, T_δ has the following properties:

- absorbing end conditions for all $s \in S_T$,
- fidelity to the partial order among states $s \in S_\delta$,
- single assignment of output variables $(o_1, o_2, \dots, o_l) \in 2^O$ across δ -paths.

Thus, the following necessarily holds:¹

$$\forall i \geq 0. Q_{\delta_i} \sqsubseteq F_\delta \{Q_{\delta_i}\} \quad (\text{Eq 3-34})$$

$$\forall i \leq 0. B_\delta \{Q_{\delta_i}\} \sqsubseteq Q_{\delta_i} \quad (\text{Eq 3-35})$$

That is, any path through T_δ is increasing in the forward direction and decreasing in the backward direction.

3.4.3.5 The Approximator Functionals $\mathcal{F}_\delta\{\tilde{F}\}$ and $\mathcal{B}_\delta\{\tilde{B}\}$

The primitive δ -image functionals are used in conjunction with their respective approximator functionals, \mathcal{F}_δ or \mathcal{B}_δ , to define the forward or backward macrostep image computation. The intent is that an approximated macrostep image functional \tilde{F} or \tilde{B} computes the infinite limit of δ -step chains. Finite approximations to this infinite limit, of course, consist of arbitrary-length δ -step chains and the approximator functionals increase the approximation chain by one step. The approximator functionals are:

$$\mathcal{F}_\delta = \lambda F. \lambda Q. F \{Q\} \quad (\text{Eq 3-36})$$

$$\mathcal{B}_\delta = \lambda B. \lambda Q. B \{Q\} \quad (\text{Eq 3-37})$$

By design, a macrostep is the least fixed point of \mathcal{F}_δ , respectively the greatest fixed point of \mathcal{B}_δ , relative a crude initial estimate. When the approximator functionals, \mathcal{F}_δ and \mathcal{B}_δ , are defined to be monotonic and continuous then successive approximations to \tilde{F} and \tilde{B} , can be incrementally computed by applying the relevant approximator functional. Thus in application form:

1. Note that Eq 3-34 is implicitly defined in terms of the \sqsubseteq_f of Eq 3-23f while Eq 3-35 is implicitly defined in terms of the \sqsubseteq_b of Eq 3-23b. These references are unambiguous because they are directly tied to the direction of the computation. In the sequel, no further mention is made of this latent reference.

$$\tilde{F}_{i+1} = \mathcal{F}_\delta\{\tilde{F}_i\} \quad (\text{Eq 3-38})$$

$$\tilde{B}_{i-1} = \mathcal{B}_\delta\{\tilde{B}_i\} \quad (\text{Eq 3-39})$$

The definitions and constructions of the previous sections have set up conditions so that Eq 3-36 and Eq 3-37 are monotonic and continuous in the infinite limit. This ensures that the infinite limit series generated by Eq 3-38 and Eq 3-39 will have finite convergence. The demonstration that monotonicity and continuity hold for Eq 3-36 and Eq 3-37 is not obvious and requires some proof. The following remains to be proved:

$$\mathcal{F}_\delta \text{ is monotonic:} \quad \tilde{F}_i \sqsubseteq \tilde{F}_{i+1}$$

$$\mathcal{B}_\delta \text{ is monotonic:} \quad \tilde{B}_{i-1} \sqsubseteq \tilde{B}_i$$

$$\mathcal{F}_\delta \text{ is continuous:} \quad \tilde{F} = \bigsqcup_{i=0}^{\infty} \tilde{F}_i$$

$$\mathcal{B}_\delta \text{ is continuous:} \quad \tilde{B} = \bigsqcup_{i=0}^{\infty} \tilde{B}_i$$

Indeed, that these properties hold for Eq 3-36 and Eq 3-37 is contingent upon the starting point of the series and the definition of \sqsubseteq .

3.4.3.6 The Approximated Image Functionals $\tilde{F}\{Q\}$ and $\tilde{B}\{Q\}$

A macrostep is the fixed point of \mathcal{F}_δ and \mathcal{B}_δ relative to some initial crude estimate,¹ and the crudest possible estimate is the unit δ -step, F_δ and B_δ , from Eq 3-30 and Eq 3-31 respectively. The semantics is computational, not existential so \tilde{F} and \tilde{B} cannot be computed directly. Rather must be incrementally approximated from the initial esti-

1. The use here of a "crude estimate of \tilde{F} " neatly side-steps the problem of understanding what \perp is for the functional domain $D = Q \rightarrow Q$. From Eq 3-1, the computation of the *least* fixed point $\mu \mathcal{F}_\delta$ requires the evaluation of a series commencing with the term $\mathcal{F}_\delta\{\perp_D\}$. It is clear that \perp_D is the "undefined function" and expanding $\mathcal{F}_\delta\{\perp_D\}$ gives $\lambda Q. \perp_D\{Q\}$, but that leaves the question of the value of $\perp_D\{Q\}$, for arbitrary Q . Is this to be \perp_Q , the "undefined state?" This is entirely unenlightening.

Fortunately this conundrum need not be pondered since the least *relative* fixed point relative to a "crude estimate" is sufficient for the purposes here. The relative least and greatest fixed points are guaranteed to exist by the construction in Section 3.4.3.3.

mate using Eq 3-38 and Eq 3-39 as:

$$\begin{aligned}
 \bar{F}_0 &= \mathcal{F}_\delta\{F_\delta\} &= \lambda Q.F_\delta\{Q\} & \bar{B}_0 &= \mathcal{B}_\delta\{B_\delta\} &= \lambda Q.B_\delta\{Q\} \\
 \bar{F}_1 &= \mathcal{F}_\delta\{\bar{F}_0\} &= \lambda Q.F_\delta\{F_\delta\{Q\}\} & \bar{B}_{-1} &= \mathcal{B}_\delta\{\bar{B}_0\} &= \lambda Q.B_\delta\{B_\delta\{Q\}\} \\
 \bar{F}_2 &= \mathcal{F}_\delta\{\bar{F}_1\} &= \lambda Q.F_\delta^{(3)}\{Q\} & \bar{B}_{-2} &= \mathcal{B}_\delta\{\bar{B}_{-1}\} &= \lambda Q.B_\delta^{(3)}\{Q\} \\
 \dots & & \dots & \dots & & \dots \\
 \bar{F}_n &= \mathcal{F}_\delta\{\bar{F}_{n-1}\} &= \lambda Q.F_\delta^{(n+1)}\{Q\} & \bar{B}_n &= \mathcal{B}_\delta\{\bar{B}_{n+1}\} &= \lambda Q.B_\delta^{(n+1)}\{Q\}
 \end{aligned}$$

The fixed point of these series are reached just when $\bar{F}_n = \bar{F}_{n-1}$ and $\bar{B}_n = \bar{B}_{n+1}$. Following Eq 3-3 and Eq 3-4, the upper bound of the infinite series defines a fixed point:

$$\bar{F} = \mu_{F_\delta} \mathcal{F}_\delta = \bigsqcup_{i=0}^{\infty} \mathcal{F}_i\{F_\delta\} \quad (\text{Eq 3-40})$$

$$\bar{B} = \nu_{B_\delta} \mathcal{B}_\delta = \bigsqcup_{i=0}^{\infty} \mathcal{B}_i\{B_\delta\} \quad (\text{Eq 3-41})$$

The starting points of these limit series is well-defined. These definitions depend on \sqsubseteq_f and \sqsubseteq_b as were defined in Eq 3-34 and Eq 3-35 respectively.

The Monotonicity of \mathcal{F}_δ and \mathcal{B}_δ

\mathcal{F}_δ and \mathcal{B}_δ are monotonic because F_δ and B_δ are (constructed to be) monotonic; as stated in Eq 3-34 and Eq 3-35.

The Continuity of \mathcal{F}_δ and \mathcal{B}_δ

The continuity of \mathcal{F}_δ and \mathcal{B}_δ follows from the continuity F_δ and B_δ . In turn, the continuity of F_δ and B_δ follows from the constructed properties of T_δ . The interesting behavior causing this is that Eq 3-32 and Eq 3-33 sensitize the “halt” transition at the end of every δ -path. Clearly there is some $k > 0$ where:

$$\forall i \geq k. Q_{\delta_{i+1}} = Q_{\delta_i} \quad (\text{Eq 3-42})$$

$$\forall i \geq k. Q_{\delta_{i+1}} = F_\delta\{Q_{\delta_i}\} \quad (\text{Eq 3-43})$$

Respectively there is some $k < 0$ where:

$$\forall i \leq k. Q_{\delta_{i-1}} = Q_{\delta_i} \quad (\text{Eq 3-44})$$

$$\forall i \leq k. Q_{\delta_{i-1}} = B_{\delta} \{ Q_{\delta_i} \} \quad (\text{Eq 3-45})$$

By construction, both of these cases are the result of the absorbing end conditions which were sensitized when all the elements in the respective Q_{δ_i} are at some final state $(s_T, o) \in Q$. The s_T sensitizes the halt instructions in T_{δ} which gets the element “stuck.” These are the limit points, each of which is computable as the upper bound of an infinite number of finite approximations. The finite approximations are constructed incrementally by \mathcal{F}_{δ} and \mathcal{B}_{δ} . Their nearest fixed point relative to the unit step is the limit point \bar{F} and \bar{B} respectively.

3.4.3.7 The Projection Π to Full Abstraction

The projection of model elements back onto the fully abstract case is straightforward. The projection operator Π need only delete out the superfluous domain components: the gratuitous output dimension and the unobservable δ -states. The projection operator is:

$$\Pi = \lambda Q. \left(\pi_{Q \rightarrow s} \circ \pi_{s \rightarrow s_T} \right) \{ Q \} \quad (\text{Eq 3-46})$$

The forward and backward image computations of the fully abstract case can be recovered directly by composing this projection operator with the computationally approximated macrostep image computations of Eq 3-40 and Eq 3-41 respectively:

$$F = \bar{F} \circ \Pi \quad (\text{Eq 3-47})$$

$$B = \bar{B} \circ \Pi \quad (\text{Eq 3-48})$$

3.4.3.8 Observations

There are three observations to be made about the non-abstract semantics of δ -time.

Serializability of Concurrent Coordination

First and foremost, this non-abstract semantics is a serialization scheme for concurrent

coordination. It uses a constrained interleaving model that exploits the monotonicity of the domain Q . The domain Q has two dimensions, state and output and it is (of course) monotonic in both.

The ordering of δ -steps is induced by the domain ordering within S_δ . In practice such a δ -state partial order might come from the loop-free control flow paths in an imperative language. An example of a language with these properties is presented in Chapter 7.

The second aspect is the restriction to single assignment of outputs across δ -time. This property is a requirement imposed by the internal structure of the output domain 2^O . In turn, it is exploited in Eq 3-34 and Eq 3-35 which declare F_δ and B_δ to be monotonic. It is also implicitly exploited again in the definition of Π . That projection operator is so simple because all the “real” information is in the state component and in particular in the macrostate. It is sufficient for Π to suppress the output component because outputs are only implicitly used to synchronize within a step. As is shown in Chapter 4, any semantics that allows multiple assignments to outputs in (its version of) δ -time will not have a well-defined projection to full abstraction.

Outputs in the State Space

This semantics stores output values in an orthogonal component of the state space Q . This is explicitly not the case in the fully-abstract semantics of Section 3.4.2. In the fully abstract case all the outputs and their coordinating effects are “compiled away,” leaving the monolithic transition relation. Here the macrostep transition relation is kept in disaggregated form as the δ -step transition relation T_δ . The output components are stored in the state element across δ -time. The projection Π erases them and makes them unobservable in macrotime. In contrast with the fully-abstract case, outputs and their coordinating effects are not “compiled away,” they are explicit and unevaluated by the semantics.

The Illusory Nature of \tilde{F} and \tilde{B}

The presentation of Section 3.4.3.5 worked from the premise that \tilde{F} and \tilde{B} would be iteratively approximated by successive application of \mathcal{F}_δ and \mathcal{B}_δ until no change resulted: when the test $\tilde{F}_n = \tilde{F}_{n-1}$ and $\tilde{B}_n = \tilde{B}_{n+1}$ returned “true.” That presentation, if implemented as stated, would require a representation of functions where equality could be tested. Defining such a representation for higher-order functions could be problematic.

In practice only a weaker form of \tilde{F} and \tilde{B} are needed. Rather than requiring a general of \tilde{F} and \tilde{B} that computes the macrostep image for any element of the domain Q , it is sufficient to construct an approximation of \tilde{F} and \tilde{B} relative to the specific Q_i at hand. Call these Q_i -specific functionals \tilde{F}_{Q_i} and \tilde{B}_{Q_i} respectively. The following development constructs these element-specific functionals and shows their relationship to the general case.

Recall that the old macrostate is referred to as δ_0 :

$$Q_i = Q_{\delta_0}$$

Also observe that the first δ -step elements are:

$$Q_{\delta_1} = F_\delta \{ Q_{\delta_0} \} \tag{Eq 3-49}$$

$$Q_{\delta_{-1}} = B_\delta \{ Q_{\delta_0} \} \tag{Eq 3-50}$$

Examining Eq 3-43 shows that when Eq 3-42 holds then Q_{δ_1} is a least fixed point of F_δ relative to Q_{δ_0} of Eq 3-49. Respectively an examination of Eq 3-45 shows that when Eq 3-44 holds then $Q_{\delta_{-1}}$ is a greatest fixed point of B_δ relative Q_{δ_0} of Eq 3-50. These properties allow for the macrostep image computations relative to Q_i to be defined. These are denoted \tilde{F}_{Q_i} and \tilde{B}_{Q_i} respectively. They are never constructed on their own but appear in application form exploiting their special property of being, in application, a relative fixed point of their respective primitive image functionals.

In application form they are:

$$Q_{i+1} = \tilde{F}_{Q_i}\{Q_i\} = \mu_{Q_{s_1}} F_{\delta} \quad (\text{Eq 3-51})$$

$$Q_{i-1} = \tilde{B}_{Q_i}\{Q_i\} = \nu_{Q_{s_{-1}}} B_{\delta} \quad (\text{Eq 3-52})$$

These specific definitions should be contrasted with the general definitions of Eq 3-40 and Eq 3-41. The significance of these two equations is that material representations of \tilde{F} and \tilde{B} need never be computed. The element-specific functionals \tilde{F}_{Q_i} and \tilde{B}_{Q_i} can always be used in their place. Presumably there are efficient symbolic representations for the Q_i which were lacking for the \tilde{F}_i and \tilde{B}_i .

3.4.4 A Non-Abstract σ -Time Semantics

This second example of a non-abstract semantics adopts the view that macrotime is not made up of an arbitrary-yet-finite number of smaller steps, but rather is made up of an explicit number of subdivisions. In fact these subdivisions form a *schedule* under which specific components of the system are activated. The schedule is called here σ and the form of time that it induces is called σ -steps. The computational model of the microsemantics is a network of generalized gates.¹ A network of such gates is depicted in Figure 3-16.

The behavior of a gate is fully abstract in the sense that it is specified solely in terms of its transition relation. A gate's transition relation may be nondeterministic and there may be combinational feedback among the gates.² The σ -time microsemantics explains how the forward and backward image computations are defined in terms of individual gate

1. The Combinational/Sequential model of Hojati and Brayton [107] [358] is an instance of this class of computational model. A more detailed analysis of the Combinational/Sequential model is presented in Section 5.3 of this work.

2. The explanation of how such seeming inconsistencies are resolved is interesting in and of itself: the inconsistencies are implicitly ignored by the microsemantics in the sense that the inconsistency is never observable outside the system. An explanation for this subtle effect is presented in Section 4.4.5.

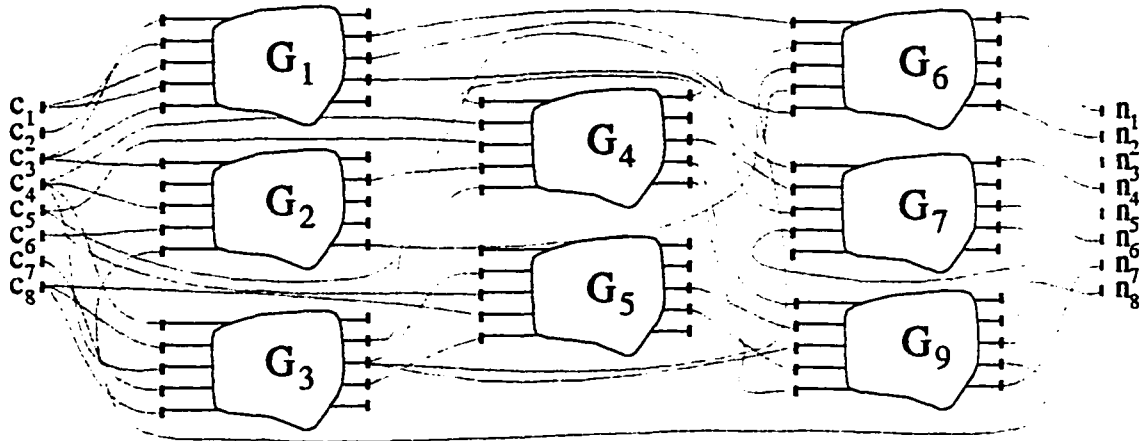


Figure 3-16. Communication in a Non-Abstract σ -Time Semantics components.

Of note in this microsemantics is that the schedule σ is entirely artificial. There are some broad constraints placed upon σ by the network structure, however there are a whole spectrum of possibilities for it. The selection of an appropriate schedule has been called the *quantification ordering problem* and has been extensively investigated [690] [107] [140] [43]. As such the presentation of this section is best viewed as an explication of existing techniques in the framework of computational semantics. Also, it should be obvious that the trivial schedule of one step is (almost) directly the full abstract semantics of Section 3.4.2. A return to these points is made in the observations of Section 3.4.4.8.

3.4.4.1 Temporal Analysis

In the σ -time model, time has a two-level structure as depicted in Figure 3-8. Within each macrostep there are a fixed number of σ -steps. This number, call it k , is a static function of the network connectivity, being related to the depth of network from the primary inputs. The rigid nature of σ -time is illustrated in Figure 3-11.

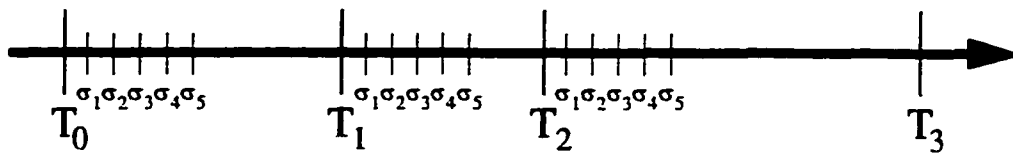


Figure 3-17. The Two Levels of a Non-Abstract σ -Time

The principle of unobservability at the finer granularity applies here as well. The σ -steps are entirely unobservable outside of M in this microsemantics just as they were in the δ -time microsemantics of Section 3.4.3. The only externally observable effect of σ -time is to induce a causal ordering relationship among the input/output elements.

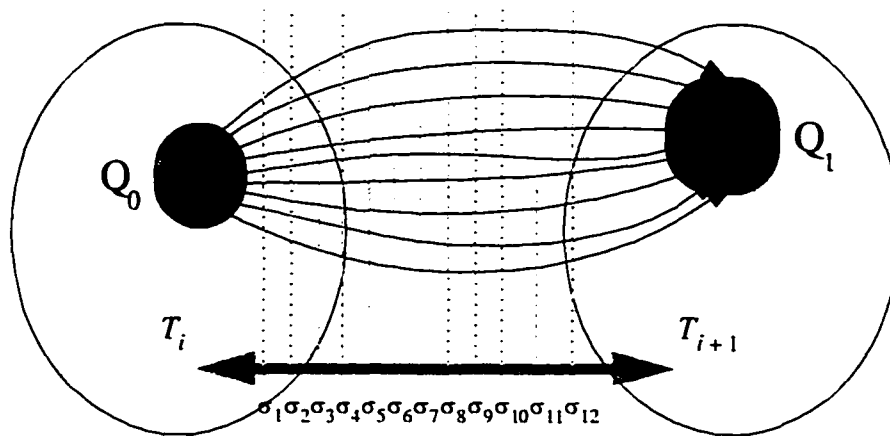


Figure 3-18. The Number of σ -Steps Are Defined by the Network Structure

3.4.4.2 Domain Definitions

The two salient characteristics of the σ -time semantics are the unobservability of the internal σ -steps and that each σ -path has the fixed length k which is dependent upon the network structure. The domains of M provide the explicit support for this, ensuring that a fixed point of the approximator functional occurs in exactly k steps. To ensure this, an

artificial schedule domain Σ is introduced, therefore:

- Σ is a vertical chain of length k ,
- S is non-flat a product domain of internal and external elements with a maximal element T in each subdomain.

The internal structure of σ -time domains depend on the particular relationships and elements in these domains so some explanation of them is required.

The Domain Σ

The domain Σ is the schedule. It is a finite set of elements which form an increasing chain:

$$\Sigma = \{\sigma_1, \sigma_2, \dots, \sigma_k\} \quad (\text{Eq 3-53})$$

By convention $\sigma_0 = \perp$, representing that the 0-th step of the schedule gives no information at all. The domain Σ is a finite vertical chain with the following order:

$$\forall 0 \leq i < k. \sigma_i \sqsubseteq \sigma_{i+1} \quad (\text{Eq 3-54})$$

Also, define the auxiliary functions:

$$succ = \lambda\sigma. \begin{cases} \text{if} & \text{then} \\ \sigma \equiv \sigma_0 & \sigma_1 \\ \sigma \equiv \sigma_1 & \sigma_2 \\ \dots & \dots \\ \sigma \equiv \sigma_{k-1} & \sigma_k \\ \sigma \equiv \sigma_k & \sigma_k \end{cases} \quad pred = \lambda\sigma. \begin{cases} \text{if} & \text{then} \\ \sigma \equiv \sigma_0 & \sigma_0 \\ \sigma \equiv \sigma_1 & \sigma_0 \\ \dots & \dots \\ \sigma \equiv \sigma_{k-1} & \sigma_{k-2} \\ \sigma \equiv \sigma_k & \sigma_{k-1} \end{cases} \quad (\text{Eq 3-55})$$

The Domain S

The construction of the state-domain S is a bit subtle because it must support the definition of monotonic functions that have the property of abstraction. That is, it must be possible to define monotonic functionals over S which *remove the "definedness" of their arguments*. This operation is motivated by the need to "undefine" (make "irrelevant") certain coordinates of S within a σ -step.

However, a monotonic functional which both introduces and removes (abstracts) definitions is infeasible in the direct sense. Using the terminology of Section 3.1, given domain elements $s \sqsubseteq t$, a function f is monotonic when $f(s) \sqsubseteq f(t)$. This means that the same monotonic function which “defines” can not also “undefine” because if $f(\perp) = s$ and $f(s) = \perp$ then f is non-monotonic. This motivates the use of a trick: an improper element \top which is the dual of \perp and is taken here to mean “irrelevant.” Thus $f(\perp) = s$ and $f(s) = \top$ and f is monotonic.

Let the “prototype” state domain S_T be a flat finite domain with a unique maximal element \top as well as a unique minimal element \perp in the usual sense:

$$S_T = \{s_{T_1}, s_{T_2}, \dots, s_{T_n}\} \quad (\text{Eq 3-56})$$

The structure of S_T is depicted in Figure 3-19. Its flat nature means that every state is incomparable to every other, yet it is possible to define monotonic functionals which can be said to “perform abstraction” on S_T . These functionals predictably have both a forward and backward interpretation and are defined as follows:¹

$$\exists_f = \lambda c. \lambda S. S_c \sqcup S_{\neg c} \quad (\text{Eq 3-57f})$$

$$\exists_b = \lambda n. \lambda S. S_n \sqcap S_{\neg n} \quad (\text{Eq 3-57b})$$

The μ -Calculus existential quantifier is here extended to the domain S_T in a way which precisely models the intuitive notion of making c , respectively n , irrelevant in S .

Consider the two-element subdomain shown in Figure 3-12. Treating that domain as primitive and considering a Cartesian pair results in the domain shown in Figure 3-13. In that domain it is always possible to define a monotonic function f which moves from a first element α where the c -coordinate is defined and the n -coordinate is undefined to a second element β where the c -coordinate is irrelevant and the n -coordinate is defined.

1. Assuming that c is defined on the boolean domain.

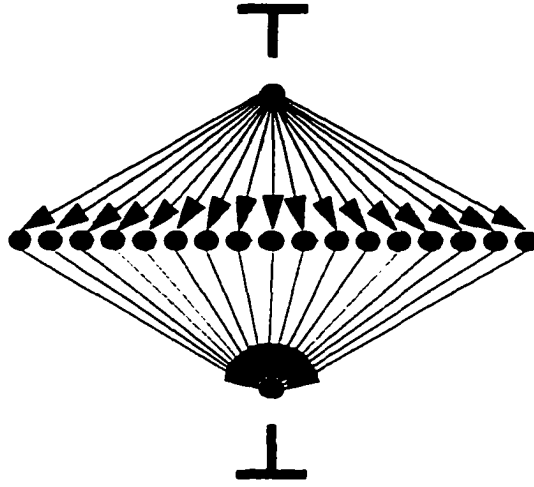


Figure 3-19. The Lattice Structure of S_T

Let S_{in} , S_c and S_n be Cartesian product domains of a number of these sorts of domains:

$$S_{in} = \prod_{i=1}^l S_{T_i} \quad (\text{Eq 3-58})$$

$$S_c = \prod_{i=1}^m S_{T_i} \quad (\text{Eq 3-59})$$

$$S_n \sim S_c. \quad (\text{Eq 3-60})$$

Thus, S_{in} will be the “internal” and S_c and S_n will be the “external” states of the system respectively. The external variables are visible to external observers. When they are read, they can be considered to be available at time σ_0 and when they are written they can be considered to be defined by σ_k . The internal states are not observable to external observers; they are written and read wholly within the σ -schedule. Presumably S_c and S_n are similar because they are the present-state and next-state components of the macrostep. The complete state domain of M is a cartesian combination of the three:

$$S = S_c \times S_{in} \times S_n \quad (\text{Eq 3-61})$$

The projectors are $\pi_{in}:S \rightarrow S_{in}$, $\pi_c:S \rightarrow S_c$ and $\pi_n:S \rightarrow S_n$ respectively.

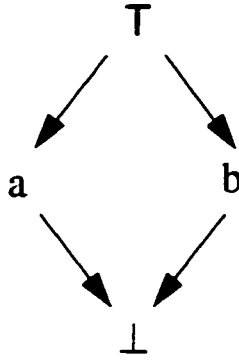


Figure 3-20. An Example of the Primitive Domain S_T

The Domain Q

The domain Q of the model involves both the schedule and the states:

$$Q = \Sigma \times S \quad (\text{Eq 3-62})$$

Its projectors are $\pi_\Sigma: Q \rightarrow \Sigma$ and $\pi_S: Q \rightarrow S$ respectively.

3.4.4.3 The Primitive Transition Relation $T_\sigma(c, n)$

The transition relation at a particular σ -cycle is the product of the generalized gates scheduled for that time slot:

$$T_G(c_G, n_G) = \text{a relation over } c_G \text{ and } n_G \text{ which is specific to } G \quad (\text{Eq 3-63})$$

$$T_{\sigma_i}(c, n) = \prod_{j=1}^{|\mathcal{G}_\sigma|} T_{G_j}(c_{G_j}, n_{G_j}) \quad (\text{Eq 3-64})$$

$$\sigma = \left\{ (c, T) \mid \begin{array}{l} c \text{ states to make irrelevant} \\ T \text{ the transition relation} \end{array} \right\} \quad (\text{Eq 3-65})$$

Presumably the *succ* and *pred* functionals have absorbing end conditions such that:

$$\text{forward:} \quad \text{succ}(\sigma_k) = \sigma_k, T_{\sigma_k} = \text{true}, c_{\sigma_k} = \emptyset.$$

$$\text{backward:} \quad \text{pred}(\sigma_0) = \sigma_0, T_{\sigma_0} = \text{true}, c_{\sigma_0} = \emptyset.$$

What is significant about this formulation is that the T_{σ_i} for $0 \leq i < k$ are all entirely unrelated. The purpose of the σ -schedule is to iterate over T_{σ} performing the actions indicated by Eq 3-64. Communication occurs through the sharing of variables across the scheduling steps σ_i . The “smoothing variables” c_{σ_i} represent the coordination variables that have already been defined, consumed and will never be used in a future scheduling step. They are “smoothed out” with the existential quantifier functionals Eq 3-57f and Eq 3-57b, thereby setting their coordinates to T .

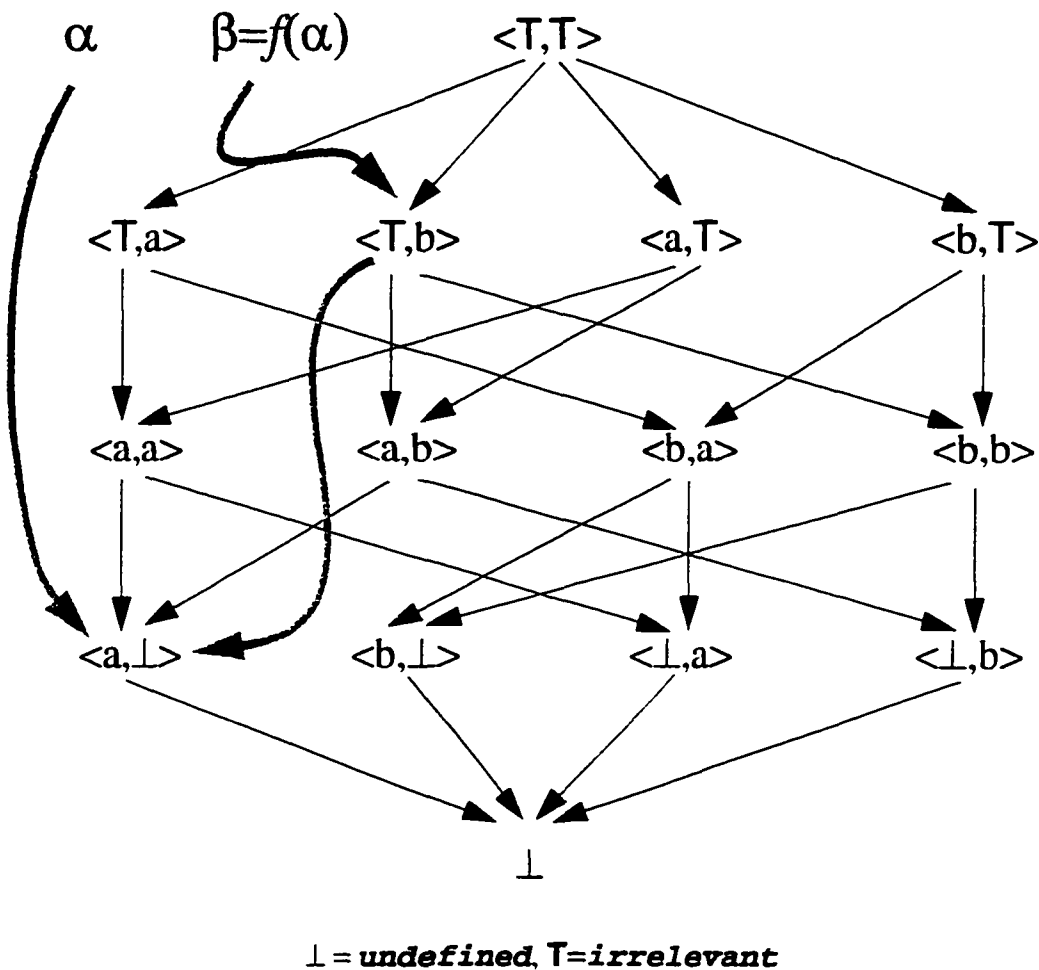


Figure 3-21. The Utility of \perp and T in the Domain $S_T \times S_T$

3.4.4.4 The Primitive σ -Image Functionals $F_\sigma \{Q\}$ and $B_\sigma \{Q\}$

In this microsemantics the primitive σ -image functionals apply the relevant element of the schedule, namely σ_i from Eq 3-65, returning that result and a sort of “continuation.” The continuation indicates which element of the schedule to compute next. In the forward case this continuation is computed by $succ(\sigma_i)$ and in the backward case by $pred(\sigma_i)$. The primitive σ -step functionals are:

$$F_\sigma = \lambda\sigma, Q. \exists_f c_\sigma. (succ(\sigma), Q(c, n) \wedge T_\sigma(c_\sigma, n_\sigma)) \quad (\text{Eq 3-66})$$

$$B_\sigma = \lambda\sigma, Q. \exists_b n_\sigma. (pred(\sigma), T_\sigma(c_\sigma, n_\sigma) \wedge Q(c, n)) \quad (\text{Eq 3-67})$$

In application form they define the σ -time as:

$$\left(\sigma_{i+1}, Q_{\sigma_{i+1}} \right) = F_\sigma \{ \sigma, Q_{\sigma_i} \} \quad (\text{Eq 3-68})$$

$$\left(\sigma_{i-1}, Q_{\sigma_{i-1}} \right) = B_\sigma \{ \sigma, Q_{\sigma_i} \} \quad (\text{Eq 3-69})$$

3.4.4.5 The Approximator Functionals $\mathcal{F}_\sigma \{ \tilde{F} \}$ and $\mathcal{B}_\sigma \{ \tilde{B} \}$

The approximator functionals are, again, merely control skeletons which are designed to apply the primitive apply primitive σ -image functionals:

$$\mathcal{F}_\sigma = \lambda F. \lambda\sigma, Q. F \{ \sigma, Q \} \quad (\text{Eq 3-70})$$

$$\mathcal{B}_\sigma = \lambda B. \lambda\sigma, Q. B \{ \sigma, Q \} \quad (\text{Eq 3-71})$$

3.4.4.6 The Approximated Image Functionals $\tilde{F} \{Q\}$ and $\tilde{B} \{Q\}$

The approximated image functionals are the least and greatest fixed points of Eq 3-70 and Eq 3-71 respectively:

$$\tilde{F} = \mu_{F_\sigma} \mathcal{F}_\sigma \quad (\text{Eq 3-72})$$

$$\tilde{B} = \nu_{B_\sigma} \mathcal{B}_\sigma \quad (\text{Eq 3-73})$$

By construction, these fixed points happen at cycle σ_k . At that point the following relationships hold:

$$\begin{aligned} \bar{F}_k &= \mathcal{F}_\sigma\{\bar{F}_k\} & \bar{B}_k &= \mathcal{B}_\sigma\{\bar{B}_k\} \\ \bar{F} &= \bar{F}_k & \bar{B} &= \bar{B}_k \\ (\sigma_k, Q_{\sigma_{k+1}}) &= \bar{F}\{\sigma_k, Q_{\sigma_k}\} & (\sigma_k, Q_{\sigma_{k-1}}) &= \bar{B}\{\sigma_k, Q_{\sigma_k}\} \end{aligned}$$

3.4.4.7 The Projection $\bar{\Pi}$ to Full Abstraction

The projection back down to the fully abstract case is actually straightforward given a simple observation. That observation is that all the real information is in the external space S_n . By definition all the internal space S_{in} and the components of the previous macrostep S_c have been sent to \bar{T} . All that is necessary is to substitute from S_n to S_c and then project away the unobservable internal components. The full abstraction projectors are as follows:

$$\Pi_f = \lambda Q. \pi_c\{\pi_S\{Q[n/c]\}\} \quad (\text{Eq 3-74f})$$

$$\Pi_b = \lambda Q. \pi_n\{\pi_S\{Q[c/n]\}\} \quad (\text{Eq 3-74b})$$

This leaves but to compose the projection functions with the approximated image functionals:

$$F = \bar{F} \circ \Pi_f \quad (\text{Eq 3-75})$$

$$B = \bar{B} \circ \Pi_b \quad (\text{Eq 3-76})$$

3.4.4.8 Observations

There are three observations to be made about the non-abstract semantics of σ -time.

Semi-Serializability of Concurrent Coordination

The generalized gate network model is a combination of serialization and concurrency. The microsemantics allows the image computations to “compile away” states which are used only for internal coordination once their effect is known and consumed. As such the

feasibility of computations using this microsemantics are strongly dependent on the connectivity within the network.

In general, communication dependencies are dense within a gate. Within a gate, each output element depends on many, if not all, inputs. In contrast, the communication dependencies at the inter-gate level are often quite simple. It is known¹ that certain sparsely connected structures such as rings, linear arrays and trees have good schedules σ . On the other hand, coordination structures which are densely connected often do not have good schedules.

Output-less State Space

In the generalized gate network model there is no explicit domain of outputs *per se*. All the domains are really domains of states. This can be seen in the domain definition of Eq 3-61 which defines the domains of all the “wires” in the network as members of the state domain. There is a distinguished domain of previous states S_c , a distinguished domain of output states S_n and of course the domain of internal wires S_{in} . The outputs of the generalized gates appear either in S_n or S_{in} . In particular the transition relations of gates, as defined in Eq 3-63, are atomic in the sense that they merely relate the gate inputs to gate outputs. Thus when there is nondeterminism in the network, it is directly translated into a multiplicity of states which are stored in an intermediate Q_{σ_i} . This is a potential source of state explosion in Q_{σ_i} during the steps of a schedule.² This form of nondeterminism can be said to be “imperative” because it is only exposed during the steps of an approximated image computation. In contrast the nondeterminism of the fully abstract case can be said to be “declarative” because it exists statically in the X_{P_i} or O_{P_i} of Eq 3-14.

1. *c.f.* Aziz *et al.* [42] [43] which summarize several classes of network connectivity based on their own analysis. Previously-published results are also summarized there.

2. In fact, it has been observed that it is often best to artificially determinize such networks with a minimal number of unconstrained inputs [653].

In contrast, the transition relation of the fully abstract semantics of Section 3.4.2, defined in Eq 3-14, had two components a “pure” transition relation $X(c, i, n)$ and an output relation $O(c, i, o)$. In that case the nondeterminism was “declarative” in the sense that it existed as a static element in either X or O . Significantly, this sort of declarative nondeterminism is exposed and resolved instantly within the single macrostep. It does not result in the same kinds of intra-step state explosion.

Illusory Nature of the Domains Σ and S_T

The presentation of the σ -time microsemantics may seem a bit odd at first. In particular there is the scheduling subdomain Σ defined in Eq 3-53 and the oddly-shaped prototype state subdomain S_T of Eq 3-56. Both of these items are rarely found in material form in actual implementations of either physical devices or formal analysis packages. That being the case, the obvious question then is why they are in the domain analysis of the microsemantics?

The answer to this question is subtle and lies with how domain and domain elements must be interpreted in the real world. Domain equations are not *direct specifications for synthesis*, that is they are not minimal descriptions of the concrete elements which any conforming implementation must have. Rather, they are *mathematical structures* which are used to precisely explain the computations being performed in the sense that the explanation is rigorously defined relative to a theory of computation and is minimally constraining in the mathematical sense of not containing gratuitous restrictions. Domains, by construction, contain at least one and possibly more improper elements which are used to denote mathematically relevant quantities. These elements denote quantities such as “is undefined,” which is written here as \perp and “is irrelevant,” which is written here as \top . When a topologically-defined function space (a mathematical structure) is used to model notions of computability in the precise sense, such elements are necessary and indeed are

required. This of course begs the question of how these domain elements *encoded* when the topologically-constructed function space is turned into a specification for synthesis.

It may be the case that in a particular implementation setting certain domain elements can have a trivial and even vacuous encoding. A common example of this is the element \perp which is taken to mean “is undefined.” That any set can be made into a flat domain by putting \perp underneath every element does *not* imply that there must be a special distinguished encoding for \perp which is stored in memory elements or in the symbolic form. In the precise sense what \perp means is a promise, or a requirement, not to make decisions based on the value. As such \perp can be encoded *arbitrarily* so long as the surrounding computations are *faithful* to the precise meaning of the element. In the case of \perp this means that the surroundings must never make decisions based on the value in that position at that time.

Analogously, the improper element \top need not have an explicit encoding either. Its encoding may remain implicit in the same way by the agreement of the surrounding computations to never read the (actual) value in that position at that time. This implicit encoding is faithful to the precise meaning of \top as denoting that the value is “irrelevant” or contradictory.

Extending this idea even further it can be observed that the schedule σ can be encoded in this implicit way so long as the particular implementation is faithful to the precise meaning of σ . Actually the domain Σ with its k schedule elements σ_i , its vertical ordering $\sigma_i \sqsubseteq \sigma_{i+1}$ and the functions *succ* and *pred* are merely describing, in the language of topology, that certain activities happen in a certain order (as described by the element of the function space which computes that order). As such, a conforming implementation could dispense with the material representation of σ with the understanding that the behavior that it controlled was implicitly encoded in some faithful way. In a physical

device, the schedule σ is implicitly implemented by the causal flow of time. In a formal analysis package, where one has the freedom to manipulate time in a non-causal way, the schedule σ can be faithfully implemented by evaluating the primitive image functionals in the indicated order.

Computations described in terms of domain equations often require domains containing improper elements or noisome control components. These are required to complete the definitional aspects of the topological spaces that are being manipulated. The power of the denotational method is that it precisely states the computations which must be performed and the relationships which must hold. The denotational theory is precise in the sense that it does not require any particular encoding for those computations or relationships. As such, any encoding which provides a faithful representation of the domain element's properties may be used. Faithful encodings are extremely convenient for improper domain elements which don't have an obvious material form. Indeed, the general utility of temporal or vacuous encodings for domain elements cannot be underestimated.

3.4.5 Focus

Microsemantic analysis was introduced in this section as a structured means for organizing and understanding microsemantics. Three semantics were analyzed in this way and shown to have particular properties. The first was the fully abstract transition semantics of coordinating Mealy machines. The microsemantics of full abstraction was shown to "compile away" all of the coordination between components and to define a single-level discrete time. The two semantics which followed were non-abstract and were of interest because they were computational.

The second microsemantics defined the time line with a two-level structure. Its finer granularity was called δ -time. The δ -time was completely unobservable at the macrotime level and there could be an arbitrary number of δ -steps in a macrostep. The relationship

between δ -time and macrotime was a fixed point of the image approximator functional. The microsemantics of δ -time represented coordination activities explicitly in the transition relation and outputs were but a coordinate of the δ -state space with a special non-flat domain ordering. This simplified the transition relation T_δ at the expense of a larger state space (most of which was unobservable in δ -time).

The third and final microsemantics had a two-level time as well. Its finer granularity was called σ -time. The σ -time was unobservable at the macrotime level as well however there were a fixed number of σ -steps in a macrostep. The relationship between σ -time and macrotime was that σ -time was k times as fast as macrotime. This semantics was computational because its approximator functional was well-defined. In fact, the schedule σ was shown to force a fixed point on the approximator functional at exactly the k -th step. The microsemantics of σ -time represented coordination activities implicitly in the network of generalized gates. The interconnection among the gates allowed for the derivation of the schedule σ in such a way that the unused intermediate state components could be abstracted away. This microsemantics was shown to be a denotational formalization of the well-known quantification ordering problem.

3.5 Review

The presentation of this chapter has outlined computational semantics as a framework for understanding the interface between a language L and its fully abstract model M or its non-abstract model M_c . Computational semantics extends the argument of Chapter 2 where the transition relation was argued to be the fundamental essence of a semantics. However, a transition relation is a non-directional entity so the concept of denotations was extended here to include a directional component in the image semantics defined by the forward and backward image computations $F\{Q\}$ and $B\{Q\}$. Computational semantics is the framework in which statements can be made about when a non-abstract semantics

S_0 can be said to substitute for the fully-abstract case. The condition of substitutability was shown to be the existence and well-definedness of a projection Π which, when composed with S_0 , obeyed the relationship $S = S_0 \circ \Pi$.

To show this, Scott's domain theory was presented as a means of connecting the concept of multi-step computations being equivalent to a final result with the already-established notion of approximation, continuity and limits from algebraic topology. This required the definition of domains which were, in simple terms, a (not necessarily finite) set D where a relationship \sqsubseteq and certain limits and consistency conditions were known to hold. In particular there was always a minimal element $\perp \in D$. This distinguished element represents the undefined or unspecified value and is called an improper element because there need not be a material representation for it. The concept of a monotonic and continuous functions was defined on domains: a function is monotonic if it preserves \sqsubseteq ; a function is continuous if it preserves monotonicity in the infinite limit. Monotonic functions on finite domains are necessarily continuous; continuity requires that infinite results be determined by the limit of finite approximations. This characterization of functions in turn allows for the notion of an approximation to be defined wherein any computable function can be understood as the upper bound of a (possibly infinite) limit series of approximations to the final result.

Continuity is a subtle concept for it necessarily precludes a definition of fairness. Because unbounded nondeterminism coupled with fairness constraints is discontinuous in the infinite limit. This was shown in the construction of a behavioral domain satisfying $B = IN \rightarrow (OUT \times B)$ in Section 3.2. Fairness is far too attractive a paradigm for this "result" to be used as a proof of its nonviability. So if computational semantics was not directly useful in defining trans-macrostep behavior, then perhaps it could be used to define behavior within a step. Having a means for approximating a macrostep by a series of smaller steps, \circ -steps in the general case, would be useful both as a means for explain-

ing when certain finer granularities of time such as δ -time or σ -time were well-defined. Such a theory could be used in practical implementations to avoid the many practical bottlenecks of full abstraction.

Microsemantic domains and microsemantic analysis was the framework used to construct macrostep image semantics which were defined in terms of iterated microstep image semantics. The key to microsemantics were the internal structure of the microsemantic domains and the \sqsubseteq relationships that held within those domains. The \sqsubseteq relationships of course implied monotonicity which was used to define the approximator functionals \mathcal{F}_\circ and \mathcal{B}_\circ for forward and backward image step. In turn, the least and greatest fixed points of these functionals, $\mu_{F_\circ} \mathcal{F}_\circ$ and $\nu_{B_\circ} \mathcal{B}_\circ$, relative to some initial estimate F_\circ and B_\circ were established as the approximated image computations $\bar{F}\{Q\}$ and $\bar{B}\{Q\}$ corresponding to $F\{Q\}$ and $B\{Q\}$ respectively. Two non-abstract microsemantics were studied in this framework, the δ -time corresponding to the semi-interleaved executions of an abstract machine and the σ -time corresponding to the execution of a generalized gate network according to a precomputed leveled schedule.

The microsemantic analysis showed that a semantic map S_\circ induces a fine structure onto time. In the fully abstract case, this fine structure was vacuous and so time only had a single level. In the case of the non-abstract semantics, the structure of time was much more interesting. Both of the non-abstract microsemantics studied in this chapter had a two-level structure to time. In the case of the δ -time an arbitrary number of δ -steps were embedded between the macrosteps, while in the case of the σ -time a fixed number of σ -steps were embedded between the macrosteps. In both cases the microsemantic structure between the macrosteps was always finite and well-defined. However it must be pointed out that there could exist microsemantics in which the microstep behavior need not be either well-defined or finite. The possibility of three-level time or even multi-dimensional time was not explored in the microsemantic framework developed here. In fact three-level

microsemantics and multi-dimensional time do exist in the semantics of discrete events and the denotational model of the SL Languages respectively. Their presentation is deferred until after Chapter 4 where certain intrinsic limitations on the microsemantic approach are explored.

The whole point of non-abstraction is that a non-abstract semantics contains extra idiosyncratic artifacts which can be ignored in a well-defined way. Once ignored, what remains is the canonical definition of the semantics: the fully abstract case. Implementation details are ignored through a projection map $\Pi: M_{\circ} \rightarrow M$ that hides the details. This establishes the path $S_{\circ} \circ \Pi$ of Figure 3-1 and in that context substitutability is the condition when $S = S_{\circ} \circ \Pi$ for some $L' \subseteq L$ and $M' \subseteq M$. Unfortunately, it turns out that there are strong limitations that force the containment of L' and M' to be proper for any semantics, although the closeness to equality is strongly dependent on the particulars of the microsemantics S_{\circ} . Those limitations are the subject of the following chapter.

4 Limits on Microsemantics

Despite the range of microsemantics that can be defined, there are some hard limits on the amount of internal structure that can exist in a semantic model. This chapter reviews the three aspects of internal structure in semantic models: *responsiveness*, *modularity* and *causality*. The origins of these properties are shown in Figure 4-1. The key insight here is that despite the convenience of computational semantics as defined in Chapter 3, the approximation approach brings along with it some mathematical baggage that just cannot be removed by the projection Π . This statement is formalized in Huizing and Gerth's RMC Barrier Theorem [378] which states that no semantics, no matter what its internal microstructure, can be responsive, modular and causal all at the same time. This places severe constraints on the approaches to the design of languages, semantics and models.

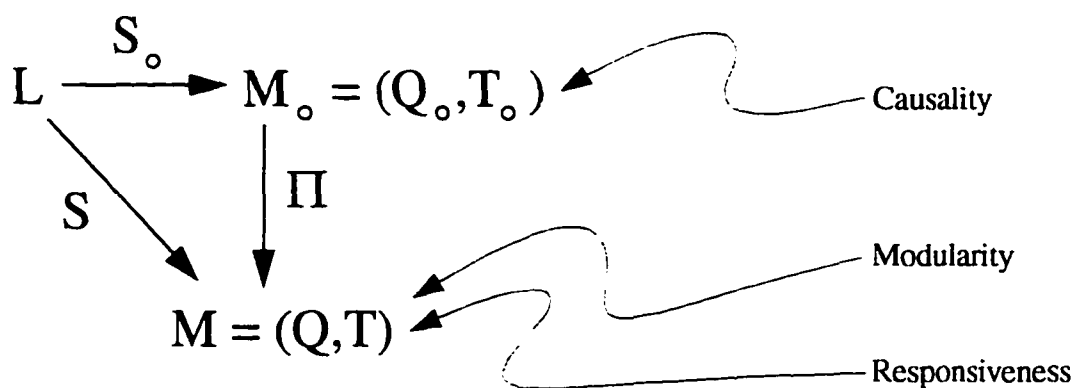


Figure 4-1. The Origins of Structural Properties in Semantics Models

The formal proof is presented here for two reasons. The first is that the concept of a limitation on the internal organization within a semantic model is certainly not intuitively obvious; a proof is therefore required. A more practical reason for including the proof is that I present it here in terms of the μ -Calculus notation rather than the process algebraic formulation used in Huizing and Gerth's original formulation. The importance of this notational change is that the process algebraic notation expresses the conditions using a point-wise or depth-first notion of computation. The μ -Calculus notation on the other hand expresses the conditions using a breadth-first relational style. As such, the proof relates directly to the OBDD-based algorithms which are assumed throughout. Seen in terms of the μ -Calculus notation, the RMC barrier shows that the limits on structure in semantic models carry over directly to limits on structure in transition relations represented symbolically as OBDDs.

4.1 Orthogonal Aspects of a Semantics

The RMC Barrier is defined in terms of three orthogonal properties of a semantic models: responsiveness (R), modularity (M) and causality (C). The preceding sections have illustrated that, except for some very non-standard models, all semantics is given by a (possibly infinite) transition relation. The focus here is exclusively on the finite-state case. There, a transition relation is a set of tuples relating the current state, the input, the output and the next state of the structure. Using the multi-valued variables c , i , o and n to represent values in these respective spaces, the transition relation is given by the characteristic function of the set:¹

1. It is an interesting irony that the transition relation which was argued to be fundamental in some deep sense in Chapter 2 is defined in Eq 4-1 in terms of a process algebra notation! This is expository and related to the notation used in Huizing and Gerth's original proof of the RMC Barrier Theorem.

The process algebra notation is not fundamental to the development here. It is used where it affords some clarity over the relational notation. The development here is such that R , M , C and the mutual incompatibility of the three can be understood exclusively in relational μ -Calculus notational framework and thus its relevance to OBDD-based symbolic methods can be directly inferred.

$$T = \{ (c, i, o, n) \mid \text{the transition } Q(c) \xrightarrow[\begin{smallmatrix} O(o) \\ I(i) \end{smallmatrix}]{Q(n)} \text{ is valid} \} \quad (\text{Eq 4-1})$$

The notational convention used here is that T is written as $T(c, i, o, n)$ to emphasize the dependence on the four variables c , i , o and n . A decomposed view of the transition relation T gives its dependence in terms of a basic transition relation X relating the current state and the input to the next state, and an output relation O relating the current state and the input to the output:

$$T(c, i, o, n) = X(c, i, n) \wedge O(c, i, o) \quad (\text{Eq 4-2})$$

Without the loss of generality, X must be *complete* in the sense of enabling a transition to some next state on every possible input:¹ $\exists c, n. X(c, i, n) = 1$. This is the fully general form where the output of the machine is dependent both on the current state c as well as the input i . Eq 4-2 is that of a Mealy machine. The Moore machine where the output is dependent only upon the state is:

$$T(c, i, o, n) = X(c, i, n) \wedge O(c, o) \quad (\text{Eq 4-3})$$

The properties R , M and C are conditions on the compatibility of X and O under composition. The following sections reiterate the definitions of the RMC properties and enumerate these conditions in terms of the transition relation T .

4.1.1 Responsiveness (R)

A system is considered responsive if the system's output comes simultaneously with the input that causes it. A semantics is responsive if it is possible to define a responsive system in the semantics. This is a condition on an instance of a semantics; it is an existential

1. Kurshan (c.f. [454], page 116) argues that incompleteness is a "flaw" in the description of any actual system. A construction is outlined that removes such flaws without removing any infinite behaviors.

statement about a system definable in the semantics. This condition states that in some state of such a system, it is possible to distinguish the input given to the system by virtue of examining the output that it produces. That is, it is possible to *exactly characterize* the input given the output. The condition concerns the possibility of defining a system with this property. It need not be true of all systems or of all states of a given system. It need only be true of one state in some system.

Let $I_1(i)$ identify a nonvacuous subset of the inputs in such a system with transition relation T . Let $O_1(o)$ be a nonvacuous subset of the outputs similarly. Define the relational operator $\Phi\{I_1, O_1\}$ as:

$$\Phi\{I_1, O_1\}(c, i, o) = \exists n. T(c, i, o, n) \wedge I_1(i) \wedge O_1(o) \quad (\text{Eq 4-4})$$

Define $\Phi\{I_2, O_2\}(c, i, o)$ similarly. A responsive state is one where there exists input sets $I_1(i)$ and $I_2(i)$ and outputs $O_1(o)$ and $O_2(o)$ such that the outputs differ based on the inputs. The following definition specifies this condition independently of the state in the form of a constraint on the generalized input/output relation:¹

$$\frac{\partial O}{\partial I}(i, o) = \begin{cases} (I_1(i) \oplus I_2(i)) \wedge (O_1(o) \oplus O_2(o)) \\ \wedge \\ (I_1(i) \Rightarrow O_1(o)) \wedge (I_2(i) \Rightarrow O_2(o)) \end{cases} \quad (\text{Eq 4-5})$$

With these definitions, the responsiveness property $R(i, o)$ is:

$$R(i, o) = \exists I_1, I_2, O_1, O_2. \exists c. \left(\begin{array}{c} \left(\Phi\{I_1, O_1\}(c, i, o) \oplus \Phi\{I_2, O_2\}(c, i, o) \right) \\ \wedge \\ \frac{\partial O}{\partial I}(i, o) \end{array} \right) \quad (\text{Eq 4-6})$$

This equation holds just when there is at least one state $Q(c)$ where input-dependent outputs can be observed. As such, responsiveness is a structural condition on the separability

1. I thank Rick McGeer for pointing out relevance and cogency of the Boolean difference notation in defining the generalized responsiveness condition.

of the basic transition relation X and the output relation O . This can be seen by examining Eq 4-6 with the transition relation for a Mealy machine (Eq 4-2) and a Moore machine (Eq 4-3).

Mealy Machines

Taking the expansion of Eq 4-6 in the Mealy case of Eq 4-2 case first:

$$R(i, o) = \exists I_1, I_2, O_1, O_2, \exists c. \left(\begin{array}{c} \left(\begin{array}{c} \exists n. T(c, i, o, n) \wedge I_1(i) \wedge O_1(o) \\ \oplus \\ \exists n. T(c, i, o, n) \wedge I_2(i) \wedge O_2(o) \end{array} \right) \\ \wedge \\ \frac{\partial O}{\partial I}(i, o) \end{array} \right) \quad (\text{Eq 4-7})$$

Expanding through the existential quantification of the next state n gives:

$$R(i, o) = \exists I_1, I_2, O_1, O_2, \exists c. \left(\begin{array}{c} \left(\begin{array}{c} T(c, i, o) \wedge I_1(i) \wedge O_1(o) \\ \oplus \\ T(c, i, o) \wedge I_2(i) \wedge O_2(o) \end{array} \right) \\ \wedge \\ \frac{\partial O}{\partial I}(i, o) \end{array} \right) \quad (\text{Eq 4-8})$$

By using the identity of the generalized cofactor, that $f_c(x) = f(x) \wedge c(y)$, produces a function $f_c(x)$ which is not dependent on y ¹ allows Eq 4-8 to be rewritten as:

$$R(i, o) = \exists I_1, I_2, O_1, O_2, \left(\exists c. \left(T_{I_1 \wedge O_1}(c) \oplus T_{I_2 \wedge O_2}(c) \right) \wedge \frac{\partial O}{\partial I}(i, o) \right) \quad (\text{Eq 4-9})$$

The first term of the conjunction, the existential quantification, identifies a state $Q(c)$ which has two (disjoint) transitions out of it on I_1/O_1 and I_2/O_2 respectively. The second term, the boolean difference, constrains the values of the I_1, I_2, O_1 and O_2 as per Eq

1. In fact, any function f_c satisfying $f \wedge c \subseteq f_c \subseteq f \vee \bar{c}$ is sufficient. The justification for the generalized cofactor can be found in Brown [120] under the treatment of orthonormal expansions. Several implementation methods have been proposed [208] [690].

4-5. This expression identifies the transition structure shown in Figure 4-2. Clearly such a structure can be created in a Mealy machine.

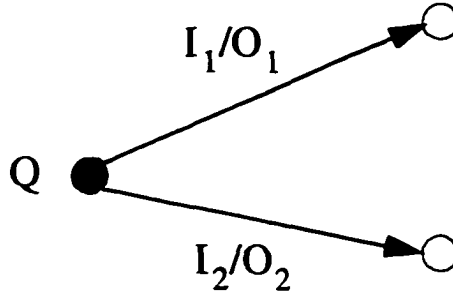


Figure 4-2. The Responsive State Transition Structure $R(i,o)$

$\Phi(c, i, o)$ may also be considered to be an aggregated representation of a state-dependent partial order $\leq_{Q(c)}(i, o)$ in the sense that operationally, for the chosen state $Q(c)$, the indicated inputs must be available to the machine before its outputs can be computed. This interpretation of Φ as a partial order is critical to the proof of the RMC Barrier Theorem.

Moore Machines

Taking the expansion of Eq 4-6 in the Moore case of Eq 4-3 one gets:

$$R(i, o) = \exists I_1, I_2, O_1, O_2, \exists c. \left(\begin{array}{c} \left(\begin{array}{c} \exists n. \left(\begin{array}{c} X(c, i, n) \wedge O(c, o) \wedge \\ I_1(i) \wedge O_1(o) \end{array} \right) \\ \oplus \\ \exists n. \left(\begin{array}{c} X(c, i, n) \wedge O(c, o) \wedge \\ I_2(i) \wedge O_2(o) \end{array} \right) \end{array} \right) \\ \wedge \\ \frac{\partial O}{\partial I}(i, o) \end{array} \right) \quad (\text{Eq 4-10})$$

Removing the existential quantification over n as before:

$$R(i, o) = \exists I_1, I_2, O_1, O_2 \left(\begin{array}{c} \exists c. \left(\begin{array}{c} X(c, i) \wedge O(c, o) \wedge I_1(i) \wedge O_1(o) \\ \oplus \\ X(c, i) \wedge O(c, o) \wedge I_2(i) \wedge O_2(o) \end{array} \right) \\ \wedge \\ \frac{\partial O}{\partial I}(i, o) \end{array} \right) \quad (\text{Eq 4-11})$$

And applying the generalized cofactor as before:

$$R(i, o) = \exists I_1, I_2, O_1, O_2 \left(\begin{array}{c} \exists c. \left(\begin{array}{c} X_{I_1}(c) \wedge O_{O_1}(c) \\ \oplus \\ X_{I_2}(c) \wedge O_{O_2}(c) \end{array} \right) \\ \wedge \\ \frac{\partial O}{\partial I}(i, o) \end{array} \right) \quad (\text{Eq 4-12})$$

Since X is complete by definition, the following identity holds: $X_{I_1}(c) = X_{I_2}(c) = 1$.

This leaves the expression:

$$R(i, o) = \exists I_1, I_2, O_1, O_2 \left(\begin{array}{c} \exists c. O_{O_1}(c) \oplus O_{O_2}(c) \\ \wedge \\ \frac{\partial O}{\partial I}(i, o) \end{array} \right) \quad (\text{Eq 4-13})$$

This form makes explicit that there must be some state $Q(c)$ which can potentially output either $O_1(o)$ or $O_2(o)$. For this to be true, it must be the case that the state(s) $Q(c)$ found by the existential quantifier (the $\exists c$ term) have the following property in the output relation $O(c, o)$:

$$\left. \begin{array}{l} Q(c) \wedge O_1(o) \wedge \overline{O_2(o)} \\ \vee \\ Q(c) \wedge \overline{O_1(o)} \wedge O_2(o) \end{array} \right\} \subseteq O(c, o) \quad (\text{Eq 4-14})$$

In other words, for a Moore machine to be potentially responsive it must have a nondeterministic output relation. But in such a case, the input which causes either output cannot be exactly characterized by examining the output. That the output is undetermined by the

input is the very definition of non-determinism (with respect to outputs). In contrast, a Moore machine with *deterministic* outputs¹ is non-responsive since it associates a unique output with each state. In the context of Eq 4-14 this means that there is but one conjunctive term. Thus there is no non-vacuous $O_2(o)$, different than $O_1(o)$, against which a difference can be observed. In that case Eq 4-13 is tautologically false: no state constructible in the semantics can satisfy the requirement of $R(i, o)$. The responsiveness property is thus exactly the distinction between a Mealy machine and a Moore machine.

A semantics is responsive just when $R(i, o)$ is not tautologically false.

4.1.2 Modularity (M)

A semantics is modular if the rule for aggregating components into a whole obeys the property that all parts of the system can be treated symmetrically, inclusive of intra-component communications and component-to-environment communications. Further, every part of the system must have the same view of the instant-to-instant computation. Modularity is fundamentally an information hiding property that the communication between two components is accomplished solely via the values on outputs. In particular modularity prevents communication from occurring via any intra-step order in which the outputs are produced or via any multiplicity of values assigned on the outputs across a step.

This statement describes the separability of the aggregate transition relation in forward image computations:²

1. The usual definition requires determinacy [368], though nondeterministic case can be made well-defined by a mild condition of *independence* between inputs and output selections [454]. Failing that, a nondeterministic Moore machine is actually the Mealy case phrased in a backwards sort of way (*i.e.* nondeterministically select the output and then identify which input must have allowed that output to occur).

2. For reasons of clarity, the syntactic substitution operation, $[n/c]$, is elided from the forward image computation in this chapter. This is done without loss of generality.

$$F\{Q\} = \exists c, i, o. Q(c) \wedge T(c, i, o, n) \quad (\text{Eq 4-15})$$

The modularity condition governs the separability of $F\{Q\}$ when generalized for concurrent composition with $Q = Q_1 \times Q_2$ and $T = T_1 \times T_2$. In the case of the state variables, c and n , the variables of the aggregate are just the concatenation of the variables of the parts:

$$\begin{aligned} c &= (c_1, c_2) \\ n &= (n_1, n_2) \end{aligned} \quad (\text{Eq 4-16})$$

The composite state $Q = Q_1 \times Q_2$ is just the conjunction of the characteristic functions of the component states. The component states are automatically “raised” to the product space because the sets are represented by their characteristic function:

$$Q(c) = Q(c_1, c_2) = (Q_1 \times Q_2)(c_1, c_2) = Q_1(c_1) \wedge Q_2(c_2) \quad (\text{Eq 4-17})$$

It must be noted that this raising operation may not be implicit for certain symbolic representations, *e.g.* the ZBDD [538].

For the case of the input and output variables, the two systems are connected together with the inputs being available to both. The outputs of the first are available to the second and vice versa. The input variables i and output variables o are merged together and called the input/output variables io . This is illustrated in Figure 4-3.

The input/output variables of the whole are defined as the concatenation of the variables of the parts:

$$\begin{aligned} io_1 &= (i, o_1) \\ io_2 &= (i, o_2) \\ io &= (i, o_1, o_2) \end{aligned} \quad (\text{Eq 4-18})$$

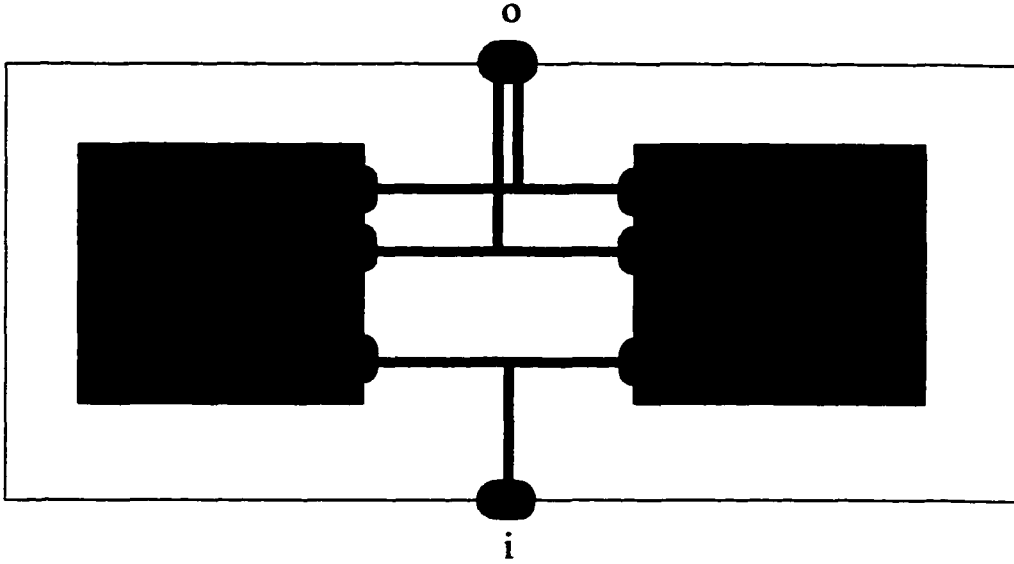


Figure 4-3. The Communication Structure of a Concurrent Composition

In this light the transition relations for the components are:

$$\begin{aligned}
 T_1(c_1, io_2, o_1, n_1) &= X_1(c_1, io_2, n_1) \wedge O_1(c_1, i, o_1) \\
 T_2(c_2, io_1, o_2, n_2) &= X_2(c_2, io_1, n_2) \wedge O_2(c_2, i, o_2)
 \end{aligned}
 \tag{Eq 4-19}$$

The composite transition relation $T = T_1 \times T_2$ is just the conjunction of the component transition relations:

$$\begin{aligned}
 T(c, io, n) &= (T_1 \times T_2)(c, io, n) \\
 &= (T_1 \times T_2)(c_1, c_2, i, o_1, o_2, n_1, n_2) \\
 &= T_1(c_1, io_1, n_1) \wedge T_2(c_2, io_2, n_2)
 \end{aligned}
 \tag{Eq 4-20}$$

The effect of the modularity condition on the forward image computation $F\{Q\}$ can now be stated:

$$F\{Q\} = \exists c, io. Q(c) \wedge T(c, io, n)$$

$$\begin{aligned}
F\{Q_1 \times Q_2\} &= \exists c, io. (Q_1 \times Q_2)(c) \wedge (T_1 \times T_2)(c, io, n) \\
&= \exists c, io. \left(\begin{array}{c} (Q_1 \times Q_2)(c) \\ \wedge \\ (T_1(c_1, io, n_1) \wedge T_2(c_2, io, n_2)) \end{array} \right)
\end{aligned}$$

This last term can be rewritten by expanding the definition of the transition relation T in terms of the basic transition relation X and the output relation O as:

$$F\{Q_1 \times Q_2\} = \exists c, io. \left(\begin{array}{c} (Q_1 \times Q_2)(c) \\ \wedge \\ X_1(c_1, io_2, n_1) \wedge O_1(c_1, io_1) \\ \wedge \\ X_2(c_2, io_1, n_2) \wedge O_2(c_2, io_2) \end{array} \right)$$

This in turn can be rewritten to separate the expressions for the progress according to the T_1 and T_2 :

$$F\{Q_1 \times Q_2\} = \exists c, io. \left(\begin{array}{c} \left(\begin{array}{c} (Q_1 \times Q_2)(c) \wedge \\ X_1(c_1, io_2, n_1) \wedge O_1(c_1, io_1) \wedge \\ O_2(c_2, io_2) \end{array} \right) \\ \wedge \\ \left(\begin{array}{c} (Q_1 \times Q_2)(c) \wedge \\ X_2(c_2, io_1, n_2) \wedge O_2(c_2, io_2) \wedge \\ O_1(c_1, io_1) \end{array} \right) \end{array} \right)$$

Recollapsing the definition of the transition relation T :

$$F\{Q_1 \times Q_2\} = \exists c, io. \left(\begin{array}{c} \left(\begin{array}{c} (Q_1 \times Q_2)(c) \wedge \\ T_1(c_1, io, n_1) \wedge O_2(c_2, io_2) \end{array} \right) \\ \wedge \\ \left(\begin{array}{c} (Q_1 \times Q_2)(c) \wedge \\ T_2(c_2, io, n_2) \wedge O_1(c_1, io_1) \end{array} \right) \end{array} \right) \quad (\text{Eq 4-21})$$

Eq 4-21 states that the forward execution of the composite machine is defined wholly in terms of the forward execution of the components. The separability of the two computations rests solely on the dependence of the basic transition relations X_1 and X_2 on the *whole* set of input/output variables $io = (i, o_1, o_2)$.

Another way of stating the modularity condition is that the execution of either component is completely determined by the outputs produced by the other component. The substance of Eq 4-21 is that the behavior of a component *is determined as if the outputs of its sibling were already present when it starts its reaction*. In the process algebraic notation this can be written as:

$$\langle Q_1, Q_2 \rangle \xrightarrow{I \cup O_2}^{O_1} \langle Q'_1, Q_2 \rangle \quad (\text{Eq 4-22})$$

and

$$\langle Q_1, Q_2 \rangle \xrightarrow{I \cup O_1}^{O_2} \langle Q_1, Q'_2 \rangle \quad (\text{Eq 4-23})$$

implies the composite evolves according to

$$\langle Q_1, Q_2 \rangle \xrightarrow{I}^{O_1 \cup O_2} \langle Q'_1, Q'_2 \rangle \quad (\text{Eq 4-24})$$

The modularity condition implies that a component's execution is independent of any fine structure within its sibling. Such a fine structure might include the production of the out-

puts in a specific order or allowing an output to adopt multiple values in a step.

4.1.3 Causality (C)

A semantics is causal if the outputs have the property that there is a state-dependent partial order relation $\leq_Q(i, o)$ that respects composition. The partial order describes that input $I(i)$ causes output $O(o)$ from $Q(c)$. In terms of the relational notation, the state-dependent partial order can be derived from the output relation as follows:

$$\leq_Q(i, o) = \exists c. Q(c) \wedge O(c, i, o) \quad (\text{Eq 4-25})$$

There is a $\leq_Q(i, o)$ for every state Q in the machine, each of which may be different.

That the partial order is respected across composition is the condition that the partial order for the composition $\leq_{Q_1 \times Q_2}(i, o)$ is more restrictive than any component partial order $\leq_{Q_j}(i_j, o_j)$ as follows:

$$\begin{aligned} \leq_{Q_1 \times Q_2}(i, o) &\subseteq \leq_{Q_1}(i_1, o_1) \\ \leq_{Q_1 \times Q_2}(i, o) &\subseteq \leq_{Q_2}(i_2, o_2) \end{aligned} \quad (\text{Eq 4-26})$$

An equivalent statement is that the causality relation for the aggregate $Q_1 \times Q_2$ can be derived from, and therefore is consistent with, the causality relations of each components Q_1 and Q_2 . Causality is transferred from the components to the composite in a consistent manner.

4.2 Theorem of the RMC Barrier

With these definitions the Theorem of the RMC Barrier can now be proved.

Theorem [378]

No semantics can be responsive, modular and causal without also being inconsistent.

Proof Outline

A semantics defines a class of systems. A semantics defines the possible transition relations T that can be constructed using its rules. Therefore the claim applies to an arbitrary transition relation $T(c, i, o, n)$. The proof starts from the definition of a semantics as the set of rules from which transition relations are constructed. A semantics is therefore, by definition, self-consistent and conflict free. Further, the definitions of the previous sections show that the R , M and C properties in singleton are self-consistent. The theorem is proved by showing a contradiction occurs when R , M and C appear together in the semantics.

In an RMC semantics, there must exist *at least one system* with a transition relation T that exhibits all the properties R , M and C . It is shown by a judicious construction of this system that all the properties R , M and C in the semantics *necessarily* implies a contradiction. Yet a semantics is, by definition, a contradiction-free set of rules for constructing transition relations. Therefore, since this construction can *always* be accomplished in any semantics which is RMC , it must be concluded that no semantics can have the properties R , M and C all at the same time.

Proof

Because T is responsive there exists some state Q_1 and a transition $Q_1 \xrightarrow[a]{b} Q'_1$ which on input a gives output b and no other input gives output b . Similarly, there exists a state Q_2 and a transition $Q_2 \xrightarrow[b]{a} Q'_2$. Again, no other input gives output a . This implies that the output relation contains elements:

$$\begin{aligned} Q_1(c) \wedge (\{a\} \equiv i) \wedge (\{b\} \equiv o) &\subseteq O(c, i, o) \\ Q_2(c) \wedge (\{b\} \equiv i) \wedge (\{a\} \equiv o) &\subseteq O(c, i, o) \end{aligned} \tag{RMC-1}$$

Because T is causal there exists a partial order $\leq_{Q_1}(i, o)$ and $\leq_{Q_2}(i, o)$ describing which input values cause which output values for the states Q_1 and Q_2 respectively. The

partial order contains the elements:

$$\leq_{Q_1}(i, o) = \{ (a, b) \} \quad (\text{RMC-2})$$

$$\leq_{Q_2}(i, o) = \{ (b, a) \} \quad (\text{RMC-3})$$

To demonstrate the effect on $\leq_{Q_1}(i, o)$ and $\leq_{Q_2}(i, o)$ under composition, copy the system and compose the two copies. The transition relations of the two copies of T are labeled T_1 and T_2 respectively. Let the variables for the composite be defined by $c = (c_1, c_2)$, $n = (n_1, n_2)$, a concatenation of the variables of the components. The internal communication structure is as shown in Figure 4-3 which is just a modification of Figure 4-3. There, the inputs and outputs of T_1 and T_2 are connected to each other as $o_{T_1} \equiv i_{T_2}$ and $o_{T_2} \equiv i_{T_1}$. The output of both components are concatenated to define the output of the composite $o = (o_1, o_2)$.

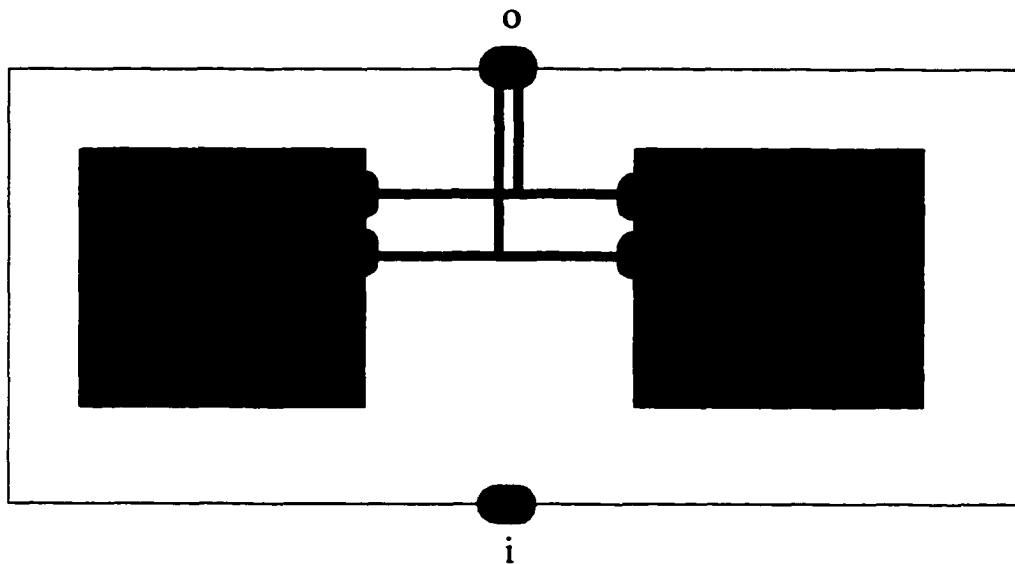


Figure 4-4. The Communication Structure of $T_1 \parallel T_2$

The state and output variables are as one might expect, concatenations of the components. As the outputs of both components define the output of the whole, there are no variables left over to become the input vector. The input variable vector for the composite is vacuous. This is denoted by defining i as the empty tuple of variables $i = ()$.¹

The transference of causality under composition is a subtle but important point and the whole of the proof rests on it. Modularity requires that each component T_1 and T_2 behave solely on the basis of its communication with its *respective* environment. In this case, the environment of each component (say T_1) is completely defined by its sibling (say T_2) and the external environment. The construction undertaken here has produced a composite $T_1 \parallel T_2$ where the component outputs are exported. By construction the composite has a simple set of inputs: none. By virtue of having a vacuous input set, the composite is said to produce outputs spontaneously with respect to its input. This must be reflected in the causality relation $\leq_{T_1 \parallel T_2}(i, o)$ for the composite.

Because T is modular the composition of T_1 and T_2 is well-defined. By construction, each component of the system contains respectively, the two transitions $Q_1 \xrightarrow{b} Q'_1$ and $Q_2 \xrightarrow{a} Q'_2$. The modularity property guarantees that there is a transition in the composite that evolves in a manner consistent with the following two relationships:

$$\langle Q_1, Q_2 \rangle \xrightarrow[\emptyset \cup \{a\}]{\{b\}} \langle Q'_1, Q_2 \rangle \quad (\text{RMC-4})$$

$$\langle Q_1, Q_2 \rangle \xrightarrow[\emptyset \cup \{b\}]{\{a\}} \langle Q'_1, Q'_2 \rangle \quad (\text{RMC-5})$$

Thus the composite evolves according to:

$$\langle Q_1, Q_2 \rangle \xrightarrow[\emptyset]{\{a, b\}} \langle Q'_1, Q'_2 \rangle \quad (\text{RMC-6})$$

In relational terms RMC-6 is written in fully expanded form as:

$$T_{\langle Q_1, Q_2 \rangle \rightarrow \langle Q'_1, Q'_2 \rangle} (c, i, o, n) = \begin{cases} (Q_1(c_1) \wedge ((\emptyset \equiv i) \wedge (\{a\} \equiv o_2)) \wedge (\{b\} \equiv o_1) \wedge Q'_1(n_1)) \\ \wedge \\ (Q_2(c_2) \wedge ((\emptyset \equiv i) \wedge (\{b\} \equiv o_1)) \wedge (\{a\} \equiv o_2) \wedge Q'_2(n_2)) \end{cases} \quad (\text{RMC-7})$$

1. The whole of the proof could be carried through with a non-vacuous definition $i = (i_\alpha, i_\beta, \dots, i_\omega)$ as well. The constitution of i is immaterial to the proof save that i is disjoint from o .

Expanding and collecting terms:

$$T_{\langle Q_1, Q_2 \rangle \rightarrow \langle Q'_1, Q'_2 \rangle}(c, i, o, n) = \left\{ \begin{array}{l} \left(\begin{array}{l} Q_1(c_1) \wedge (\emptyset \equiv i) \wedge (\{a\} \equiv o_2) \wedge Q'_1(n_1) \wedge \\ Q_1(c_1) \wedge (\emptyset \equiv i) \wedge (\{b\} \equiv o_1) \wedge \\ Q_2(c_2) \wedge (\emptyset \equiv i) \wedge (\{a\} \equiv o_2) \end{array} \right) \\ \wedge \\ \left(\begin{array}{l} Q_2(c_2) \wedge (\emptyset \equiv i) \wedge (\{b\} \equiv o_1) \wedge Q'_2(n_2) \wedge \\ Q_2(c_2) \wedge (\emptyset \equiv i) \wedge (\{a\} \equiv o_2) \wedge \\ Q_1(c_2) \wedge (\emptyset \equiv i) \wedge (\{b\} \equiv o_1) \end{array} \right) \end{array} \right\} \quad (\text{RMC-8})$$

Collapsing according to the definitions of X and O :

$$T_{\langle Q_1, Q_2 \rangle \rightarrow \langle Q'_1, Q'_2 \rangle}(c, i, o, n) = \left\{ \begin{array}{l} X_1(c_1, io_2, n_1) \wedge O_1(c_1, i, o_1) \wedge O_2(c_2, i, o_2) \\ \wedge \\ X_2(c_2, io_1, n_2) \wedge O_2(c_2, i, o_2) \wedge O_1(c_1, i, o_1) \end{array} \right\} \quad (\text{RMC-9})$$

And collapsing according to the definition of T as $T = X \wedge O$ is:

$$T_{\langle Q_1, Q_2 \rangle \rightarrow \langle Q'_1, Q'_2 \rangle}(c, i, o, n) = \left\{ \begin{array}{l} T_1(c_1, io, n_1) \wedge O_2(c_2, i, o_2) \\ \wedge \\ T_2(c_2, io, n_2) \wedge O_1(c_1, i, o_1) \end{array} \right\} \quad (\text{RMC-10})$$

RMC-10 can then be used directly in the forward image computation of Eq 4-21 as:

$$F\{Q_1 \times Q_2\} = \exists c, io. \left((Q_1 \times Q_2)(c_1, c_2) \wedge T_{\langle Q_1, Q_2 \rangle \rightarrow \langle Q'_1, Q'_2 \rangle}(c, i, o, n) \right) \quad (\text{RMC-11})$$

$$F\{Q_1 \times Q_2\} = \exists c, io. \left(\begin{array}{l} (Q_1 \times Q_2)(c_1, c_2) \wedge T_1(c_1, io, n_1) \wedge O_2(c_2, io_2) \\ \wedge \\ (Q_1 \times Q_2)(c_1, c_2) \wedge T_2(c_2, io, n_2) \wedge O_1(c_1, io_1) \end{array} \right) \quad (\text{RMC-12})$$

RMC-10 and RMC-12 shows that the composite $T_1 \parallel T_2$ makes a transition in an instant *spontaneously* with respect to its input set i . The causality relation for $Q_1 \times Q_2$ is, by these equations and the modularity property, stated as:

$$\leq_{Q_1 \times Q_2}(i, o) = \{(\emptyset, \langle b, a \rangle)\} \quad (\text{RMC-13})$$

To prove that a semantics can be all of R , M and C at the same time it now only suffices to

show that the composition $T_1 \parallel T_2$ has a partial ordering $\leq_{Q_1 \times Q_2}(i, o)$ that respects the partial orders of both the components $\leq_{Q_1}(i, o)$ and $\leq_{Q_2}(i, o)$ and also obeys RMC-13. However, this claim is contradictory.

From RMC-13 it is observed that the value $\langle b, a \rangle$ occurs on the output $o = (o_1, o_2)$. Its occurrence is, by definition, consistent with RMC-13. From RMC-2 it is observed that $o_1 \leq o_2$ which reflects that b on o_1 causes a on o_2 . From RMC-3 it is observed that $o_2 \leq o_1$ which reflects that a on o_2 causes b on o_1 . But $o_1 \neq o_2$ because the two wires are not connected.

This is a contradiction. Because this construction can be followed for any *RMC* semantics, it must be concluded that there exists *no semantics which is all of Responsive (R), Modular (M) and Causal (C)*. Q.E.D.

4.3 Microsemantics

The previous section gave a proof for the RMC Barrier Theorem in a very general setting. Specifically, the RMC Barrier Theorem is phrased in terms of the cycle-to-cycle behavior that can be described in a semantics. No mention was made in either the definitions of *R*, *M* or *C* or in the proof about any fine structure that might exist in a transition relation *T* other than that of a decomposition in terms of the output relation *O* and the pure transition relation *X*. A common fine structure one might find in a semantics is a two-level scheme where the cycle-to-cycle behavior is given in terms of an approximation by a series of smaller transitions. The upper level is called the *macrosemantics* and the lower level the *microsemantics*. The steps at the different levels are referred to respectively as macrosteps and microsteps.

4.3.1 The R , M and C Properties

A semantics can be crudely characterized in terms of which properties appear and which do not. The notation used here is that R , M or C is written if the semantics is responsive, modular or causal and \bar{R} , \bar{M} or \bar{C} if it lacks that property. So while a semantics that is fully $RM C$ is impossible, there are other possibilities such as $\bar{R}M C$, $R\bar{M}C$ and $RM\bar{C}$ with various micro-level alternatives within each characterization. Before examining how microsemantic structuring rules imply the properties R , M and C , it is worth reviewing from a practical perspective why each property is desirable in a semantics.

Responsive

The definition of responsiveness (R) states that it is possible to construct systems where the input to the system can be distinguished by examining the output. At the macrosemantic level, responsiveness is simply the Mealy machine condition, that O is given as $O(c, i, o)$. The Mealy condition implies that the output value is computed based on the outputs given by sibling components. At the microsemantic level this implies that the decision of which output value to give can be made *during* the course of a chain of microsteps, *while* the decision of about the successor state is being computed. In contrast, a semantics which is \bar{R} necessarily expresses outputs in a macrosteps in a form which is independent of other components' outputs as $O(c, o)$. As such, the decision of which output to give is conceptually made *before* the decision about the successor state is made. This is the fundamental distinction between Mealy and Moore machines and relates to the compactness of system descriptions possible in the semantics.

Modularity

The definition of modularity (M) states that the macrostep behavior of a component is dependent only on the outputs that are produced by the component's concurrent siblings. As such, the macrostep behavior of a component can be understood solely in terms of the

macrostep behavior of its siblings. In contrast, in a semantics which is \bar{M} , the macrostep behavior of the component is dependent upon something else other than the outputs produced by its siblings. Commonly, this dependence is upon the *order* in which those outputs are produced within a macro step.

Causality

The definition of causality (C) states that there is a state-dependent partial order $\leq_Q(i, o)$ among the microsteps which is also respected under concurrent composition. The partial order is an ordering among the microsteps where the macrostep behavior of a composition is simply the mutually interleaved ordered execution of the microsteps of each component. In a semantics which is C , at each microstep, the possible successor microsteps are dependent only on previously computed microsteps. For a semantics which is \bar{C} , the noncausality condition implies there exist microsteps which depend upon their own future within the macrostep.

4.3.2 Micro States and Output Variables

A two-level semantics defines the macrostep transition structure and outputs with a series of microsteps. Over this series of microsteps decisions about the successor macrostate and the output are made. There is a subtle difference between the macrostep transition relation T and the microstep transition relation T_δ . In the former case there is a qualitative difference between the state variables and the output variables. In the latter case the output variables become part of the state space of the system, though in a local and constrained way.

Eq 4-2 defined the macrostep transition relation T as the conjunction of the basic transition relation X and the output relation O . In the macro-time framework, the basic transition relation X defined the relationship between state variables c and n . The output relation O declared the relationship between states in variable c and input/output pairs

(i, o). The state variables c define the configuration of the system and are fundamental in the sense that the state completely defines the system at any given instant (hence the name). The outputs on the other hand are *associated* with the state and can be separated entirely from it.

For a series of microsteps to define a single macrotime step, there must be some conditions placed on the transitions at the micro-time level. In particular, the outputs must be consistently maintained across multiple microsteps so that the chain can be seen to aggregate into the macrostep by ignoring the details of intermediate micro-states. At the micro-time level the variables c_δ and n_δ must encompass not only the microstate but also the outputs as well. Thus in micro-time the variables c_δ and n_δ have a state component as well as an output component. This is stated as:

$$\begin{aligned} c_\delta &= (c_Q, c_O) \\ n_\delta &= (n_Q, n_O) \end{aligned} \tag{Eq 4-27}$$

In particular the microstep path between two macrotime states Q and Q' can be decomposed into a series of transitions between micro states. Some of these micro transitions contribute to the definition of the outputs and some merely contribute to the selection of the succeeding macrostate. Informally these transitions can be classified as *output-defining* or *successor-deciding*. A path from a macrostate through the microstates to another macrostate can therefore be written as:¹

1. Huizing and Gerth [378] define an elaborate process algebra style notation which expresses state-transitions $Q \rightarrow Q'$, inputs I , outputs O and the enabling condition $[E]$. Their notation for a macrostep is

$$Q \xrightarrow{I} [E] Q' \text{ and for microsteps } Q \xrightarrow{I_0}^{O_0} [E_0] q_1 \xrightarrow{I_1}^{O_1} [E_1] q_2 \xrightarrow{I_2}^{O_2} \dots \xrightarrow{I_{n-1}}^{O_{n-1}} [E_{n-1}] q_n \xrightarrow{I_n}^{O_n} Q'$$

Their process algebraic notation is left in favor of the relational μ -calculus one which relates more directly to an effective symbolic computational procedure.

$$Q \rightarrow q_{\delta_1} \xrightarrow{O_1} q_{\delta_2} \xrightarrow{O_2} q_{\delta_3} \rightarrow \dots \rightarrow q_{\delta_{k-2}} \xrightarrow{O_i} q_{\delta_{k-1}} \rightarrow q_{\delta_k} \rightarrow Q'$$

where:

$$O = \bigcup_{\delta_i} O_i$$

The two features that distinguish the various two-level semantics are the ordering of the output-defining microsteps relative to the successor-deciding microsteps and the consistency conditions $E_{\delta}^{(\Phi)}(c_{\delta})$ that hold across the microsteps. Intuitively, \bar{R} (non-responsiveness) corresponds to an ordering requirement that all output-defining microsteps can occur before all successor-deciding microsteps. The M and C properties are governed by the subtleties of the consistency conditions $E_{\delta}^{(\Phi)}(c_{\delta})$.

The condition Φ represents a policy on the enabling conditions at each δ -step. In general the policy conditions can be quite subtle even to the extreme of being path-dependent on the whole microstep path $\hat{\delta}$. The policy consistency function is written as $\Phi(\hat{\delta})$. This leads to the definition of the microstep transition relation T_{δ} in terms of two components: the basic microstep transition X_{δ} and the enabling conditions $E_{\delta}^{(\Phi)}$ that must hold for the transition to be valid. This defines the microtime transition relation as:

$$T_{\delta}(c_{\delta}, n_{\delta}) = \sum_{\delta_i} E_{\delta_i}^{(\Phi(\hat{\delta}))}(c_{\delta}) \wedge X_{\delta_i}(c_{\delta}, n_{\delta}) \quad (\text{Eq 4-28})$$

There is no microstep output relation O_{δ} analogous to the macrostep case because the output component is entailed in the state component variable definitions of Eq 4-27.

4.3.3 Microstep Paths

Within this framework a microsemantics corresponds to a choice about the policy function Φ . The function $\Phi(\hat{\delta})$ controls the internal ordering of the output-defining and successor-deciding microsteps and also the microstep-to-microstep consistency conditions in the enabling predicate $E_{\delta}^{(\Phi)}$. In the development of the examples semantics in the next

section each semantics is given by exhibiting the term $E_{\delta_i}^{(\Phi)}$ in the transition relation T_{δ} from Eq 4-28. In this sense the actual definition of Φ is incorporated in the definition of the per-microstep enabling predicate E_{δ_i} . The function Φ merely provides a recipe for the construction of E_{δ_i} for each step δ_i .

In Eq 4-28 each $E_{\delta_i}^{(\Phi)}$ defines the conditions when a particular basic microstep transition X_{δ_i} , among all the possible disjuncts of T_{δ} , is a valid microstep. The enabling conditions are phrased in terms of the output status or the output value that is present at δ_i or will be present at some δ_j on the path $\hat{\delta}$ for $i < j$. The output status is one of **undefined**, **present** or **absent** for each output according to the domain from of Figure 3-12. The output status and value is represented in the output part of the microstep state vector c_{δ} as defined in Eq 4-27.¹

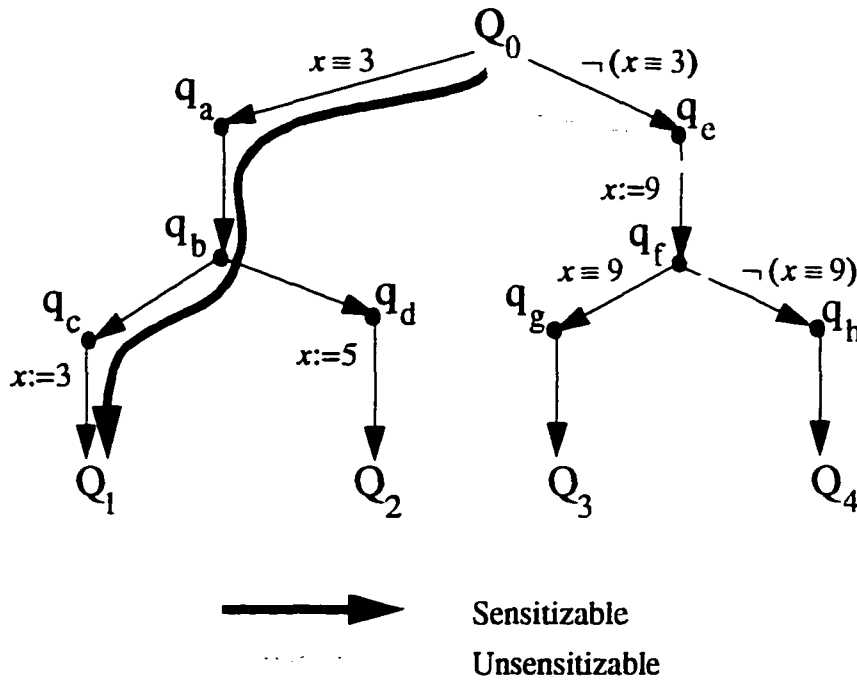
The important distinction between the status and the value is that they are two separate entities which may be specified independently of each other within microtime.² When output variables are restricted to single assignment across the microsteps of a path it may be sensible to refer to the value of an output variable *before* it has been assigned. On a given path $\hat{\delta}$ when there is only one possible value that an output variable can adopt no ambiguity can arise. This situation is illustrated in Figure 4-5. From the perspective of the macrosemantics where a whole path of transitions $\hat{\delta}$ is collapsed to a single transition

1. In an actual realization of any semantics, the decision about whether there is an explicit encoding for the states **undefined**, **present** and **absent** is idiosyncratic. It may or may not be necessary or convenient to make the elements of the domain explicit. In particular, an explicit representation for the information ordering is used in the development in Chapter 7.

On counterpoint, the traditional implementations of the example semantics described in the sequel use an implicit representation of the status. In such cases, the implementation guarantees that no output is read (or tested) before it has been written. This guarantees that the output is never referenced while in the **undefined** state.

2. This independence gives rise to output value domains consisting of a single value. In such cases only the status is of interest, that being one of **undefined**, **present** or **absent**. A prototypical example of such a case are the “pure signals” in Berry’s Esterel [79] [295].

the subtlety of the (causal) ordering of the output definition and any test of its value is abstracted. Only a single consistent output value is seen across the macrostep.



Let “ \equiv ” be an “if” test; let “ $:=$ ” be an assignment

Figure 4-5. An Output’s Status and Value are Independent

Also illustrated in Figure 4-5 is the path $Q_0 \rightarrow Q_2$ which is contradictory. It describes at Q_2 that the following must hold: $(x \equiv 3) \wedge (x \equiv 5)$. This is false for all possible values of x . The various semantics are distinguished by the conditions in $E_{\delta_i}^{(\Phi)}$ which maintain consistency across the path $\hat{\delta}$ and which constrain when an output’s value can be referenced in relation to that output’s one definition on the path $\hat{\delta}$.

The variables c_{O_k} in the output part c_O contribute to the terms of the condition $E_{\delta_i}^{(\Phi)}$. The conditions are expressed in terms of the output variable c_{O_k} referring to the value or the status. The output value predicate is the equivalence of an output O_k to some constant v as follows:

$$c_{O_k} \equiv v \tag{Exp 4-29}$$

On a more abstract level, the $E_{\delta_i}^{(\Phi)}$ may merely refer to the status aspect of the output:

$$c_{O_k} \equiv \text{present} \quad (\text{Exp 4-30})$$

$$c_{O_k} \equiv \text{absent} \quad (\text{Exp 4-31})$$

The conjunction of Exp 4-30 and Exp 4-31 is the most significant:

$$\left(c_{O_k} \equiv \text{absent} \right) \vee \left(c_{O_k} \equiv \text{present} \right)$$

This would be tautological except that there is a third alternative which is **undefined**.

Thus the expression that requires that the value of an output has been set is:

$$\neg \left(c_{O_k} \equiv \text{undefined} \right) \quad (\text{Exp 4-32})$$

Having an explicit test for **undefined** is crucial to the development which follows.

When used in an enabling condition $E_{\delta_i}^{(\Phi)}$, Exp 4-32 requires that the output c_{O_k} be defined, either **present** or **absent**, before the transition X_{δ_i} is valid. The conditions $E_{\delta_i}^{(\Phi)}$ are predicates made up of conjunctions and disjunctions of Exp 4-29, Exp 4-30, Exp 4-31 and Exp 4-32 over a subset of the outputs. The interesting output subsets are:

- All outputs O_k of all machines.
This is denoted by $\forall O_k$.
- All outputs O_k defined on microstep paths terminating at δ_i .
This set is denoted by $\forall O_k \in \overset{\rightarrow}{\delta}_{0\dots i}$
- All outputs O_k defined on microstep paths $\overset{\rightarrow}{\delta}$ which run through δ_i .
This set is denoted by $\forall O_k \in \overset{\rightarrow}{\delta}$.

4.3.4 Example Microsemantics¹

In the following sections, every microsemantics is assumed to be given by a primitive image computation $F_{\delta} \{Q\}$ and $B_{\delta} \{Q\}$ as per Eq 3-30 and Eq 3-31 with the macrose-

1. The development here follows Huizing and Gerth's [378] though, again, the notation used is the μ -Calculus instead of a process algebra. The characterization in Section 4.3.4.5 is new to this presentation.

manantics of $\bar{F}\{Q\}$ and $\bar{B}\{Q\}$ defined according to Eq 3-40 and Eq 3-41. This is relaxed in Section 4.3.4.6 when a *three-level* microsemantics is presented. Until that point, each microsemantics is distinguished solely by the form of the enabling and consistency conditions in the term $E_{\delta_i}^{(\Phi)}$ and as such can be characterized by exhibiting only the form of the terms of $E_{\delta_i}^{(\Phi)}$.

4.3.4.1 Example of \bar{RMC} (Coordinating Moore Machines)

The simplest semantics is that of a network of communicating Moore machines. The microstep policy function Φ requires that the output of all the Moore machines be defined before any machine makes a transition. Intuitively, each machine has at least two microsteps δ_o and δ_x . On δ_o the machine defines its outputs while on δ_x the machine decides its successor state. A generalization of this model to multiple output-defining microsteps δ_{o_i} , or to multiple successor-deciding microsteps δ_{x_i} is straightforward.

$$E_{\delta_{o_i}}^{(\Phi)} = \sum_{\forall O_k} (c_{O_k} \equiv \text{undefined}) \quad (\text{Ex-4.3.4.1-1})$$

$$E_{\delta_{x_i}}^{(\Phi)} = \prod_{\forall O_k} \neg(c_{O_k} \equiv \text{undefined}) \quad (\text{Ex-4.3.4.1-2})$$

These conditions state that any output-defining microsteps is enabled while any output remains undefined. Further, the successor- deciding microsteps are enabled only after all outputs of all machines are defined. This ordering is depicted in Figure 4-6.

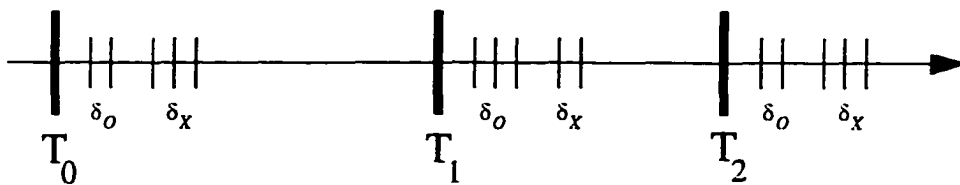


Figure 4-6. The Microstructure of Time for Example 4.3.4.1

Responsiveness

By inspection it can be observed that all output-defining microsteps are enabled and must occur before any successor-choosing microstep is enabled. By definition the input to a machine M_i is the output of one or more other machines M_j . Ex 4.3.4.1-1 shows that there is no way to refer to the output of another machine before *all* outputs are completed. Thus there is no way to produce an output pair on a given machine so that the input to that machine can be distinguished. The semantics is \bar{R} .

Modularity

The modularity condition revolves around the separability of the macrostep transition relation T into an output-defining component O and a basic transition relation X . The essence of this is summarized with respect to $F\{Q\}$ in Eq 4-21 which states that the behavior of a component is determined as if the outputs of its siblings were already present when it starts its successor-deciding transitions. Ex 4.3.4.1-2 requires that no component begins a successor-deciding transition until all components have defined their outputs. The semantics is M .

Causality

The causality condition requires that there be a state-dependent partial order $\leq_Q(i, o)$ which is respected under composition. The semantics is symmetric with respect to all possible states Q and requires that all for all components, outputs become defined before any component refers to its inputs. The $\leq_Q(i, o)$ can be given as the state-independent identity relation $\equiv(i, o)$. The semantics is C .

4.3.4.2 Example of \bar{RMC} (StateCharts #1¹)

The previous example can be extended by allowing the output-defining microsteps to

1. As per Harel *et al.* [333].

occur at any point on the path $\hat{\delta}$ but subject to the restriction that the successor-deciding microsteps to refer only to outputs that are not in the **undefined** state.

$$O_{\hat{\delta}_{0\dots i}}(c_0) = \prod_{\forall O_t \in \hat{\delta}_{0\dots i}} \neg(c_{O_t} \equiv \text{undefined}) \quad (\text{Ex-4.3.4.2-1})$$

$$E_{\hat{\delta}_i}^{(\Phi)} = O_{\hat{\delta}_{0\dots i}}(c_0) \quad (\text{Ex-4.3.4.2-2})$$

This semantics is essentially that of a network of Mealy machines with highly order-dependent coordination between the machines. As an example of this, consider the two deterministic microstep paths illustrated in Figure 4-7.

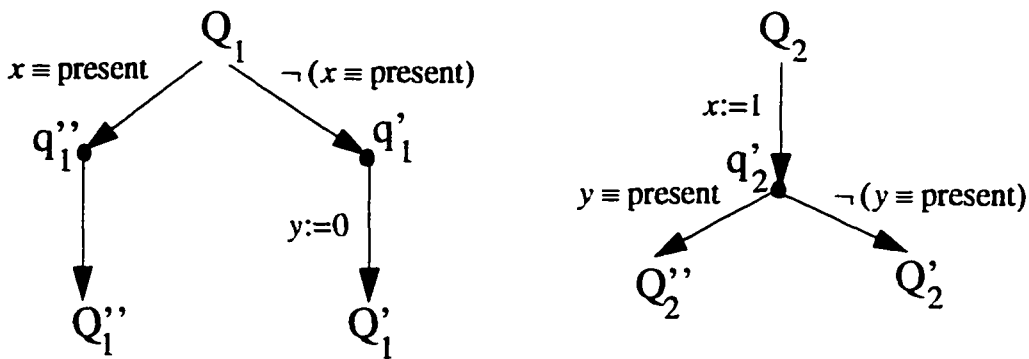


Figure 4-7. Concurrent Composition for Semantics of Example 4.3.4.2

There are three macrostep transitions possible from the concurrent composition starting at $\langle Q_1, Q_2 \rangle$. The macrostep actually selected depends on the internal ordering of the microsteps. From $\langle Q_1, Q_2 \rangle$ there are:

$$\langle Q_1, Q_2 \rangle \rightarrow \langle Q'_1, Q''_2 \rangle \quad (\text{Ex-4.3.4.2-3})$$

$$\langle Q_1, Q_2 \rangle \rightarrow \langle Q''_1, Q'_2 \rangle \quad (\text{Ex-4.3.4.2-4})$$

$$\langle Q_1, Q_2 \rangle \rightarrow \langle Q'_1, Q'_2 \rangle \quad (\text{Ex-4.3.4.2-5})$$

These macrosteps are derived according to the following chains, respectively:

$$\langle Q_1, Q_2 \rangle \rightarrow \langle q'_1, Q_2 \rangle \rightarrow \langle Q'_1, Q_2 \rangle \rightarrow \langle Q'_1, q'_2 \rangle \rightarrow \langle Q'_1, Q''_2 \rangle \quad (\text{Ex-4.3.4.2-6})$$

$$\langle Q_1, Q_2 \rangle \rightarrow \langle Q_1, q'_2 \rangle \rightarrow \langle q''_1, q'_2 \rangle \rightarrow \langle Q''_1, q'_2 \rangle \rightarrow \langle Q''_1, Q'_2 \rangle \quad (\text{Ex-4.3.4.2-7})$$

$$\langle Q_1, Q_2 \rangle \rightarrow \langle q'_1, Q_2 \rangle \rightarrow \langle q'_1, q'_2 \rangle \rightarrow \langle q'_1, Q'_2 \rangle \rightarrow \langle Q'_1, Q'_2 \rangle \quad (\text{Ex-4.3.4.2-8})$$

This ordering dependence is due to the subtlety that an output which is not **present** can be either **absent** or **undefined**. As a result, this semantics is highly nondeterministic under concurrent composition.

Responsiveness

By construction, it is possible to have an output-defining transition follow a successor-deciding transition. This means that it is possible to construct a situation where the input can be determined by examining the output. An example would be a microstep transition relation T_δ which evolved from q_1 to q_2 but defining O_2 as either c or d depending on whether the input O_1 was a or b respectively:

$$q_1 \xrightarrow[a]{c} q_2 \qquad q_1 \xrightarrow[b]{d} q_2$$

In the transition relation this is expressed as:

$$X_\delta(c_\delta, n_\delta) = \begin{cases} (c_Q \equiv q_1 \wedge c_{O_1} \equiv a) \wedge (n_Q \equiv q_2 \wedge n_{O_1} \equiv a \wedge n_{O_2} \equiv c) \\ \vee \\ (c_Q \equiv q_1 \wedge c_{O_1} \equiv b) \wedge (n_Q \equiv q_2 \wedge n_{O_1} \equiv b \wedge n_{O_2} \equiv d) \end{cases}$$

and

$$E_\delta^{(\Phi)}(c_\delta) = \neg(c_{O_1} \equiv \text{undefined})$$

The semantics is R .

Modularity

The modularity condition can be understood by examining two systems T_α and T_β which are defined at the microstep level so that T_{α_s} outputs b on input a and T_{β_s} outputs a on input b . This is illustrated in Figure 4-8 and given in transition relation form as follows:

$$T_{\alpha_s}(c_\alpha, n_\alpha) = X_{\alpha_{s_1}}(c_\alpha, n_\alpha) \wedge E_{\alpha_{s_1}}^{(\Phi)}(c_\alpha)$$

where:

$$X_{\alpha_{s_1}}(c_\alpha, n_\alpha) = (c_{Q_\alpha} \equiv q_1 \wedge c_{O_\beta} \equiv b) \wedge (n_{Q_\alpha} \equiv q'_1 \wedge n_{O_\beta} \equiv b \wedge n_{O_\alpha} \equiv a)$$

$$E_{\alpha_{s_1}}^{(\Phi)}(c_\alpha) = \neg(c_{O_\beta} \equiv \text{undefined})$$

Similarly:

$$T_{\beta_s}(c_\beta, n_\beta) = X_{\beta_{s_1}}(c_\beta, n_\beta) \wedge E_{\beta_{s_1}}^{(\Phi)}(c_\beta)$$

where:

$$X_{\beta_{s_1}}(c_\beta, n_\beta) = (c_{Q_\beta} \equiv q_2 \wedge c_{O_\alpha} \equiv a) \wedge (n_{Q_\beta} \equiv q'_2 \wedge n_{O_\alpha} \equiv a \wedge n_{O_\beta} \equiv b)$$

$$E_{\beta_{s_1}}^{(\Phi)}(c_\beta) = \neg(c_{O_\alpha} \equiv \text{undefined})$$

Consider the concurrent composition $T_\alpha \parallel T_\beta$ as defined in Eq 4-21 for the composite state $Q = (Q_1, Q_2)$. Taking $F_{T_\alpha \parallel T_\beta} \{ (Q_1, Q_2) \}$ as defined in Eq 3-40 as the fixed point of the microsteps. It can be seen by inspection from $E_{\alpha_{s_1}}^{(\Phi)}(c_\alpha)$ and $E_{\beta_{s_1}}^{(\Phi)}(c_\beta)$ that for $T_{\alpha_{s_1}}$ to be applicable, the output O_β must be defined and for $T_{\beta_{s_1}}$ to be applicable, the output O_α must be defined. It is not possible to take a forward step from (Q_1, Q_2) without assuming the definition of either output O_α or O_β . So, Eq 4-22 holds and Eq 4-23 but in conjunction they do not imply Eq 4-24. The semantics is \bar{M} .

Causality

By inspection, it can be seen that no basic microstep transition X_{δ_i} makes reference to

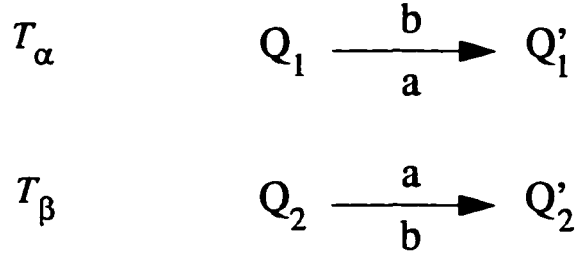


Figure 4-8. Systems T_α and T_β Showing the \bar{M} of Example 4.3.4.2
 an output until it is not **undefined**. This naturally defines a path-dependent ordering relation $\leq_{\delta}(i, o)$ between the inputs and outputs of any machine; a state-dependent ordering relation $\leq_Q(i, o)$ is derivable from $\leq_{\delta}(i, o)$ in a straightforward way by considering each start state Q of the paths $\hat{\delta}$ and deriving a relation among the $\leq_{\delta_Q}(i, o)$ which is consistent with them all:

$$\leq_Q(i, o) = \prod \leq_{\delta_Q}(i, o)$$

The semantics is C .

4.3.4.3 Example of $R\bar{M}C$ (StateCharts #2¹)

As before, the previous example can be modified by adding path-spanning consistency conditions. The previous semantics can be extended by requiring that, in addition to the enabling conditions given in Eq 4.3.4.2-2, that all microsteps on a path $\hat{\delta}$ be enabled by the output status s_k and value v_k defined for output O_k at the end of the path. The consistency conditions are as follows:

$$O_{\hat{\delta}_{0\dots i}}(c_O) = \prod_{\forall O_k \in \hat{\delta}_{0\dots i}} \neg(c_{O_k} \equiv \text{undefined}) \quad (\text{Ex-4.3.4.3-1})$$

$$O_{\hat{\delta}}(c_O) = \prod_{\forall O_k \in \hat{\delta}} (c_{O_k} \equiv v_k) \wedge (c_{O_k} \equiv s_k) \quad (\text{Ex-4.3.4.3-2})$$

1. As per Pnueli and Shalev [602].

$$E_{\delta_i}^{(\Phi)} = O_{\delta_{0..i}}(c_O) \wedge O_{\delta_i}(c_O) \quad (\text{Ex-4.3.4.3-3})$$

This semantics is like the Mealy network of the previous section only there is a consistency constraint on each output O_k across all the microsteps of a path $\hat{\delta}$. The consistency constraints have the effect of allowing only the transition of Ex 4.3.4.2-4 by the consistent path given in Ex 4.3.4.2-7.

The path-spanning consistency constraint makes for a deterministic semantics, but still at the expense of M . In aggregate, this semantics has exactly the same properties, \overline{RMC} , as the semantics of Section 4.3.4.2. The extra conditions of Ex 4.3.4.3-2 merely reduce the amount of intra-macrostep nondeterminism. In particular the proofs of the previous section can be carried through directly with Ex 4.3.4.3-3 substituted instead of Ex 4.3.4.2-2. for the definition of $E_{\delta}^{(\Phi)}(c_{\delta})$.

Responsiveness

The semantics is R , as per Example 4.3.4.2.

Modularity

The semantics is \overline{M} , as per Example 4.3.4.2.

Causality

The semantics is C , as per Example 4.3.4.2.

4.3.4.4 Example of \overline{RMC} (The Synchronous Languages¹)

Again modifying the previous example, a new semantics can be defined removing the

1. Halbwachs [320] and the series of articles in the special issue edited by Benveniste and Berry [62] (the articles [101] [195] [321] [465] [469]) are good overviews.

The Synchronous Languages are: Esterel [75] [295], Lustre [153] [321] [320], Signal [464] [65] [465] and Argos [501] [502] [503].

Also, see the presentation of Section 5.7.

local causality conditions of Ex 4.3.4.3-1 but keeping the path-spanning consistency conditions of Ex 4.3.4.3-2. The consistency conditions are thus as follows:

$$O_{\hat{\delta}}(c_O) = \prod_{\forall O_k \in \hat{\delta}} (c_{O_k} \equiv v_k) \wedge (c_{O_k} \equiv s_k) \quad (\text{Ex-4.3.4.4-1})$$

$$E_{\delta_i}^{(\Phi)} = O_{\hat{\delta}}(c_O) \quad (\text{Ex-4.3.4.4-2})$$

This semantics keeps the deterministic aspect of Example 4.3.4.3 but does not require that transitions possible at δ_i be enabled by the outputs made available up to that point. Transitions at δ_i are enabled by outputs defined in any microstep δ_j for $j \in 0 \dots n$ on the full path $\hat{\delta} = \delta_{0 \dots n}$.

Responsiveness

As in the previous examples, because an output-defining microstep can appear after a successor-deciding microstep, the semantics is R . The formal of this would be exactly as in Example 4.3.4.2.

Modularity

The proof of the modularity property follows from the example used to show that Example 4.3.4.2 is \bar{M} . In this example, the construction shows that the current semantics is M . As before, consider two systems T_α and T_β which are defined at the microstep level so that T_{α_s} outputs b on input a and T_{β_s} outputs a on input b as follows:

$$T_{\alpha_s}(c_\alpha, n_\alpha) = X_{\alpha_{s_i}}(c_\alpha, n_\alpha) \wedge E_{\alpha_{s_i}}^{(\Phi)}(c_\alpha)$$

where:

$$X_{\alpha_{s_i}}(c_\alpha, n_\alpha) = (c_{O_\alpha} \equiv q_1 \wedge c_{O_\beta} \equiv b) \wedge (n_{O_\alpha} \equiv q'_1 \wedge n_{O_\beta} \equiv b \wedge n_{O_\alpha} \equiv a)$$

$$E_{\alpha_{s_i}}^{(\Phi)}(c_\alpha) = (c_{O_\alpha} \equiv a \wedge c_{O_\alpha} \equiv \text{present}) \wedge (c_{O_\beta} \equiv b \wedge c_{O_\beta} \equiv \text{present})$$

Similarly:

$$T_{\beta_s}(c_\beta, n_\beta) = X_{\beta_{s_1}}(c_\beta, n_\beta) \wedge E_{\beta_{s_1}}^{(\Phi)}(c_\beta)$$

where:

$$X_{\beta_{s_1}}(c_\beta, n_\beta) = (c_{Q_\beta} \equiv q_2 \wedge c_{O_\alpha} \equiv a) \wedge (n_{Q_\beta} \equiv q'_2 \wedge n_{O_\alpha} \equiv a \wedge n_{O_\beta} \equiv b)$$

$$E_{\beta_{s_1}}^{(\Phi)}(c_\beta) = (c_{O_\alpha} \equiv a \wedge c_{O_\alpha} \equiv \text{present}) \wedge (c_{O_\beta} \equiv b \wedge c_{O_\beta} \equiv \text{present})$$

Again, consider the concurrent composition $T_\alpha \parallel T_\beta$. Here it is possible to take a forward step from (Q_1, Q_2) without assuming the definition of either output O_α or O_β (the assumption of both occurs *within* the microstep sequence). As a consequence, Eq 4-22 holds, Eq 4-23 holds and in conjunction they imply Eq 4-24. The semantics is M .

Causality

This very same example can be used to demonstrate the causality property. In T_α the state-dependent partial order from Q_1 is

$$\leq_{Q_1}(O_\beta, O_\alpha) = \{(a, b)\}$$

In T_β the state-dependent partial order from Q_2 is

$$\leq_{Q_2}(O_\alpha, O_\beta) = \{(b, a)\}$$

The principle of causality requires that the state-dependent partial order for the concurrent composition $\leq_{Q_1 \times Q_2}(I, O)$ be consistent with both $\leq_{Q_1}(O_\beta, O_\alpha)$ and $\leq_{Q_2}(O_\alpha, O_\beta)$. For the composite, as in the example constructed for RMC-13, the inputs and outputs are defined as:

$$I = ()$$

$$O = (O_\alpha, O_\beta)$$

and as with that example,

$$\leq_{Q_1 \times Q_2}(I, O) = \{(\emptyset, \langle b, a \rangle)\}$$

Yet, by construction $O_\beta \leq O_\alpha$ and $O_\alpha \leq O_\beta$ but $O_\alpha \neq O_\beta$. The semantics is \bar{C} .

4.3.4.5 Example of \bar{RMC} (Codesign Finite State Machines¹)

The previous examples have approached the semantics definition problem from the viewpoint that a microsemantics is a decomposition of a macrosemantics. It is possible to adopt the opposite viewpoint, namely that a macrosemantics is a summarization of a microsemantics. The example of this section demonstrates that the RMC Barrier applies equally to such constructions.

Here the fundamental level of discourse are microsteps: the microstep transition relation T_δ is the primary entity with the macrostep transition relation T being derived from it. The behaviors allowed in the (micro)semantics then are the set of possible chains of transitions in T_δ . Call any chain of transitions in T_δ a *trace*, specifically a *prefix-closed trace* having the property that $\forall t \in T. \exists s \in T. \exists s' \in \Sigma^*. t = ss'$. Within such a trace, consider two sorts of transitions $t_{\delta_i} \in T_\delta$ as before: successor-deciding transitions and output-defining transitions. The successor-deciding transitions do not define outputs while the output-defining transitions define a value for an output.

In order to break up a trace of microsteps into a series of macrosteps, a means for identifying the end of a macrostep must be defined. Call a set of successor-deciding microsteps a *cause* and call a set of output-defining microsteps a *reaction* with causes being labeled C_i and reactions being labeled R_i . Without loss of generality, assume that for all macrosteps i that $|C_i| > 0$ and $|R_i| > 0$. This makes it meaningful to speak of the *range* of C_i or R_i as the pair $(\delta_{min}, \delta_{max})$ which are the minimum and maximum micro-time points in the set. A macrosemantics is a projection of a microstep path $\hat{\delta}$ down onto a macrostep path $\vec{\sigma}$ of causes-reaction pairs $\sigma_i = (C_i, R_i)$ subject to the following gen-

1. As per Chiodo *et al.* [167]. Also see the presentation of Section 5.4.

eral conditions:¹

1. All cause sets C_i and C_j are disjoint: $C_i \cap C_j = \emptyset$.
2. Any two cause sets C_i and C_j may overlap in microtime; as is illustrated in Figure 4-9.
3. Reaction sets R_i and R_j , $i \neq j$ may *not* overlap in microtime; thus they must be disjoint.
4. Without loss of generality, in any macrostep $\sigma_i = (C_i, R_i)$, all the causes $c \in C_i$ must all occur before any of the reactions $r \in R_i$; the C_i and R_i may not overlap.

Subject to these rules the semantics of a composition $T_{1_s} \parallel T_{2_s}$ is defined again in terms of microstep traces. The traces admitted by the composition is the set of traces in the set of shufflings of traces² of T_{1_s} and T_{2_s} that are consistent with the interconnections between T_{1_s} and T_{2_s} . This is an existential condition on composition, that there *exists* a projection of the microstep trace $\vec{\delta}_{T_{1_s} \parallel T_{2_s}}$ that is a macrostep trace $\vec{\sigma}_{T_{1_s} \parallel T_{2_s}}$ which is consistent with consistency and ordering rules. For the semantics to be M then such a projection must always exist.

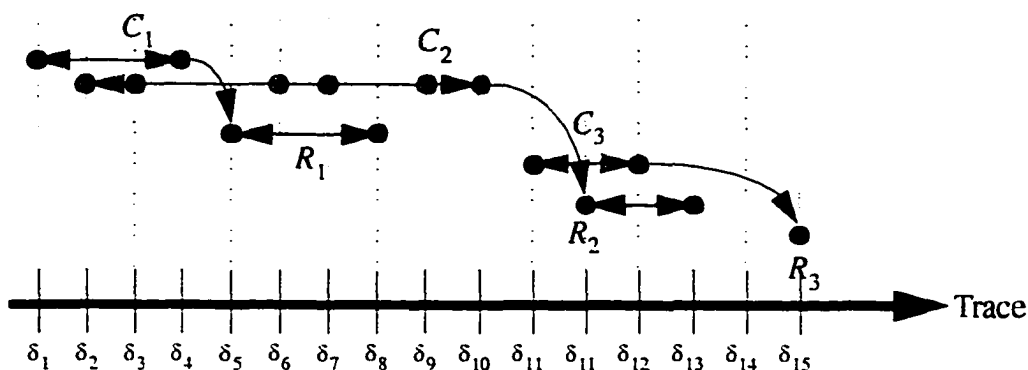


Figure 4-9. A Semantics of Microsteps defining Macrosteps

1. What is significant in this formulation is that a macrostep transition need *not* be a fixed point on the raw microstep trace $\vec{\delta}$ directly. It is not possible to characterize a macrostep trace $\vec{\sigma}$ as a chain of fixed points in T_s since the ranges of cause sets C_i can overlap.
2. Trace shufflings are used in the trace-based semantics of NES, *c.f.* Section 2.3.1.

The view adopted in this semantics is still that of a reactive system. Causes still generate reactions and the behavior of the system is defined as the set of possible causes and reactions. There are new degrees of freedoms in this semantics that were not present in the previous examples. The most notable degree of freedom is the ability to overlap the ranges of one ore more causes. This expressiveness can be understood as the ability of the system to observe multiple triggering conditions before reacting to either. The second degree of freedom is the ability to skew various causation conditions and reactions across time. This is freedom allows for pipelining effects to be described in the semantics.

Responsiveness

In all macrosteps σ_i , the cause C_i must precede the reaction R_i . The semantics is R by construction.

Modularity

The modularity property of the semantics can be understood by examining two systems much like the ones used in Example 4.3.4.2. The two systems in the form of their microstep traces are as shown in Figure 4-9.

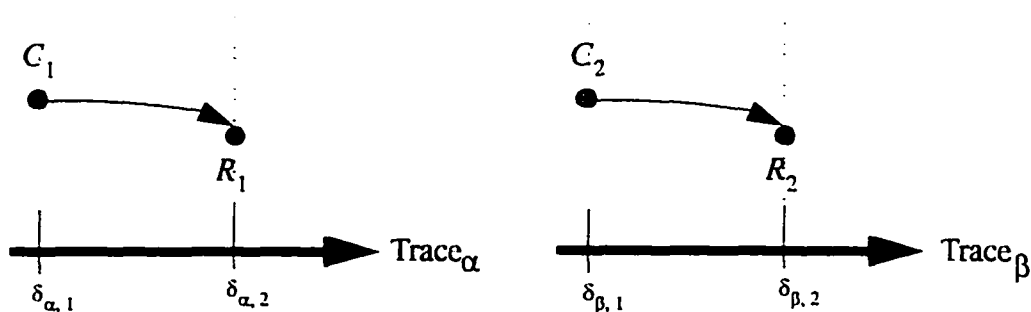


Figure 4-10. Two Microstep Traces for Composition in Example 4.3.4.5

In the case of this example modularity must be shown by the composition of traces rather than the conjunction of microstep transitions relations. To show modularity, the *existence* of a valid macrostep trace for the composition must be exhibited.

Consistent with Example 4.3.4.2, take of $\delta_{\alpha,1}$, $\delta_{\alpha,2}$, $\delta_{\beta,1}$, and $\delta_{\beta,2}$ to be as follows:

$$\delta_{\alpha,1} = Q_1 \xrightarrow{a} q_1 \quad (\text{Ex-4.3.4.5-1})$$

$$\delta_{\alpha,2} = q_1 \xrightarrow{b} Q'_1$$

$$\delta_{\beta,1} = Q_2 \xrightarrow{b} q_2 \quad (\text{Ex-4.3.4.5-2})$$

$$\delta_{\beta,2} = q_2 \xrightarrow{a} Q'_2$$

According to the structure of Figure 4-9 the (unit-length) macrostep traces $\vec{\sigma}_\alpha$ and $\vec{\sigma}_\beta$ are given by the causes C_1 and C_2 , and reactions R_1 and R_2 as follows:

$$\vec{\sigma}_\alpha = (C_1, R_1) = \left(\{O_\beta \equiv a\}, \{O_\alpha \equiv b\} \right) \quad (\text{Ex-4.3.4.5-3})$$

$$\vec{\sigma}_\beta = (C_2, R_2) = \left(\{O_\alpha \equiv b\}, \{O_\beta \equiv a\} \right) \quad (\text{Ex-4.3.4.5-4})$$

Since the macrostep traces $\vec{\sigma}_\alpha$ and $\vec{\sigma}_\beta$ are only one macrostep long, the state-dependent causal partial orders for T_α and T_β can be observed directly:

$$\leq_{Q_1} (O_\beta, O_\alpha) = \{ (a, b) \} \quad (\text{Ex-4.3.4.5-5})$$

$$\leq_{Q_2} (O_\alpha, O_\beta) = \{ (b, a) \} \quad (\text{Ex-4.3.4.5-6})$$

In terms of the microsteps these partial orders can be rewritten as:

$$\leq_{Q_1} (O_\beta, O_\alpha) = \{ (\delta_{\alpha,1}, \delta_{\alpha,2}) \} \quad (\text{Ex-4.3.4.5-7})$$

$$\leq_{Q_2} (O_\alpha, O_\beta) = \{ (\delta_{\beta,1}, \delta_{\beta,2}) \} \quad (\text{Ex-4.3.4.5-8})$$

In the composition $T_{\alpha_s} \parallel T_{\beta_s}$ it must also be the case that the output-defining microsteps must precede corresponding successor-deciding microsteps (input accepting microsteps).

This causality requirement is given in terms of an ordering on microsteps as:

$$\{ (\delta_{\alpha,2}, \delta_{\beta,1}), (\delta_{\beta,2}, \delta_{\alpha,1}) \} \subseteq \leq_{Q_1 \times Q_2} (I, O_{\alpha\beta}) \quad (\text{Ex-4.3.4.5-9})$$

The unit-length macrostep trace of the composition $\vec{\sigma}_{T_{\alpha_s} \parallel T_{\beta_s}}$ consists of a cause $C_{T_{\alpha_s} \parallel T_{\beta_s}}$

and a reaction $R_{T_{\alpha_s} \parallel T_{\beta_s}}$ which has the property that every output-defining microstep $r \in R_{T_{\alpha_s} \parallel T_{\beta_s}}$ occur after every successor-deciding microstep $c \in C_{T_{\alpha_s} \parallel T_{\beta_s}}$. This requires that, in addition to Ex 4.3.4.5-7 and Ex 4.3.4.5-8 that the following be true:

$$\{(\delta_{\alpha,1}, \delta_{\beta,2}), (\delta_{\beta,1}, \delta_{\alpha,2})\} \subseteq \leq_{Q_1 \times Q_2} (I, O_{\alpha\beta}) \quad (\text{Ex-4.3.4.5-10})$$

Examining Ex 4.3.4.5-9 and Ex 4.3.4.5-10 shows that any valid macrostep trace $\vec{\sigma}_{T_{\alpha_s} \parallel T_{\beta_s}}$ must be the projection down from a microstep trace $\vec{\delta}_{T_{\alpha_s} \parallel T_{\beta_s}}$ that has the following internal orderings:

$$\delta_{\alpha,1} \leq \delta_{\beta,2} \wedge \delta_{\beta,2} \leq \delta_{\alpha,1} \quad (\text{Ex-4.3.4.5-11})$$

$$\delta_{\beta,1} \leq \delta_{\alpha,2} \wedge \delta_{\alpha,2} \leq \delta_{\beta,1} \quad (\text{Ex-4.3.4.5-12})$$

Yet, by definition $\delta_{\alpha,1} \neq \delta_{\beta,2}$ and $\delta_{\beta,1} \neq \delta_{\alpha,2}$. Thus there exists no valid microstep trace $\vec{\delta}_{T_{\alpha_s} \parallel T_{\beta_s}}$. As such there exists no valid macrostep trace $\vec{\sigma}_{T_{\alpha_s} \parallel T_{\beta_s}}$. The semantics is \bar{M} .

Causality

The behavior of compositions is the interleaving of their microstep traces $\vec{\delta}_{T_{\alpha_s}}$ and $\vec{\delta}_{T_{\beta_s}}$ subject to the usual causal ordering of each. The composition $T_{\alpha_s} \parallel T_{\beta_s}$ is by definition subject to the condition that $\vec{\delta}_{T_{\alpha_s} \parallel T_{\beta_s}}$ has a consistent projection $\vec{\sigma}_{T_{\alpha_s} \parallel T_{\beta_s}}$ where cause sets C_i precede reaction sets R_i . The semantics is C by construction.

4.3.4.6 Example of \bar{RMC} (StateCharts #3¹, Discrete Event Semantics²)

The final example semantics is a three-level semantics. Here, the three levels of time are called macrotime, microtime and nanotime. The three levels of time are depicted in Figure 4-11. The intuitive idea of the semantics is that a reaction at the microtime level is composed of a series of nanosteps and a reaction at the macrotime level is composed of a

1. As per i-Logix Inc. [383].
2. See Section 6.2.

series of microsteps. The output asserted by the macrostep is the union of the outputs produced in all the microsteps δ_i :

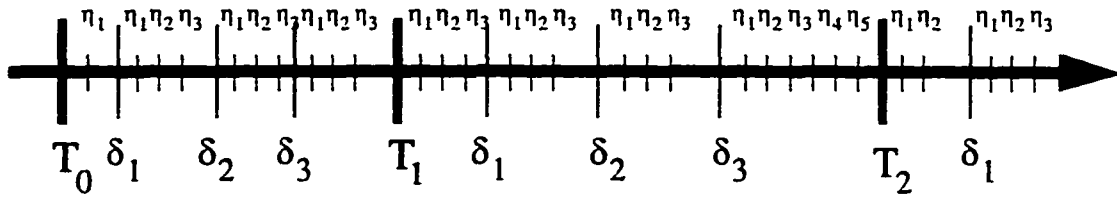


Figure 4-11. The Three-Level Model of Time for Example 4.3.4.6

$$O = \bigcup_{\forall \delta_i \in \delta} O_{\delta_i} \quad (\text{Ex-4.3.4.6-1})$$

Transitions at the nanotime level are structured so that all output-defining transitions occur after all successor-deciding transitions which means that outputs defined during a microstep do not become available until the next microstep. This case is the same as the one in Example 4.3.4.1 except that in this case, the \bar{R} property holds over microsteps. Outputs events produced in a microstep δ_i become visible only during δ_{i+1} and are thereafter unseen for $j > i + 1$. The output values of course are persistent in the future δ_j . This property is called \bar{R}_δ because \bar{R} holds over δ -steps.

What makes the \bar{R}_δ semantics interesting is that the interfaces between the two levels are *almost* characterized by the fixed point of approximator functionals in the style of Eq 3-40 and Eq 3-41, yet those equations do not apply. The strong difference is that in this case, outputs which become defined in microstep δ_j are visible only the next microstep, δ_{j+1} . This prevents the characterization of microtime as a fixed point over nanosteps. The second strong difference is that output (events) become available for a microstep and then disappear thereafter. The ability of an output to be both **present** and **absent** within a macrostep implies that $F_\delta\{Q\}$ and $B_\delta\{Q\}$ are not monotonic with respect to the output information status as has been the case in previous examples.

That $F_\delta\{Q\}$ and $B_\delta\{Q\}$ are computable functions can be established by empirical observation. Therefore, by Scott's theory, the image computations must be monotonic and continuous over *some* information measure in Q . This information measure however is thought to be too horrendously complex to be usefully characterized.¹ Because $F_\delta\{Q\}$ and $B_\delta\{Q\}$ are not monotonic with respect to outputs O_k , they are not guaranteed to have a least fixed point relative to the outputs O_k . On counterpoint, $F_\eta\{Q\}$ and $B_\eta\{Q\}$ are trivially monotonic in the output information status because output definition actually occurs but once and at the start of the successive δ -steps. Output definition occurs *after* the η -step fixed point of the previous δ -step has been reached. Their fixed point is trivially reached after all steps triggered at the start of the δ -step have been taken.

Responsiveness

It is possible to have an output-defining nanostep following a successor-deciding nanostep, albeit across separate microsteps. Although output-defining nanosteps are required to precede successor-deciding nanosteps within a single microstep, if two adjacent microsteps δ_i and δ_{i+1} are considered, it is clear that the output-defining nanosteps of δ_{i+1} occur after the successor-deciding nanosteps of microstep δ_i . The semantics is R .

Modularity

The semantics allows outputs to appear and disappear across the microtime steps. Whether an output from one component is sensed by another component is completely

1. One of the advantages of the process algebraic notation is that such non-monotonic transition systems can be *expressed* in a convenient and compact notation. The advantage of the μ -calculus is that it provides an expressive notation which is also effective computationally. The μ -calculus however requires formal monotonicity of functionals to ensure that the greatest and least fixed points exist.

The first attempt at formalizing the semantics of StateCharts [333] demonstrated the non-monotonicity of the StateChart's macrostep transition relation. This failure led to further refinements of the semantics [602] and ultimately to the definition of the Argos [501] variant of StateCharts and the RMC Barrier Theorem [378].

dependent upon the microstep in which the output is sensed. Further, dependent upon this ordering, it is possible for the multiple **presence** and **absence** status of an output across microsteps to enable behavior in a sibling. The semantics is \bar{M} .

Causality

The semantics is \bar{C} by inspection. In particular, it is possible to have unbounded sequences of microsteps δ_i .

4.4 Beyond the RMC Barrier

A number of strategies have been developed to approach the RMC Barrier. Interestingly it *is possible* to move beyond the RMC Barrier, though at some cost of expressiveness elsewhere in the semantics. There are a number of avenues available, each involving some sort of trade-off. Among these avenues, it should be noted that the RMC Barrier applies to

- a semantics as a class of systems, but not to individual systems within the class,
- a single semantics considered as a unit, but not to multiple separated semantics,
- a semantics in full, but not to semantics restricted by structural or reachability analysis,

By conscious design and by accidental evolution these aspects have been exploited to approach and even move beyond the RMC Barrier. Five strategies can be distinguished:

1. The Two-of-Three Choice

A choice among \bar{RMC} , $R\bar{MC}$ and $RM\bar{C}$ is made and justified.

2. Separated Semantics

The whole semantics is given in two parts, one for component-building and one for aggregation: a different two-of-three choice is made for each part.

3. Structural Restriction to RMC

A structural restriction is made on the system-building primitives so that RMC always holds. While the semantics is internally contradictory by the RMC Barrier Theorem, no contradiction can arise from the kinds of structural compositions admitted.

4. Semantic Restriction to RMC

A semantic restriction is made so that RMC always holds. Again, the semantics has RMC and is internally contradictory by the theorem, but a reachability analysis is invoked to disallow contradictory system descriptions.

5. Vacated Semantics

Here the model has RMC but the semantics simply ignores any contradictory interpretations. The semantics “vacates” the problematic denotations. They just don’t count.

The following sections describe each of these five approaches in greater depth explaining the purpose and effect of each approach. This analysis and the survey of semantic models from Chapter 2 form the basis of the survey and classification of applied semantics given in Chapter 5.

4.4.1 The Two-of-Three Choice

The RMC Barrier states that of the three properties R , M and C , only two can appear in any semantics; adding the third causes a contradiction. The two-of-three strategy is merely the admission that a choice must be made: two primary properties are named and the third is disallowed. In a strong sense the two-of-three choice forces the language designer’s philosophy on the programmer/designer. In the best of cases, this choice is motivated by some technique that becomes possible if the restriction is adopted. In less fortunate circumstances, the choice is one of convenience. Examples of the Two-of-Three Choice can be found for all three possible two-out-of-three choices:

- For $\bar{R}MC$ the CSML language [123] can be cited. It was designed with an eye towards compositional verification algorithms based on reduction.
- For $R\bar{M}C$ the modified semantics of Pnueli and Shalev [602] for the StateCharts formalism [330] can be cited. It was designed with an eye towards establishing a deterministic and causal semantics for the StateCharts semantics and improved on earlier definitions which did not have these properties [333].
- For $RM\bar{C}$ the branching time temporal logic CTL [184] [249] can be cited. It was designed with an eye towards expressing future existential possibilities.¹

The One-of-Three Choice

That two out of three of R , M and C can be chosen naturally leads to the question of whether it is meaningful to choose but one of the three.

- For \overline{RMC} the original semantics of StateCharts [330] can be cited. It was designed with an eye towards ensuring that any syntactically legal chart had a semantic interpretation.
- For $\overline{R}MC$ and $R\overline{M}\overline{C}$ there does not seem to be any obvious examples. While such could be artificially constructed, the point here is to cite existing schemes.

The Zero-of-Three Choice

The remaining choice combination is Zero-of-Three, a semantics that is $\overline{R}\overline{M}\overline{C}$. It can be observed that all of the properties R , M and C relate to how inputs and outputs are treated in the semantics. A semantics that admits neither inputs or outputs can be understood as being $\overline{R}\overline{M}\overline{C}$. Clearly such a semantics is not R as an input-output pair that differ cannot be exhibited; it is not M because the evolution of a concurrent combination is not based on the outputs of the siblings; it is not C because no partial order $\leq(i, o)$ exists. A pure Kripke structure which consists of $K = (Q, T, L)$ is an $\overline{R}\overline{M}\overline{C}$ semantics.

Another example of an $\overline{R}\overline{M}\overline{C}$ semantics can be seen in the so-called asynchronous shared memory model of concurrent computation. In that model a system is composed of some number of concurrent processes and a shared global memory. Each process evolves according to its own state graph with interprocess communication being accomplished exclusively through the global shared memory. The model of concurrency is fundamentally one of interleaving where at any time point, only one process makes a forward step. The total behavior of the system is the arbitrary interleaving of all the process' actions. Practical instances of $\overline{R}\overline{M}\overline{C}$ semantics include Pnueli's Temporal Logic languages [599] [601], Chandy and Misra's UNITY language [159] or more recently Dill *et al.*'s Murφ

1. Recall that a noncausal system is one that anticipates its own future. A semantics that gives meaning to statements such as EXP (there exists a next state where p holds) is clearly one which anticipates the future. A non-causal system is not computationally effective in the real world where time only moves forward.

language [239].¹

4.4.2 Separated Semantics

If a choice had to be made, Huizing and Gerth's proposal for moving beyond the RMC Barrier was to split the semantics for system description into two levels. Intuitively there would be a "high level" used to aggregate components and a "low level" used to build the individual components. A different two-of-three choice is used on the different levels. Their specific proposal was to use \overline{RMC} for the low-level bodies and to aggregate these component bodies using an $RM\overline{C}$ semantics. They give the semantics of a *module* operator that hides internal communications and ensures that the semantics is M . The hiding operator has the effect of allowing outputs to occur spontaneously with respect to the module's inputs. This has the effect of giving the upper-level semantics the \overline{C} property.

4.4.3 Static Restriction to RMC

The two previous strategies involved accepting the RMC Barrier and living within its constraints. However, the RMC Barrier applies to semantics, whole classes of systems, but is silent on the issue of individual systems. This leaves open the opportunity to admit individual instances of a possibly inconsistent semantics. Thus, RMC can be guaranteed to hold on any admissible system. The simplest way to ensure that RMC holds on a system is to use a system aggregation scheme that is guaranteed to preserve two of the three properties. That the third property holds is guaranteed by some sort of structural restriction on the constructions that are allowed.

The most common instance of this strategy starts with an underlying semantics where concurrent composition preserves R and M but does not necessarily preserve C . The

1. This is not to say that \overline{RMC} semantics are not used to model reactive systems [332], rather that the focus is on the atomic and interleaved aspects of concurrency.

structural restriction ensures that C always holds. For C to hold, there must be a partial order $\leq(i, o)$ which respects modularity and a simple way to ensure that such a partial order holds is to allow only system descriptions where such an ordering can be constructed from the communication graph of the system's components.

The communication graph is an abstraction of the communication among concurrent components. It is a directed graph where the vertices P_i are the autonomous unit of execution¹ and there is an edge from P_i to P_j if the output of P_i is an input to P_j .² For the class of finite-state hardware and embedded software systems the communication graph is considered to be a static abstraction of the structure of the system description. The restriction placed on the communication graph is that it must be acyclic. Having an acyclic communication graph ensures that $\leq(i, o)$ exists for it can be constructed from the structure of the system, independent of the semantics. Examples where this approach have been taken are the languages SMV [518], Lustre [153] or S/R [424] [454]. In these cases the legal system descriptions are those where there is a topological order on, respectively, the variable definitions or processes.

4.4.4 Semantic Restriction to RMC

The natural extension of static analyses to guarantee RMC are semantic analyses to guarantee the property. As with the examples of the previous section, the typical case is to start with a description scheme which is RMC . The admitted system descriptions are a subset of these where RMC holds. Whereas in the previous case, the structural aspects of the description were used to verify C , in dynamic analysis the check is based on the feasible behaviors of the system. As such dynamic analysis for C is intrinsically related to

1. The clumsy term "autonomous unit of computation" is used here as a generalization of the various terms used in various languages: process, module, machine, thread, function.

2. Communication graphs have been defined in a number of places. One recent exposition can be found in Aziz, Tasiran and Brayton's heuristics for OBDD variable ordering [43].

reachability analysis. Checking that the C property holds is generally referred to as *causality checking*. Examples of languages where semantics-based causality checking is used are the Synchronous languages which are reviewed in Section 5.7.

Concretely, a dynamic analysis must ensure that the state-dependent partial order for a concurrent combination, $\leq_{q_1 \times q_2}(i, o)$, is consistently defined for all possible reachable states $(q_1, q_2) \subseteq Q_1 \times Q_2$ of the system. In the worst case there are $O(|Q_1||Q_2|)$ checks which must be performed. Each check ensures that $\leq_{q_1 \times q_2}(i, o)$ is consistent with both $\leq_{q_1}(i, o)$ and $\leq_{q_2}(i, o)$. A deeper presentation of static causality checking is deferred until Chapter 7.

4.4.5 Vacated Semantics

The final strategy for attaining *RMC* on a system description is called here the *vacated semantics*.¹ In that approach the inconsistencies inherent in an *RMC* semantics are not in the set of behaviors admitted by the semantics. So while the structure of the system description could be interpreted to exhibit a contradiction, the semantics ignores them: the contradictions just don't count. An example of a vacated semantics can be seen in the Combinational/Sequential model [364] [358] and the BLIF-MV [107] representation of logic networks for formal verification algorithms [37].

In BLIF-MV, all primitive semantics is given by means of transition relations which are called *tables*. The BLIF-MV tables are generalizations of truth tables in the sense that they can declare nondeterministic outputs for a gate. Combinational logic is described by

1. This name is derived from the legal term that describes a court of law adopting a new doctrine, thereby repudiating the old: a court is said to "vacate" a previous ruling. That is exactly the sense here.

means of composing tables through shared variables which model interconnecting wires. The transition relations for the basic AND, OR and NOT gates are shown in Figure 4-12.

The semantics of a network of gates is determined from a composition of the tables for the individual gates according to the formula:

$$T_{A \parallel B}(x, y) = T_A(x_A, y_A) \wedge T_B(x_B, y_B) \quad (\text{BLIF-MV-1})$$

This expresses how $A \parallel B$ evolves in one step. In particular with the output of A connected to the input of B , existential quantification is used to declare that the output of A has the same value as the input of B :

$$T_{B \circ A}(x, y) = \exists z. (T_A(x, z) \wedge T_B(z, y)) \quad (\text{BLIF-MV-2})$$

This leads to the interesting paradox of the circuit shown in Figure 4-13.¹ By inspection it can be seen that the circuit implements $y = x$ for all possible the values of x . That is, the circuit has the behavior that $x \equiv 0 \Rightarrow y \equiv 0$ and $x \equiv 1 \Rightarrow y \equiv 1$. This behavior can be observed by focusing on the AND gate labeled D. Clearly $x \equiv 0 \Rightarrow y \equiv 0$. The paradox is that the BLIF-MV composition semantics only admits the behavior where $x \equiv 1$. The behavior when $x \equiv 0$ is simply ignored.

This circuit is distinguished by having an internal combinational feedback path. The feedback is of a special kind in the sense that the local structure of the circuit is contradictory however the global function of the circuit never allows this inconsistency to be observed. Malik has formalized this notion of local structural inconsistency versus global functional consistency to define the dynamic causality checking problem in the general case [494] [495]. In practice, such examples might arise from a synthesis or translation procedure which constructs circuit descriptions that contain feedback loops [242] [76]

1. This particular example was provided by Tom Shiple in the context of dynamic causality checking. The observations about the circuit under Combinational/Sequential semantics are due to Sharad Malik. This effect was independently observed by Burch *et al.* [142].



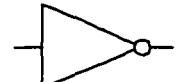
Gate	Function	Table
AND 	$z = x \wedge y$	$T_{AND} = \begin{bmatrix} x & y & z \\ 0 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}$
OR 	$z = x \vee y$	$T_{OR} = \begin{bmatrix} x & y & z \\ 0 & 0 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix}$
NOT 	$z = \neg x$	$T_{NOT} = \begin{bmatrix} x & z \\ 0 & 1 \\ 1 & 0 \end{bmatrix}$

Figure 4-12. BLIF-MV Tables for the Common Gates

[78]. A more detailed presentation of dynamic causality checking is given in Chapter 7. The relevance here is that the vacated semantics of the Combinational/Sequential model does not admit an inconsistency. There is no causality checking problem because the semantics simply ignores the behavior: only the locally-consistent behavior of $x \equiv 1 \Rightarrow y \equiv 1$ is admitted.

Having a mathematical formalism that ignores unpleasanties has precedent in the fairness constraints of ω -automata. In that body of theory,¹ an automaton over an infinite sequence is given by the usual means: states and a transition relation. Additionally a set of

1. Some of the theory of ω -automata was related in Chapter 3. Other overviews are Kurshan [454] and Thomas [684].

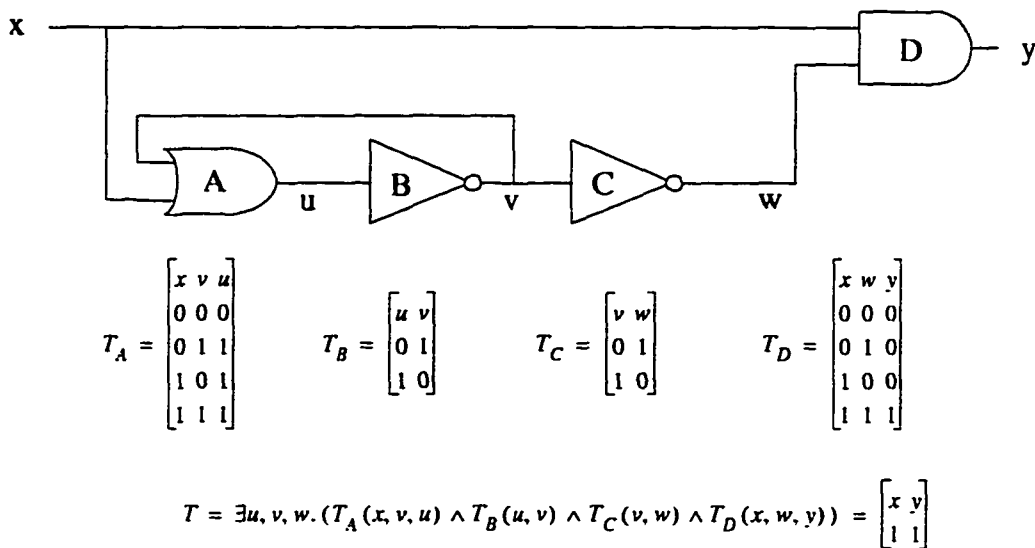


Figure 4-13. A Non-Causal Circuit Under a Vacated Semantics

fairness constraints are provided which declare that certain behaviors of the automaton are to be ignored (conversely accepted). The ω -automata can be seen as an instance of vacated semantics where problematic infinite behaviors are ignored (conversely accepted). The vacated semantics approach is extremely convenient from a theoretical perspective as there is no need for clumsy Two-of-three choices, separated semantics or *post hoc* restrictions on admissible descriptions.

Where the vacated semantics approach is problematic is at the border between verification and synthesis. The scenario is the top-down verification methodology proposed by Kurshan [454]. There an abstract design is progressively refined and verified until finally at the most concrete level implementation is produced (the final synthesis step). The methodology implies that properties are proved on a system description under a vacated semantics and then an implementation is synthesized based on the “verified” description. The problem is that a given design property proved on the abstract model need not hold on the actual implementation. Consider the property $AG\neg(y \equiv 0)$ in the example of Figure

4-13. The property succeeds because the circuit is a model for the property under Combinational/Sequential semantics yet an implementation of the circuit produces $y \equiv 0$ when $x \equiv 0$.

4.5 Review

The limits on semantic models have been explored in this chapter. Three orthogonal aspects of a semantic model were defined. These were *responsiveness* which is ability to define output values that are specific to an input value, *modularity* which is the exclusive reliance on output values as the concurrent coordination vehicle and *causality* which is the existence of a consistent order on microstep execution that is preserved in concurrent compositions. These three attributes were shown, in sum, to be incompatible by the RMC Barrier Theorem: no semantics can be defined which has these three properties and is not self contradictory.

Within the limitation of the RMC Barrier Theorem, the some of the many microsemantic variations were explored. The microsemantics for the communicating Moore machines, the “Codesign” Finite State Machines, several versions of the StateCharts and the Synchronous Languages were presented. Finally, the possibilities for surpassing the RMC Barrier were explored. These possibilities ranged across accepting the limitation directly, constructing a separated semantics with different choices for small and large scale composition, selecting only systems that have *RMC* and simply ignoring the contradictions.

The RMC Barrier and microsemantic decomposition of the transition relation forms a framework for analysis and classification of semantic models. Any finite state semantic model can be classified according to its microsemantic construction, its position with respect to the RMC Barrier and its method for surpassing it. The next two chapters survey other more applied instances of semantic models and programming languages in this light.

5 Applied Semantics

The previous two chapters dealt with semantic models at a fundamental level and the limits on the structure that can exist in a finite state semantic model. In Chapter 4 two possible approaches were investigated: the use of a microsemantics and selective attainment of RMC. Another method for tailoring a semantic model for a specific use is to dress it up with some extra syntax or other structure. This chapter reviews some of the many sugarings that have been proposed for the standard models.

Raw semantic models are extremely abstract and often don't relate directly to an application area. Historically, extra notation and conventions have been added in order to induce enough flavor and structure into the underlying semantic model that its use in relates more directly to a real-world application. The flavor aspect implies that in many cases models are domain-oriented if not outright domain-specific. Intended application areas range from recurrence equations in digital signal processing to inter-component coordination in a control-dominated embedded system settings. The extra structure arises from the need to make the model more directly usable in verification or synthesis. One of the most common examples is the definition of a composition operation which allows a system to be defined in terms of many components of manageable size.

The following sections present semantics in a more applied context using the RMC Barrier Theorem to analyze the features and fundamental expressiveness of the semantics.

The sections are ordered along the lines increasing complexity and internal structure. The first instance, the Asynchronous Shared Memory model offers the fewest semantic features for coordination. The examples increase in complexity and expressiveness to the case of the Communicating Sequential Processes and the Clarke Languages where the decision problems are undecidable. Backing away from undecidability one step gives the Synchronous Languages. The argument presented here is that the $RM\bar{C}$ semantics of the Synchronous Languages when constrained to C offers the best compromise of expressiveness while retaining the decidability of finite state semantics.

5.1 Asynchronous Shared Memory (ASM)

Probably the simplest model of concurrent computation is the Asynchronous Shared Memory (ASM) model [498] [500]. In that model, computation occurs amongst a number of processes, each having their own local state space. Inter-process coordination occurs through a global shared memory that can be read and written by all processes. This is depicted in Figure 5-1. The model of concurrency is a nondeterministic interleaving of the computations of each of the processes.

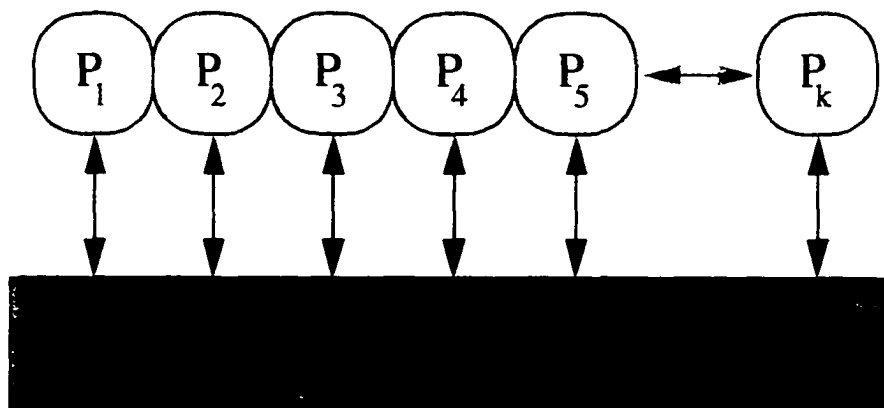


Figure 5-1. The Communication Model of Asynchronous Shared Memory

The transition relation of an operation op_i in a single process is of the form:

$T_i^{(op)}(c, n_i)$ which expresses that the operation is enabled on the whole system state c but it affects only a subset of the next state n_i . Additionally, there is the stability constraint which requires that the states not changed in op_i remain the same in the next step. The transition relation for the system as a whole is given in the following form:

$$T_i(c, n) = T_i^{(op)}(c, n_i) \wedge \prod_{j \in \{c_j | c - c_i\}} (c_j \equiv n_j) \quad (\text{Eq 5-1})$$

$$T(c, n) = \sum_i T_i(c, n) \quad (\text{Eq 5-2})$$

In particular, the structure of $T_i^{(op)}$ in Eq 5-1 is phrased in terms of a “control”-transition component $X_i^{(pc)}$ and a “data”-transition $X_i^{(data)}$ component as follows:

$$X_i^{(pc)}(c_{pc_k}, n_{pc_k}) = (c_{pc_k} \equiv L(\alpha)) \wedge (n_{pc_k} \equiv L(\beta)) \quad (\text{Eq 5-3})$$

$$T_i^{(op)}(c, n_i) = X_i^{(pc)}(c_{pc_k}, n_{pc_k}) \wedge X_i^{(data)}(c, n_i) \quad (\text{Eq 5-4})$$

This form makes it clear that the transition T_i of Eq 5-1 is enabled just when the program counter of the process P_k has the value specified in the relation $X_i^{(pc)}$, namely $L(\alpha)$. What is convenient about this representation is that this enabling condition is independent within each term T_i in the summation of Eq 5-2.¹ The T_i are independent in the sense that adding more processes P_k , or adding more transitions within the existing process set merely adds more disjuncts to Eq 5-2.

Transition independence is an important property because in case $F \{Q\}$ and $B \{Q\}$ are monotonic and continuous then the explicit construction of $T(c, n)$ can be avoided. This can result in a tremendous increase in efficiency in verification because the construction of a monolithic form of $T(c, n)$ is one of the major bottlenecks in verification schemes. The exact statement of the conditions when T can remain disaggregated and their ramifications on microsemantics are deferred until Chapter 7 where the properties of $F \{Q\}$ and

1. The enabling conditions are not necessarily *disjoint* because the transition relation T may be nondeterministic, having more than one T_i enabled at a given step.

$B\{Q\}$ are studied in detail.¹ The relevance however in this chapter, is the observation that there are other semantic models where the components T_i are *not* independent and where an aggregated form of $T(c, n)$ must exist for the $F\{Q\}$ and $B\{Q\}$ computations.

The ASM model was originally developed to model concurrency as one might find in a multitasking operating system and as such its major artifacts, multiple independent processes and a shared memory, reflect that background. In the context of the RMC Barrier from Chapter 4, the ASM model is a zero-of-three choice: it is $\bar{R}\bar{M}\bar{C}$. This is because there are no outputs or inputs to processes (\bar{R}) and the concurrent behavior of the system in a step is governed by the transitions enabled by the state of the global memory and the running process (\bar{M} and \bar{C}).

5.2 Selection/Resolution (S/R)

One step away from a single global memory where any process can read and write any memory cell is a system with a structured use of the global memory. In Kurshan and Goni-path's Selection/Resolution (S/R) model [8] [449] [424] each process is endowed with a local state space as before. The distinction comes in the restriction that each process can only write to the part of the global memory that is assigned to it. A process is allowed read from any cell of the global memory. Structurally the system is as shown in Figure 5-2.²

Operationally the system computes according to the following recipe: in each cycle every process defines its outputs and then every process modifies its internal state based on the global memory. The output definition step is called *selection* and the per-process computations are called *resolution*. This simple structure leads to a two-phase structure of time where every process selects and then every process resolves. In terms of a microstep

1. Succinctly stated though this condition is that existential quantification distributes across disjunction:

$$\exists x.f(x, y) \vee g(x, y) = (\exists x.f(x, y)) \vee (\exists x.g(x, y)).$$

2. Based on Figure 7.1, page 111 in Kurshan [454].

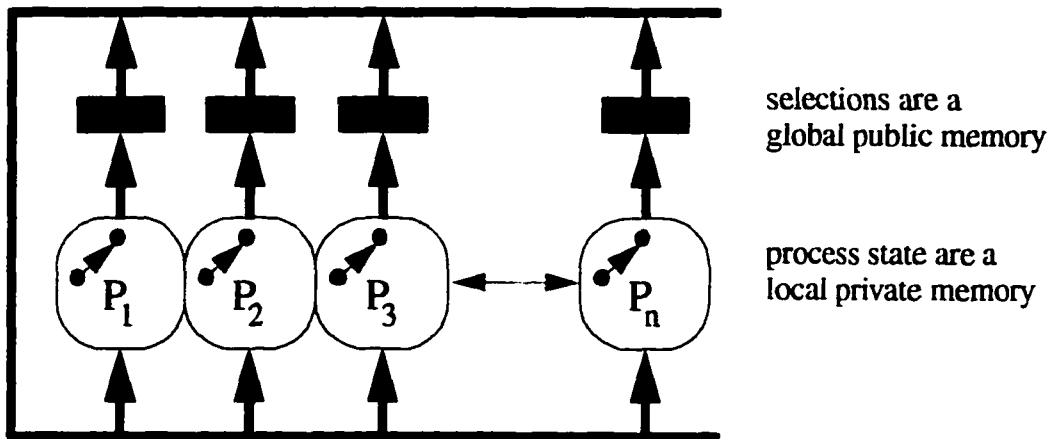


Figure 5-2. A Network of Selection/Resolution Processes

behavior, all the output-defining microsteps occur before any of the successor-deciding microsteps. The two views of the structure of time are shown in Figure 5-3.

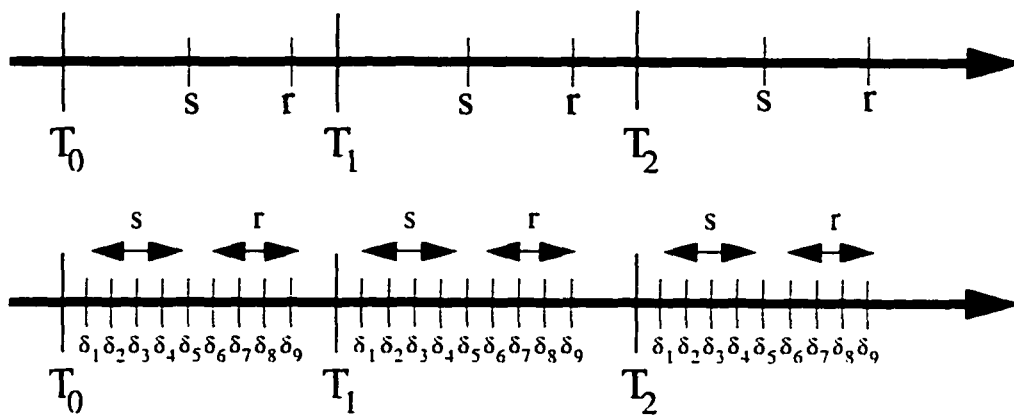


Figure 5-3. Two Views of Time under Selection/Resolution

In the pure form, the two-phase selection/resolution semantics allows for arbitrary reordering of the output-defining microsteps within the selection phase. In this case, no output is dependent upon any input from another process. The output relation O_i of each process is completely determined by the state of the processes exactly as in Eq 4-3. In this case, the transition relation of every process is of the form:

$$T_i(c_p, i, o_p, n_i) = X_i(c_p, i, n_i) \wedge O_i(c_p, o_i) \quad (\text{Eq 5-5})$$

Thus, in the pure form the S/R semantic model is \bar{RMC} and has a microsemantics as was given in Example 4.3.4.1.

Further extensions to S/R, the Mealy-complete networks [454], allow for the output of a process to be determined by the inputs received from other processes. This gives a transition relation following the form of Eq 4-2:

$$T_i(c_p, i, o_p, n_i) = X_i(c_p, i, n_i) \wedge O_i(c_p, i, o_i) \quad (\text{Eq 5-6})$$

The dependence of the output relation O_i on the input i clearly makes the semantics R . For this reason, some extra constraints on the system description must hold. The first is that the pure transition relation X_i of each process is subject to a condition of *Mealy completeness*. This condition ensures that each process in the network always has some transition enabled no matter what input it receives. This condition is also called *lockup free* and is a requirement that each process' behavior be completely defined for all possible inputs.

The second condition is that the network communication graph must be acyclic. This ensures that there is always a state-independent causal order for the network \leq_N which is consistent with the state-independent causal order of each process \leq_{p_i} . The partial order \leq_N is the minimal set of constraints on the order in which the process selections must be defined. In this extended case the semantics of S/R is $RM\bar{C}$ but the structural restriction on the communication graph ensures that only systems where RMC holds are admitted.

Under either the pure or the extended semantics, the macrostep behavior of the whole network is defined by considering the outputs of all the processes as the inputs to each: in either Eq 5-5 or Eq 5-6 take $i = (o_1, \dots, o_{i-1}, o_{i+1}, \dots, o_n)$. Coupled with the output o_i , from the interconnection (i, o_i) is just $o = (o_1, \dots, o_n)$. The coordination among all the processes in the network is the set of steps that is consistent with the coordination of each process:

$$T(c, n) = \exists o. \prod_i T_i(c_i, o, n_i) \quad (\text{Eq 5-7})$$

5.3 Combinational/Sequential (C/S)

The Combinational/Sequential (C/S) model [107] [37] [358] was developed to model behavior in from a hardware perspective. Behavior is described in the form of a network of generalized gates and latches. Gates are generalized in the sense that they are multi-valued, have multiple outputs and may have nondeterministic input/output relations. Gates are referred to as *tables* because the behavior of a gate is specified by explicit enumeration of the elements in the transition relation in tabular form. Latches are generalized in the sense that they may have nondeterministic initial values.

The C/S model is the semantic basis for the BLIF-MV internal representation which was described in Section 4.4.5. In that presentation the macrostep behavior of a C/S network was given in equations BLIF-MV-1 and BLIF-MV-2. Those equations are repeated here for concreteness with the two composition situations being depicted in Figure 5-4 and Figure 5-5 respectively.

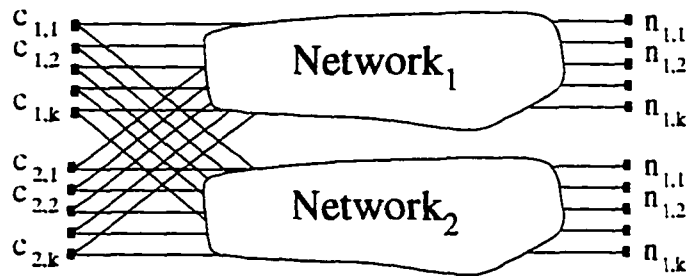


Figure 5-4. The Parallel Composition of Two C/S Networks

The parallel composition of several networks is defined by the conjunction of the transition relations of each of the components:

$$T(c, n) = \prod_i T_i(c, n_i) \quad (\text{Eq 5-8})$$

Here, the transition relation of each subnetwork is written $T_i(c, n_i)$ to denote that T_i may

depend on the output of any latch, the whole of c , but only defines a subset n_i of the latches.

The transition relation T_i of a network is typically given in terms of multiple levels of logic where the transition relation for each gate can be reasonably defined by enumerating the rows of its transition relation in sum-of-products form. In this case, the transition relation of the network $T(c, n)$ is given in terms of a set of intermediate variables z .

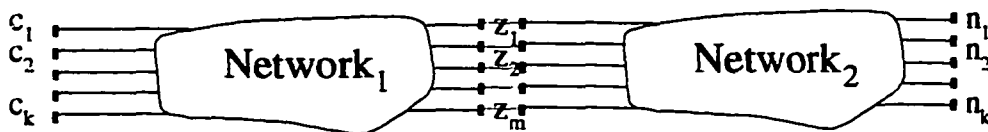


Figure 5-5. The Serial Composition of Two C/S Networks

The transition relation of the serial composition of two networks is given by:

$$T_{B \circ A}(c, n) = \exists z. T_A(c, z) \wedge T_B(z, n) \quad (\text{Eq 5-9})$$

With regard to the sense of independence which was used to in the previous section, the transition-relation behavior of C/S networks are not independent for either parallel or serial composition. The addition of a new subnetwork in either the parallel or the serial case means more conjuncts with interdependence amongst the conjuncts in Eq 5-9 being explicit in the shared intermediate variables z . The conjunctive form of both Eq 5-8 and Eq 5-9 require that an aggregated version of $T(c, n)$ be used for the $F\{Q\}$ and $B\{Q\}$ computations. As a result a number of heuristic techniques have been developed to manage the conjunction and quantification computations [42] [43] [360] even to the extent of performing minimization on intermediate computations [655].¹

The C/S model addresses the RMC Barrier by way of being a vacated semantics. As was illustrated in Section 4.4.5 the contradictory denotations in the network simply disap-

1. For example, Aziz, *et al.* [43] report that for moderately-sized examples produced from a Verilog compiler [164] that there are roughly 1600 transition relations and 1500 intermediate variables in Eq 5-9.

pear when interpreted under Eq 5-9.

5.4 “Codesign” Finite State Machine (CFSM)

The previous semantic models concentrated on defining behavior exclusively in terms of macrosteps. In contrast, semantic activity can be defined at the microstep level with the notion of a macrostep being a purely derived quantity. The “Codesign” Finite State Machine (CFSM) [167] is a semantic model that starts with a set of behaviors at the microstep level and derives macrosteps as causal projections of the microstep traces.

The CFSM microsemantics was presented in Example 4.3.4.5 so the presentation here is restricted to the presentational aspects of the model. A “Codesign” Finite State Machine is defined as:

$$C = (I, E, O, R, F)$$

where:

I is a set of typed input events. For $i \in I$ is given by $i = (n, V)$ where n is an event name and V is a finite set of values $V = \{v_0, v_1, \dots, v_k\}$ and $|V| = k$.

$E \subseteq I$ is a set of trigger events where for $e \in E$ the value set of the event is a singleton. $|V| = 1$.

O is a set of typed output events in the same style as I subject to $O \cap E = \emptyset$.

$R \subseteq \{ (n, v_j) \mid (n, V) \in O, v_j \in V \}$ defines the reset states of the outputs.

$F \subseteq 2^I \times 2^O$ is the transition relation of the machine.¹

Events are considered to be time stamped and of the form $e = (n, v, t)$ indicating that an event n occurs with value v at time t . Time is isomorphic to the natural numbers which leads directly to the denotation of a CFSM as a set of traces.

A CFSM is described in the *Software/Hardware Intermediate Format (SHIFT)* format

1. The transition relation F is subject to other ancillary conditions such as the input containing at least one trigger event. These peculiarities are important for pedagogical reasons but can be ignored for the treatment here.

[168] which, much like the BLIF-MV intermediate format [107], uses transition relations to specify behavior. The difference in SHIFT is that explicit access to the information ordering is allowed. For any event in $I \cup O$ it is possible to refer to the value aspect w and to its information ordering aspect $*w$, which is called the *trigger of w* . Compilation from high-level languages into the CFSM formalism is accomplished by creating a SHIFT description of a network of CFSMs which, in aggregate, mimic the desired semantics of the high-level language.¹

A network of CFSMs is defined in terms of the sharing of outputs and input event sets between two or more machines. Broadcast communication between machines is straightforward: the output event set of one machine can be mentioned in the input event set of any number of machines. The denotation of a CFSM network is a trace structure that satisfies all of the traces of all its components. The CFSM semantic model addresses the RMC Barrier as a two-of-three choice, opting to select \bar{M} as was shown in Example 4.3.4.5. Unfortunately, due to the \bar{M} property, there is no guarantee that an extensible trace structure exists for every network.²

The CFSM model was designed for an application area of mixed hardware and software

1. One claim [169] is that the semantic model of the CFSM network is complete for Synchronous Languages such as Esterel. This class of languages [62], among which are Esterel, Lustre, Signal, and State-Charts (specifically the Argos variant), have been shown to be $RM\bar{C}$ [378]. The CFSM model was shown in Example 4.3.4.5 to be $RM\bar{C}$. Thus the CFSMs are not complete for the Synchronous Languages.

It is feasible to produce semantically faithful CFSM networks only for subsets of the Synchronous Languages. Yee [733] has defined a CFSM network semantics for a subset of Esterel which is causal. The subset of Esterel is guaranteed to be C by structurally disallowing instantaneous dialog. In a strong sense, this subset is no longer Esterel, it is something else more akin to SML [98] or CSML [195].

2. Were CFSM trace structures defined by the ω -regular languages, then this statement could be as simple as stating that the resulting trace set is vacuous. The CFSM trace structures are prefix-closed which makes the statement a bit more complex because prefix-closed traces are necessarily finite. The presence of a deadlock, which is the dynamic manifestation of \bar{M} (c.f. the proof of \bar{M} in Example 4.3.4.5), does not render a trace vacuous, but it does make it shorter than one might expect. It makes a trace structure non-extensible.

Extensibility of a trace structure T is defined as $\forall t \in T. \exists s \in T. \exists t' \in \Sigma^*. s = tt'$ which means that if there is some behavior t in the trace set, there is another legal behavior s which progresses t by a few more steps. Deadlock prevents the extension of t to t' .

systems [169]. The perspective adopted is that the exact timing of events in either domain can vary widely between the two implementation domains and that the semantic model must take into account the nonzero delay that exists in physical systems reacting in the real world. Accordingly, Berry's *perfect synchrony hypothesis*,¹ that the system reacts faster than its environment, is discounted and the opposite view is assumed. The CFSM model assumes that reaction takes some time and uses this assumption to develop a theory where overlapping and interleaving of causes and the natural skew between causes and reactions is modeled directly. This was illustrated in Figure 4-9.

The CFSM Synthesis Theorem [168] gives a constructive method for deriving realistic implementations of CFSM networks in terms of a network of Mealy machines. The synthesis theorem states that for every CFSM trace over events e_i there exists a FSM trace over pairs of sets of events structured into cause-reaction pairs (C_j, R_j) . The converse holds as well. The theorem constructs a projection of microstep event traces in microtime onto a trace of cause-reaction pairs (C_j, R_j) . This projection defines macrosteps in terms of a causal ordering of microsteps and forms the basis of a constructive synthesis procedure.²

1. The definition and implications of the pure synchrony hypothesis are variously presented in Berry and Cosserat's early publication of Esterel [79], Benveniste and Berry's overview of the Synchronous Languages [62], or Halbwachs [320]. This is also covered in Section 5.7.1 of this work.

2. The CFSM Synthesis Theorem also highlights a major confusion in the interpretation of the perfect synchrony hypothesis. Perfect synchrony refers to time *relative to events* rather than to an absolutist notion of metric time. That a reaction takes "no time" means in context that no events *of that kind* occur again in the cause assessment for the reaction. That distinct cause sets may be interleaved or that causes and reactions are skewed across microtime is illusory. Halbwachs [320], Chapter 7 reviews this as does Section 5.7.1.1 of this work.

Despite the interleaving and pipelining allowed in the CFSM trace semantics, exactly such a restriction to a single reaction per cause is placed upon microstep event traces as well. This is fundamental to the CFSM Synthesis Theorem statement that for every microstep event trace there exists a macrostep trace. As such, the totally-ordered notion of time at the microstep event level corresponds at the macrostep level to the state-dependent partial order $\leq_Q(i, o)$ of the RMC Barrier Theorem. The CFSM Synthesis Theorem implies that CFSM macrostep traces are consistent with Berry's pure synchrony hypothesis.

5.5 Communicating Sequential Processes (CSP)

A second trace-based model is Hoare's Communicating Sequential Processes (CSP) [353] [354]. CSP was designed as a means for describing *any* computation, not merely those falling into the reactive system paradigm. As a result, the formalism is extremely general providing primitives for sequentiality, concurrency, deterministic choice, termination, failure, symbol change, concealment and nondeterminism. With all these features, CSP programs are complete with respect to the computable functions and can model any coordination scheme. In the general case CSP programs are not restricted to finite or even bounded state so an analysis according to the RMC Barrier as in the previous sections is not directly applicable.

Undecidability and non-finiteness are problematic in application domains such as plant control or behavioral modeling so the interest in CSP for these areas has focused on subsets which have bounded or finite state and where the analyses of interest are decidable. In these cases, which are detailed in the sequel, one can readily identify the \overline{RMC} properties based on exactly the same trace-theoretic arguments which were used for the CFSM model in Example 4.3.4.5.

The CSP subsets are also interesting because they exactly identify the next set of semantic features that might be added to a semantics to circumvent the RMC Barrier. The CSP subsets provide a framework where these extensions can be incrementally and independently added to the automata-theoretic basis that has been used to this point. Theoretical results exist to show that these extensions in sum, make the decision problems undecidable or imply a semantic model with unbounded state.

5.5.1 Theoretical Communicating Sequential Processes (TCSP)

The theoretical aspect of CSP are often referred to as TCSP to distinguish the process algebraic aspects of the language. Within the process algebra, the exclusive focus of the

language is the use of communication as a means of synchronization. All inter-process synchronization and communication is accomplished through message passing over *channels* which are an unbuffered single-cast communication medium. The two important operations in this regard are the sending of a message expression E by a process P , and the reception of a message by a process P into a local variable V . The first operation is called *send* and is written $P!E$ while the second is called message *receive* is written $P?V$.

The significant semantic feature of the message traffic over a channel is that the message send/receive operation is synchronous between the sender and the receiver. If $P!E$ is invoked and the receiver is not also executing $Q?V$ then the sender blocks. The converse condition is true as well with the receiver executing $Q?V$ and awaiting the reception of a message written by $P!E$. Synchronization of this sort is often presented in concrete programming languages as a *rendezvous* as in Ada [24] or a remote procedure call.

In the full presentation of CSP [355] a simple simulator was presented in tandem with the CSP language elements in order to demonstrate the realizability of the language. This gave an operational meaning to structures in the language but did not directly define the behavior of a CSP program. The simulator also did not provide any direct means to formulate analyses of CSP programs. The trace-theoretic model [118] [119] appeared later and forms the basis for such analyses over prefix-closed traces. In that model the behavior of a program is interpreted relative to a set of successful traces S and a set of failure traces F . These two sets define the executions of the program that end in termination and deadlock respectively. That trace theory was presented in Section 2.2.1.

5.5.2 Finitely Recursive Processes (FRP)

One interesting subset of CSP is the Finitely Recursive Processes (FRP) of Inan and Varaiya [388]. Their application area is plant control where the distinction between sending a message and receiving a message is somewhat blurred. A physical plant executing

P!E isn't going to wait until the plant observer/controller gets around to executing $Q?V$. Their variant of CSP reflects this by defining communication in terms of shared events rather than in terms of the simultaneous execution of send and receive. In this view, if two or more processes perform the same event α at the same time point then a synchronization and communication between them is said to occur. It is unspecified which process is understood to perform the send and which performed the receive.

Of interest here is their proposal to define verification problems on FRP systems in terms of containment of trace sets tr :

$$tr\{SYS_{FRP}\} \subseteq tr\{TASK\} \subseteq \Sigma^*$$

The verification problem that SYS_{FRP} obeys $TASK$ is defined in terms of the containment of the language defined by the trace set of SYS_{FRP} by the language defined by the trace set of $TASK$. This is verification posed in terms of language containment.

They note that when the FRP systems can be shown to have finite state, then the languages of the trace sets are guaranteed to be *-regular and the containment problem is decidable.¹ The remaining obstacles blocking the practical implementation of this idea were the lack of a means for specifying the design property $TASK$, the practical means for manipulating large trace sets and the identification of constraints that guarantee that the language of a given FRP system is *-regular.

An FRP scheme is a system of n -dimensional recursion equations in the variables X

1. A table listing the decision properties for the various classes of languages can be found in Hopcroft & Ullman [368], page 281.

and Y where:¹

$$X = (X_1, X_2, \dots, X_n)$$

$$Y = (Y_1, Y_2, \dots, Y_n)$$

and the system is specified as:

$$Y = g_0(X)$$

$$X = f(X)$$

where g_0 is made up of:

1. The constant processes *STOP* and *SKIP*,
2. The projection process $\pi_i(X) = X_i$,
3. The local change operator $f^{[-B+C]}$
The global change operator $f^{\llbracket -B+C \rrbracket}$,
4. Synchronous composition $f_1(X) \parallel f_2(X)$,
5. Sequential composition $f_1(X); f_2(X)$.

and f is of the form:

$$f(X) = (f_1(X), f_2(X), \dots, f_n(X))$$

$$f_i(X) = (\alpha_{i,1} \rightarrow f_{i,1}(X), \alpha_{i,2} \rightarrow f_{i,2}(X), \dots, \alpha_{i,k} \rightarrow f_{i,k}(X))$$

The system (f, g_0) can be thought of as defining a state machine where f is the next-state function and g is the output function. Unfortunately such systems do not necessarily have finite state due to the presence of sequential composition (rule 5). The languages defined by FRP systems are shown to be equivalent the Petri net languages² so a practical algorithm for language containment is not readily apparent.

Despite the excess power of the FRP formalism, Inan and Varaiya argue that sequential

1. Inan and Varaiya [388] adopt some liberties with the TCSP notation. In particular they allow the event set of a process to change dynamically (rule 3) and use this feature to model termination. This allows the sequential composition operator to be defined without reference to any special termination event \surd and removes certain ambiguities in the definition of mutual recursion under sequential and synchronous composition. They also write processes in the form of vector recurrence equations instead of using Hoare's fixed point operator over processes $Y = \mu Y.F(Y)$. Their notation is used here directly.

2. A review of Petri nets as a semantic model was given in Section 2.2.3. A language-theoretic characterization of Petri nets is given in Peterson [584].

A short summary of the Petri net languages are larger than the *-regular languages but incomparable to context-free languages. Some context-free languages cannot be generated by any Petri net and some Petri net languages cannot be recognized by any pushdown finite automaton.

composition operator is important. They note that although FRP systems are as expressive as Petri nets which are strictly more powerful than Finite State Machine (FSM) systems, neither formalism supports sequential composition. A definition of sequential composition is possible in the FSM paradigm so long as termination is a state-specific property, termination being modeled as an anointed “final” state. Termination in the FSM formalism however cannot be extended to be a path-specific property encompassing historical information. In the Petri net formalism where this is possible, is not obvious how to express the termination of a net.

At a deeper level, their argument defends the recursion equation formulation of the CSP process algebra as a fundamental contribution. Their view is that the data-centric focus of the FRP has a number of advantages over the state-centric view of processes as the states of an FSM. The data-centric aspect of the FRP describes behavior in terms of a task model where *action* is the primary aspect and state is a derived quantity. This can be seen in the very definition of the FRP system (f, g_0) in which the state is *encoded* in an n -dimensional vector of variables while the system description rests primarily in the process functions f and g_0 . This allows the data-centric FRP descriptions to be much more compact than their state-centric counterparts. Also of interest is their observation that FRP recurrence equations are independent in the sense defined in Section 5.1. This independence makes it extremely easy to modify the behavior of a system by simply adding a new equation. Finally, the use of mutual recursion in the definition of f defines a natural notion of hierarchy in terms of modules. A module of equations is a set of mutually recursive equations. A set of equations $\{X_1, X_2, \dots, X_k\}$ are said to be mutually recursive when X cannot be written as $X = Y \cup Z$ where $Y \cap Z = \emptyset$ and each of Y and Z are mutually recursive. This notion of modularity allows for independent implementation and scheduling of modules. These arguments are recapitulated in the justifications for synchronous dataflow languages which are reviewed in Section 5.7. The relevance here is that FRP can

be viewed as an independent approach to a synchronous dataflow language, starting from a basis in the CSP process algebra.

5.5.3 Reduced Communicating Sequential Processes (RCSP)

Following the FRP, Cieslak and Varaiya [175] gave a more restricted version of CSP which developed stronger connections between the FSMs, production systems and Petri nets. In the Reduced CSP (RCSP), a system (f, g_0) is again given as an n -dimensional vector of recurrence equations:¹

$$Y = g_0(X)$$

$$X = f(X)$$

where g_0 is made up of:

1. The constant processes *STOP*,
2. The constant process *SKIP*,
3. The projection process $\pi_i(X) = X_i$,
4. Synchronous composition $f_1(X) \parallel f_2(X)$
5. Sequential composition $f_1(X); f_2(X)$

and f is of the form:

$$f(X) = (f_1(X), f_2(X), \dots, f_n(X))$$

$$f_i(X) = (\alpha_{i,1} \rightarrow f_{i,1}(X), \alpha_{i,2} \rightarrow f_{i,2}(X), \dots, \alpha_{i,k} \rightarrow f_{i,k}(X))$$

Within these rules, the system $RCSP_k$ is defined as the full $RCSP$ system but without the rule k .² The key observation is that an $RCSP_5$ system is equivalent to an FSM while an $RCSP_4$ system is equivalent to a simple grammar or production system. The full $RCSP$ system is shown to retain the expressiveness of the Petri net languages as was the case with the FRP.

1. Cieslak and Varaiya [175] continue with the use of a recurrence equation notation used in the FRP presentation [388], though they retain a more traditional definition of termination and sequential composition based on an explicit "exit" event \surd . Again, their notation is used here directly.

2. The nomenclature $RCSP_k$ is idiosyncratic to this presentation but follows used in the presentation of the Clarke Languages (L_k) [178]. These languages are presented in Section 5.7. The denotational theory of the Synchronous Languages (SL_k) [64] [65] uses this style of notation but in a different way. The latter theory is presented in Section 5.7.2.

Using this framework, Cieslak and Varaiya give proofs which show that *RCSP* is undecidable for the standard verification problems: deadlock, termination, potential executability, liveness, boundedness and language equality. Of these decision problems, all but reachability are decidable for *RCSP*₄ while for *RCSP*₅ all the problems are decidable. The important contribution of the *RCSP* semantic model is its structure in terms of separable rules in which the subsets *RCSP*₄ and *RCSP*₅ readily correspond to other known semantic models. In the context of the development here, the *RCSP* formalism identifies which semantic features provide the next level of expressiveness beyond the finite state semantic models discussed so far. Conversely, the structure of *RCSP* shows how a fully general model of computation such as *TCSP* can be progressively restricted and structured to the point where its decision problems are solvable.

5.6 The Clarke Languages

A second limitation on applied semantics was provided by Clarke in defining the *characterization problem* [179] for a large class of programming languages which have since become known as *the Clarke Languages* [178]. The characterization problem establishes the impossibility of defining a relatively sound and complete Hoare Logic¹ for the Clarke Languages. Further, for lack of a relatively sound and complete Hoare Logic, the halting problem for these languages is undecidable, even when the variables in the programs are restricted to finite domains.² This result is significant because it established an explicit list of five programming language features which, when appearing in combination, cause undecidability. Any one of these features when removed restores decidability. It is espe-

1. *c.f.* Section 2.1.1.

2. The proof that the Clarke Languages are undecidable for $|D| \geq 2$ revolves around being able to program any computable function with a Clarke Language. By Church's Thesis a Turing Machine can compute any computable function. Pascal code to simulate a Turing Machine is given in Cousot [213], pages 924-929. This construction is nontrivial because a dynamically allocated data structure is not used to represent the unbounded tape; `new/dispose` is not used.

cially powerful because it applies even in case the variable domains in the program are finite but non-trivial ($|D| \geq 2$).

The context of the characterization problem is the definition of an axiomatic semantics for pointerless subsets of imperative programming languages such as Algol [556] [557] or Pascal [723]. A Clarke Language¹ L is an imperative programming language, under some mild restrictions, which includes procedures and which has the following five features:

- i. procedures as parameters of procedure calls (without self-application),
- ii. recursion,
- iii. static scoping,
- iv. use of global variables in procedure bodies,
- v. nested internal procedures as parameters of procedure calls.

The Clarke Language L is undecidable. At finer level of classification, the Clarke Language L_j is defined by disallowing feature j . These languages are decidable.

The detailed restrictions on the language are indeed quite mild. Heap storage allocation is disallowed but the allocation of fixed-size activation records on an unbounded stack is allowed. The procedures must take only a finite number of parameters and they must contain only a finite number of local variables. All variables must be defined over finite domains and there must be no sharing of variables via aliasing. The proscription against aliasing clearly disallows pointers. It also precludes procedure call-by-reference but not copy-in/copy-out. It however may or may not disallow the renaming declarations such as the **renames** of Ada or the **alias** of VHDL, depending on whether the particular declaration instance can be formulated as a syntactic rewrite.

Of interest at the finer level is the language L_{ii} , the Clarke Language without recursion. The fact that halting is undecidable even for the recursionless subclass is extremely pro-

1. Following the presentations in Clarke [180] and Cousot [216]. Other presentations [186] have used L_1 to denote "while" programs with L_2 through L_6 denoting what is here referred to as L_i through L_v .

found. In contrast, the halting problem is decidable for finite-state “while” programs [410]. Theoretically at least, one has but to enumerate the reachable states of the program while searching for either a halting state or a repeated state. Another surprising consequent of this result is that any Clarke Language allowing recursion, say choose L_i or L_{iii} , is undecidable. In contrast, language acceptance by a pushdown automaton (PDA) is decidable [411].¹ Thus it is not possible to establish decidability of these languages through a naive analogy between a stack machine executing a recursive Clarke Language and acceptance of a string by a PDA. These distinctions center around two aspects of the Hoare Logic of Clarke Languages: the complexity of the predicates in the preconditions, statement and postcondition and the treatment of names in those predicates.

A Hoare Logic for a language defines the semantics of the language through a set of precondition-statement-postcondition structures $\{P\} S \{Q\}$ where P and Q are predicates over a finite set of state variables. In Section 2.1.1 this form was shown to be equivalent to a relational form $(\gamma_i, \gamma_{i+1}) \in Op[[S]]$ where the γ_j are predicates over the some finite set of state variables. The predicates γ_j are the characteristic functions of the program states before and after the statement S is executed. Implicit in the formulation of $Op[[S]]$ is a frame assumption [162], that the states variables not mentioned explicitly as changing across the transition γ_i to γ_{i+1} must remain the same. In a sound and complete Hoare Logic, the variables in the predicates γ_i and γ_{i+1} are all used in one-to-one correspondence with the value-containing objects of the program. All such objects have distinct program variable names so the correspondence is direct and importantly, any object which cannot be explicitly referenced through a named program variable cannot change during the execution of a statement S .

In contrast, for the Clarke Languages, there is *not* a one-to-one correspondence between

1. *c.f.* the table of decidability results in Hopcroft & Ullman [368], page 281.

program variable names and state variable names in the predicates γ_j . There exist anonymous value-containing objects in these programs which *can* be modified in a statement S . As such the precondition-to-postcondition transition relation cannot be expressed in terms of a predicate pairs over the finite set of state variables. For example at an arbitrary statement S_k , objects buried on the stack cannot be accessed through any named program variable. These objects are anonymous from the particular vantage point of S_k so they are out of reach of statements in the logic. They cannot be characterized in terms of a predicate γ_j over the finite set of state variables. In case S_k is a procedure call, then thanks to static scoping rules, these objects can be modified. Yet $Op[[S]]$ can only express how named objects change with a frame assumption declaring that anonymous remain unchanged. This is a contradiction which means that there is no way to characterize programs with complex control structures and variable reference conventions in terms transition which are defined in a syntax-directed fashion relative to statements S .

In summary, the characterization problem for the Clarke Languages centers around the complexity of control structure and program variable reference conventions. Hoare Logic expresses semantics from a local vantage point with respect to statements S . The Clarke Languages give the programming language constructs which make the control structure and variable reference conventions too complex for the local perspective to accurately characterize the global state transitions of the system. Specifically, Clarke's contribution was in showing that the complexity of the language L is too great to allow for automated analysis by any means.

5.7 The Synchronous Languages

The semantics described in the previous sections have increased in complexity and expressiveness to the point where the last two semantics offered were undecidable. The Synchronous Languages in contrast offer the same sorts of description styles and syntax

as the previous examples while retaining the decidability benefit of having finite and simple control structures. The Synchronous Languages are a class of widely diverse languages that all share a common set of semantic properties. Their diversity ranges across state-centric languages with imperative control flow such as Esterel [79] [295] or a State-Chart-style graphical notation such as Argos [501] [502] to data-centric languages given in the form of recurrence equations such as Lustre [68] [153] or Signal [464] [280]. At a most fundamental semantic level, all these languages have an RMC semantics. They bypass the RMC Barrier by admitting only system descriptions where RMC applies, though the actual form of the restriction, static or dynamic, is idiosyncratic to the language.

With a presentation of the microsemantics having been given in Example 4.3.4.4 and the approach to the RMC Barrier being clear, this section is devoted to characterizing the range of Synchronous Languages and the semantic commonality among them. The working definition used here is that a Synchronous Language has an RMC semantics and that the RMC Barrier is addressed with a per-system restriction to RMC . At a deep level, a formalization of the common semantic aspects remains an open problem though intuitive and semiformal arguments can be readily related. The informal attributes of the criteria for being a Synchronous Language is related in Section 5.7.1. Tutorial publications have drawn on intuitive notions of synchronous semantics to provide high-level context for the application domains and language design goals. Unfortunately this informal characterization has led to the inclusion of languages that are beyond the bounds of the class yet which informally seem to have the appropriate properties.¹

Halbwachs obliquely addresses this in his overview presentation,² pointing out that the

1. Specifically the introduction [62] and the ensuing article [195]. There, SML and CSML are characterized as being part of the Synchronous Language class. While their semantics is defined in terms of macrosteps, the fine structure is such that outputs only become defined in the succeeding cycle. They are $\bar{R}MC$.

2. Halbwachs [320], page *xi*.

synchronous point of view has been adopted almost exclusively in languages developed by researchers cooperating between four French research institutes¹ with other formalisms using these ideas only partially or *a posteriori*. With this in mind, it would be convenient if there were some consistent set of fundamental properties that were common among the group. Benveniste and Le Guernic have attempted such a characterization with their Denotational Theory of Synchronous Communicating Systems. An overview of their theory is the subject of Section 5.7.2. In Section 5.7.3 the various Synchronous Languages are placed within this theory.

5.7.1 Defining Attributes

Though very disparate in their presentations the Synchronous Languages all share a common philosophical outlook relating to the model of time, macrosemantics, concurrency and determinism. Any particular Synchronous Language is designed within this framework taking into account the other design goals such as imperative control flow or functional style. The next sections review the philosophy common to all the Synchronous Languages.

5.7.1.1 Perfect Synchrony

Probably the most striking feature of these languages is their model of time. Within the synchronous paradigm the model of time is intrinsically tied to the semantics of the language by modeling time in the form of events. Thus in a particular parlance, time could be referred to in units of *seconds*, *meters*, *ticks* or *resets*. There is a strong distinction between *metric time* as measured on a timepiece by an external observer and *event time* as measured by events from within the semantics. In the synchronous paradigm the qualifier

1. These are Ecole Nationale Supérieure des Mines des Paris (ENSM), Institut National de Recherche en Informatique et Automatique (INRIA), Institut de Recherche en Informatique et Systèmes Aléatoires (IRISA), Institut d'Informatique et de Mathématiques Appliquées de Grenoble (IMAG).

event is usually dropped when referring to time. This convention is adopted here as well where confusion cannot occur.

The direct effect of this association is the requirement that the system (programmed in the Synchronous Language) compute its responses enough faster than its environment that the abstraction is a reasonable one.¹ This has been called the *perfect synchrony hypothesis* by Berry [79]. It is a requirement that event recovery and reaction computation take no (event) time. The system must have completed one computation before the environment sends it another. This amounts to a timing analysis obligation of the same sort that VLSI design engineers must perform on synchronous circuit designs: it must be the case that the combinational logic computing each macrostep be faster than the system clock. When the perfect synchrony hypothesis holds then the behavior of an implementation is guaranteed to match the behavior predicted by the mathematics of the semantics. Synchronous languages have the property that the realization of every macrostep relation, in either instructions or gates, is simple in the sense that its critical paths can be easily identified and estimated. In the hardware case this implies that macrosteps be combinational and in the software case that the instruction stream be loop-free.

5.7.1.2 Multiform Time

Time also has a secondary aspect which is unique to the synchronous model. Time is measured exclusively in the form of events so the view of time is consistently a local one where each process measures time in its own context. The act of composition, either con-

1. In many introductory presentations, *c.f.* Benveniste and Berry [62] or Halbwachs [320], this requirement is phrased in terms of the system being *infinitely* faster than its environment.

Letting the system response frequency be f_{sys} and the environment event generation frequency as f_{env} , then requirement that $\lim_{f_{env}} \frac{f_{sys}}{f_{env}} = \infty$ is clearly too strong.

All that is required is that in all cases the system is *marginally* faster than its environment. The actual requirement is that the system is always finished reacting before the next event occurs. An example of the confusion between the allegorical sense of *infinitely faster* and the practical need for *marginally faster* in the interpretation of the perfect synchrony hypothesis was illustrated in Section 5.4.

current or sequential, necessarily brings the local timelines together. The result is a new timeline that is globally consistent with the timelines of both components. At this new level, the composition has a set of inputs and outputs and timeline of the composition is again local relative to these inputs.

The significant aspect of the Synchronous Languages is that there are many notions of time, each having its own progression. These timelines are related by structures in the language and by compositions of structures in the language. The RMC Barrier Theorem states that not all compositions are non-contradictory. As such every Synchronous Language compiler must perform computations which relate all the local timelines and ensures that there is a consistent global timeline. This takes the form of checking for causality, the lack of internal nondeterminism or that the timelines are infinite.

5.7.1.3 Projective Semantics

The granularity implied by the multiform notion of time illustrates a third aspect of the Synchronous Languages: that the macrostep semantics is defined in terms of microsteps. External observation of the system is defined only at macrostep instants and the intra-macrostep behavior is not directly observable. This allows one to make claims about a system's behavior as having an *instantaneous* response where inputs cause reactions and their outputs occur in zero time. Here the notion of *event* time is used in the declaration that inputs and outputs are simultaneous. They are simultaneous in event time but need not be simultaneous in metric time.

Gonthier has called the abstraction step which hides microsteps inside a macrostep a *quotient* operation.¹ The references in this presentation is to a *projective semantics*. This is the projection Π from the development of Chapter 3. Either the quotient or projection

1. *c.f.* Gonthier [295], Chapter 3.

analogy is appropriate because the sense is that finer structure of microstep ordering is abstracted away. The power of this abstraction comes from the fact that it is only the projected system which is actually constructed or is ever observed. The microstep system is but a mathematical construct whose shadow is seen in the actual implementation. This observation is important for it is used to justify the non-contradictory definition of the projection.

5.7.1.4 Concurrency

Synchronous languages are fundamentally concurrent. At the very least there is concurrency between the system and its environment in the form of the instant-to-instant macrosteps. More commonly there is intra-system concurrency where the total system behavior is defined in terms of a number of logical tasks, each of which is operating concurrently in synchrony. In a control-centric view the tasks take the form of processes which coordinate with each other by means of events and reactions just as they would with the environment. In the data-centric view, the tasks take the form of mutually dependent recurrence equations.

5.7.1.5 Determinism

The final defining aspect of the Synchronous Language semantics is that they are completely deterministic. Concurrency in Synchronous Language semantics is deterministic, and is a semantic feature which is fundamental in its own right. This sort of deterministic concurrency is distinct from the concurrency in the ASM model for example. There, concurrency is a derived feature being interpreted as the nondeterministic interleaving of different threads of control. Determinism in Synchronous Language semantics is important because it guarantees that for the same input stream the system gives the same output stream. This is a critical aspect because it means that implementations are predictable. It is also important in the context of verification because it allows *selection nondeterminism*¹

to be interpreted in a declarative way as a form of modular abstraction. This is in contrast with *ordering nondeterminism* which is not modular by definition. Selection nondeterminism declares that there are multiple sensitizable paths away from a state though the order of δ -steps along each of those paths, once commenced, is completely determined. Ordering nondeterminism on the other hand corresponds directly to the modelling of concurrency with nondeterministic interleaving as per the ASM model.

In this sense, selection nondeterminism is declarative because it is not an effective (directly executable) operation in the semantics. It is meaningful in the sense of declaring multiple behaviors in compact form. For example, a property that states that “P’s occur after 5 Q’s” can be abstracted to one in which “P’s eventually occur after some Q’s.” This results in a reduction from a deterministic ten-state system to a nondeterministic two-state system. This example is illustrated in Figure 5-6. A distinction arises between the two sorts of nondeterminism only in the case of non-abstract semantics where each step is broken down into multiple microsteps.

The key point in Synchronous Language semantics is that nondeterminism is an orthogonal semantic feature. It has a specific interpretation and is invoked at the explicit discretion of the programmer

5.7.1.6 Focus

The preceding sections presented the high-level philosophy of the Synchronous Language semantics. Within these broad constraints Synchronous Languages have been defined which present imperative control flow, graphical StateChart-style and recurrence equation dataflow styles of description. Too, as the examples in the previous section have illustrated, there are a number of proposed formalisms which are *synchronous-like* in the

1. Gajski et al. [277], page 83.

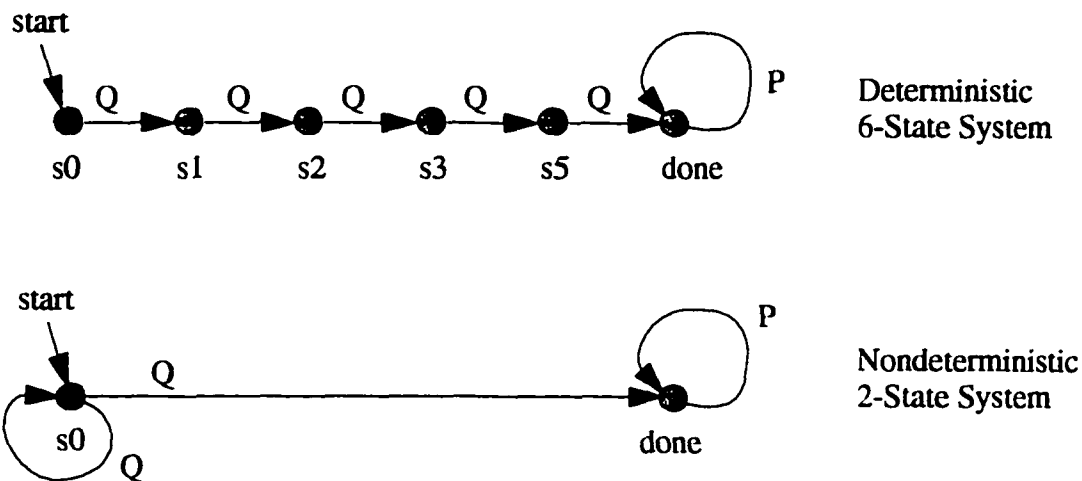


Figure 5-6. Selection Nondeterminism Supports for Modular Abstraction

sense that they fit the adjectival descriptions but which for one reason or another are not fully within the class of Synchronous Languages. It was also pointed out that the breadth of Synchronous Languages somewhat contributes to this confusion by making it difficult to identify what is common among the Synchronous Language semantics and how they differ from other formalisms. A unifying definition is needed.

5.7.2 The SL Languages

Earlier in this section a working definition of Synchronous Language semantics based on the RMC Barrier was proposed. In contrast, Benveniste and Le Guernic have attempted a characterization of Synchronous Language semantics in the Denotational Theory of Synchronous Communicating Systems [64] [65]. That theory is based on a clock calculus, called the Ω -calculus, and a suite of six instructions which are used to define the mini-languages SL_k . These languages are shown to be increasingly expressive ($SL_i \subseteq SL_{i+1}$). More importantly though SL_2 , SL_3 and SL_4 are shown to be associated with the semantics of Lustre, Esterel and Signal respectively.

This theory is interesting because offers a characterization of synchronous semantics based on Kahn's very general causal process model [414] [415] rather than on a theory of fixed points of finite transition relations. As well as being effective for Synchronous Languages such as Signal where clocks are explicit artifacts, the theory provides a powerful explanation for the differing attributes of the actual languages. In particular the necessity of trans-microstep consistency in a macrostep is explained while at the same time preserving the non-observability of intra-microstep behavior. For completeness, this section reviews the denotational theory omitting much of the notational machinery and explicit proofs in the interest of succinctness.

5.7.2.1 Synchronous Systems

A synchronous system is defined by tuples of values indexed by a discrete time index

$$\begin{aligned} X_t &= (x_t^{(1)}, x_t^{(2)}, \dots, x_t^{(n)}) \\ Y_t &= (y_t^{(1)}, y_t^{(2)}, \dots, y_t^{(n)}) \end{aligned} \quad (\text{Eq 5-10})$$

The domain over which the $x^{(i)}$ and $y^{(j)}$ are defined is left unspecified in this summary presentation. Its structure is not material to the presentation here. It need only be finite. The inputs to a system up to time t is defined in terms of a sequences of these tuples as

$$X_{1\dots t} = (X_1, X_2, \dots, X_t) \quad (\text{Eq 5-11})$$

An externally observable behavior is defined as a map associating input sequences to their corresponding output sequences.

$$\Phi = \{ (X_{1\dots t}, Y_{1\dots t}) \mid t \in \omega \} \quad (\text{Eq 5-12})$$

This model of behavior is an extremely general one for all it requires is that input initial segments $X_{1\dots t}$ imply output initial segments $Y_{1\dots t}$. Such maps Φ are referred to as *causal processes* because their defining property is that the output initial segments do not anticipate the inputs. This is the very definition of causality.

5.7.2.2 Six Primitive Instructions

The SL Languages are defined in terms of a suite of six primitive instructions derived in large measure from the Signal language. In combination they generate different subclasses of causal processes Φ . The primitive instructions are defined in Figure 5-7.

Rule	Name	Instruction	Semantics
o	flow	$P(x^{(1)}, x^{(2)}, \dots, x^{(n)})$	$\forall t \in \omega. P(x_t^{(1)}, x_t^{(2)}, \dots, x_t^{(n)})$
i	register	$y = u \rightarrow \$x$	$y_1 = u \quad \forall t > 1. y_t = x_{t-1}$
ii	condition	$y = x \text{ when } b$	$\forall t \in \omega. y_t = x_t \quad (x_t \equiv \text{present}) \wedge b_t$
iii	oversample	$y = \text{mux } c$	$\forall t \in \omega, c \in \mathbb{N}, 0 \leq i < c. y_{t+i}$
iv	merge	$y = u \text{ default } v$	$\forall t \in \omega. y_t = \begin{cases} u_t & (u_t \equiv \text{present}) \\ v_t & \text{else} \end{cases}$
v	concurrency	$P \parallel Q$	Operation of P and Q on consistent clocks

Figure 5-7. The Primitive Instructions of the SL Languages

The rule (o) defines a flow in terms of an instantaneous relation P over the streams $x^{(i)}$. The meaning of (o) is the direct extension of the instantaneous P over the time index t . It could be phrased less generally as $y = f(x^{(1)}, x^{(2)}, \dots, x^{(n)})$ which makes plain its combinational properties. The relational aspect is relevant when synchronization is specified. In particular the relational notation **synchro** $\{x^{(1)}, x^{(2)}, \dots, x^{(k)}\}$ is used to specify that $x^{(1)}, x^{(2)}, \dots, x^{(k)}$ share the same clock. The primitive instructions shown in Figure 5-7 are self explanatory save for (iii) which is the subject of the next section.

Using these six primitives, the languages SL_k are defined in terms of combinations of the instructions according to the rules of Figure 5-8. What is plain from the table is that $SL_0 \subseteq SL_1 \subseteq SL_2 \subseteq SL_3$ because each successive class of mini-language allows more instructions than the previous. The relationship that $SL_3 \subseteq SL_4$ is shown in a proof that

the features of the *mux* instruction are subsumed by those of the *default* instruction. How this embedding is carried out is explained in Section 5.7.2.6.

Language	Rule					
	o	i	ii	iii	iv	v
SL_0	✓	-	-	-	-	✓
SL_1	✓	✓	-	-	-	✓
SL_2	✓	✓	✓	-	-	✓
SL_3	✓	✓	✓	✓	-	✓
SL_4	✓	✓	✓	-	✓	✓

Figure 5-8. The SL Languages

5.7.2.3 Oversampling as a Multi-Dimensional Time

The *mux* instruction, rule (iii), produces in y a clock that has c occurrences for every single occurrence of c . The *mux* instruction defines the *oversampling* of the clock of c according to the value of c . Operationally, the *mux* instruction can be thought of as defining a counter whose initial value is the value on c when c appears and which decrements at each successive microstep. The succeeding macrostep is reached when the counter reaches zero. The *mux* is the primitive responsible for the definition of microsteps as found in the control flow of Esterel.¹

Without oversampling, the model of time defined by Eq 5-12 is one-dimensional, discrete and is isomorphic to the natural numbers \mathbb{N} . The significance of the rule (iii) is that it induces a ragged multi-dimensional structure in $\mathbb{N} \times \prod_{i=1}^k \mathbb{N}_i$ onto time as shown in Figure 5-9. The embedding is multi-dimensional instead of two-dimensional because every instance of multiplexing creates a two-dimensional temporal structure $\mathbb{N} \times \mathbb{N}_\delta$, each of which are incomparable save for their first dimension. The oversampling *rate* is taken

1. Actually this claim is left as a conjecture. *c.f.* Benveniste and Le Guernic [64] in both Section 4.2.4 and Chapter 5. The reasons for this are dealt with in Section 5.7.2.6 of this work.

from the value of c which is by definition a finite set so there can be no infinite oversamplings. The macrostep timeline of Eq 5-12 corresponds to the elements in $\aleph \times \prod_{i=0}^k \{0\}_i$.

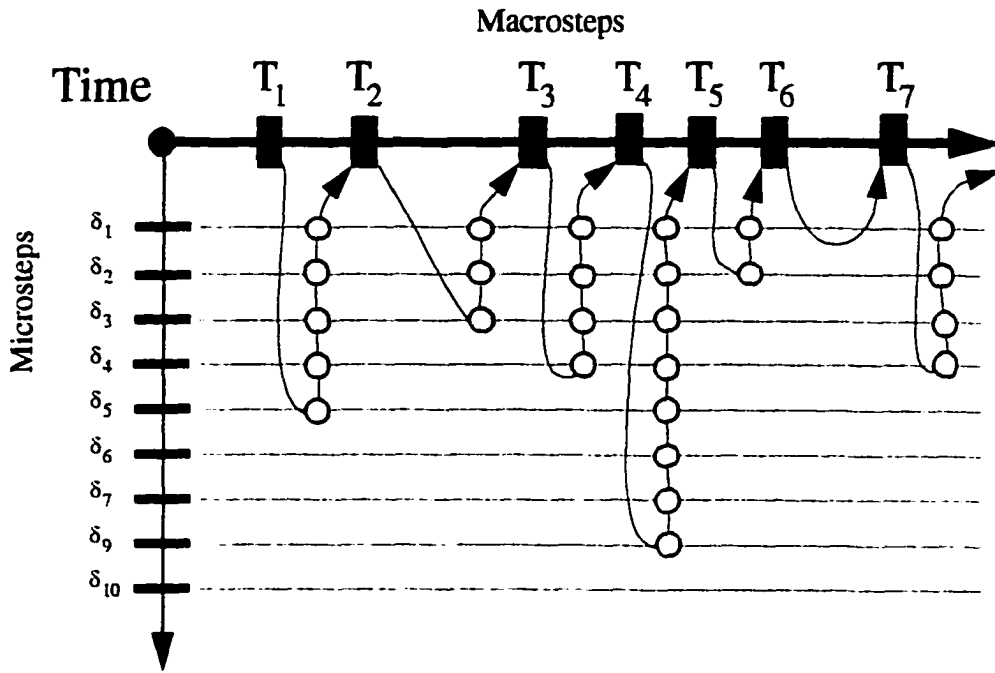


Figure 5-9. The Multi-Dimensional Structure of Time under Rule (iii)

5.7.2.4 The Ω -Calculus of Clocks

Within this framework, the clock calculus of a process (an SL_k program) is defined. A clock calculus is an abstraction of actual behavior that focuses exclusively on the synchronization aspects of the computation. The abstraction is simply that a boolean clock is **true**, **false** or **absent** in an instant with the obvious information ordering of presence being greater than absence. The clock calculus of a process can be thought of as the synchronization skeleton of that process.

The *algebraic clock calculus* is a completely formal system in which a set of manipulations and reductions are defined. These completely formal operations on expressions in the algebra can be interpreted as predictions about the potential behavior of the process in

the same way that algebraic operations on differential equations can be said to predict the potential behavior of a circuit. An algebraic clock calculus is defined only for the class of *totally observed* processes where two conditions hold:¹

1. no clock or signal that contributes to the state of the process has been masked,
2. the basic clock dominates (is faster than) all other clocks in the process.

The basic clock is the clock that is always present at every macrotime step. It corresponds to **tick** in Esterel. The first condition clearly relates to the expressibility of the process' state in the algebra. The second condition is a requirement that time be linear. The further ramifications of these requirements are returned to in Section 5.7.2.6.

In this case the algebraic clock calculus is defined relative to the three-element field over the integers $\{-1, 0, 1\}$ which is denoted by $Z/3Z$. These integers are interpreted as **absent** is 0, **true** is 1 and **false** is -1 . Presence is interpreted as either **true** or **false**. Constraints are represented in this system by quadratic polynomials over $Z/3Z$.

- $b^2 = 1$ requires that the signal b is present at every time step,
- $y^2 = x^2$ declares that both y and x must both be present or both be absent at once and that x and y have the same clock,
- $y^2 = x^2(-b - b^2)$ declares that y is present only when b is **true**.

The algebraic clock calculus of a process with n signals is shown to be given by functions $\aleph \rightarrow Z/3Z^n$ mapping time points $t \in \aleph$ onto points in the state space $\xi \in Z/3Z^n$. In this regime a process' behavior is a dynamical system given by:

$$P \subseteq Z/3Z^n \times Z/3Z^n \text{ which is a transition relation,}$$

$$I \in Z/3Z^n \text{ which are the initial conditions}$$

Equivalently P can be viewed as a set of k polynomial equations viewed as constraints:

1. In the original presentation [64], the condition of total observability is formally defined in terms of time-ordered sequences Π of information flows $\sigma(A)$ over variables A . The informal definitions are used here because the extra notational machinery is not of direct use in this presentation.

$$P_1(\xi_p, \xi_{t-1}) = 0$$

$$\dots$$

$$P_k(\xi_p, \xi_{t-1}) = 0$$

The trajectories of the system in $Z/3Z^n$ are those which satisfy all of the P_i . With the system synchronization skeleton in this form there are computationally effective methods for causality checking [65].

5.7.2.5 The Denotational Semantics of SL_k

The denotational semantics for the languages SL_k is given in terms of the Ω -calculus. In the usual style of a denotational semantics, the denotations of processes as sequences is defined in a syntax-directed fashion. The denotation of the whole process expression is defined in terms of the composition of the denotations of its parts: the denotation of a program is given by a valuation function:

$$M[[program]] = \left(\begin{array}{c} EPROC[[program]] \\ SYNCH[[program]] \\ VAL[[program]] \\ ALGCLOCK[[program]] \end{array} \right)$$

where:

program is an SL_4 program.

$EPROC[[program]]$ gives the denotational extension of *program*,

$SYNCH[[program]]$ specifies the external clock relationships that must hold.

$VAL[[program]]$ specifies the external value relationships that must hold.

$ALGCLOCK[[program]]$ is an abstraction of *program* that summarizes its synchronization properties.

The denotation of *program* is completely determined by the first three components of $M[[program]]$. $EPROC[[program]]$ expresses the denotation of *program* in terms of sequences of mappings (X_t, Y_t) defined over macrotime steps t by means of a syntax-directed decomposition. The terms $SYNCH[[program]]$ and $VAL[[program]]$ respectively give the environmental dependencies of *program*.¹

The fourth component, $ALGCLOCK \llbracket program \rrbracket$, is the material representation of the algebraic clock calculus for $program$. It is the set of polynomials over $Z/3Z^t$ which are the constraints on $program$. This is actually the significant field for any effective means for deciding the well-definedness of a $program \in SL_4$ must manipulate polynomials in $Z/3Z^t$ implied by $ALGCLOCK \llbracket program \rrbracket$.

Conveniently, the semantic valuation function $M \llbracket program \rrbracket$ has the property that the algebraic clock constraints implied by a concurrent composition are just the union of the constraints implied by each of the components:

$$ALGCLOCK \llbracket P \parallel Q \rrbracket = ALGCLOCK \llbracket P \rrbracket \cup ALGCLOCK \llbracket Q \rrbracket$$

This union rule implies that the algebraic clock calculus of a system is defined whenever the algebraic clock calculus of all its leaf instructions is defined. This definedness condition is the kernel of the difference between SL_3 and SL_4 .

5.7.2.6 Observability in SL_3 and SL_4

The denotational definition of the mux instructions offers an interesting insight into the difference between SL_3 and SL_4 . By extension these differences highlight the deep differences between Esterel and Signal. The previous section gave the semantic valuation function $M \llbracket program \rrbracket$ in terms of a tuple of four components, the first three of which completely characterize the denotation of $program$, while the fourth component, $ALGCLOCK \llbracket program \rrbracket$ was a formal abstraction that provides a practical means for reasoning about the properties of the denotation. In Section 5.7.2.4 the algebraic clock calculus was defined for totally observed processes. This restriction to total observability

1. The full presentation of these fields and the metalanguage used to express them can be found in Benveniste and Le Guernic [64] [66]. A presentation that is specific to the Signal language because it excludes rule (iii) can be found in Benveniste and Le Guernic [65]. The elaboration of the metalanguage is elided here because the focus is on *when* a synchronous semantics is well-defined rather than the specific details of *how* it is defined. The interest here is on the definedness of the algebraic clock calculus based on the referenced theorem that an algebraic clock calculus exists only for totally observed processes.

offers the possibility that a well-defined denotation for a *program* may exist yet it could be that there is no algebraic clock calculus for it. Such is the case in SL_3 with its definition of oversampling.

The observability of the mini-language SL_3 can be understood by studying rule (iii) and its nonlinear effect on time relative of the definition of observability. SL_3 violates total observability as defined in Section 5.7.2.4 on both counts. The generated oversample clock y is suppressed from the macrotime view. This violates the first condition. The basic clock, present in every macrotime step, does not dominate every oversample-generated clock y . Neither can it be stated that every oversample-generated clock y dominate the basic clock. Thus there is no fastest clock. This violates the second condition and so it must be concluded that there is no $ALGCLOCK[[program]]$ for an arbitrary $program \in SL_3$.

In contrast, the other three items in $M[[program]]$ are well-defined for SL_3 because they involve well-defined operations on clocks. Clocks are sequences defined on some domain containing an element \perp which is interpreted as absence. Rule (ii) is defined in terms of the projection and intersection of such sequences and rule (iii) is defined in terms of finite insertion into such sequences. Thus every $program \in SL_3$ has a well-defined denotation yet has no $ALGCLOCK[[program]]$.

Informally, SL_4 has the property of total observability whereas SL_3 does not because of the differing models of time relative to Eq 5-12. In SL_4 time is linear and all clocks and signals have valuations in instants $t \in \aleph$. In SL_3 time is non-linear, having two dimensions, and only some signals and clocks have valuations at instants $t \in \aleph \times \{0\}$. Those signals and clocks which do not have valuations on the macrotime axis are suppressed from the sequences reported in Eq 5-12. These internal properties are not expressed in relationships visible from the axis $\aleph \times \{0\}$ so there is no single clock that can be used

linearize time as required by Eq 5-12. The linearity of time is fundamental here and relates directly to the proof that oversampling can be rederived in a totally observable framework, namely that $SL_3 \subseteq SL_4$.

5.7.2.7 Containment of SL_3 in SL_4

The containment relationship $SL_0 \subseteq SL_1 \subseteq SL_2 \subseteq SL_3$ is clear by inspection the rules defining the languages SL_k which are given in Figure 5-8. What is not so clear is the relationship between SL_3 which includes rule (iii) but not (iv) and SL_4 where the reverse is true. The relationship $SL_3 \subseteq SL_4$ is shown by reconstructing the **mux** instruction using the **default** instruction of SL_4 . The **multiplexer** program in SL_4 which performs the **mux** operation of rule (iii) is shown in Figure 5-10.¹

```

multiplexer ::= [
    u = zu - 1
    ||
    zu = raz default past_u
    ||
    past_u = u_0 ->$ u
    ||
    raz = C when past_stop
    ||
    past_stop = true ->$ stop
    ||
    stop = (u = 0)
    ||
    synchro C, raz
]

```

Figure 5-10. The Redefinition of **mux** in SL_4

The proof that the **mux** instruction of rule (iii) can be embedded into **multiplexer** involves constructing its denotation according to $M[\llbracket \mathbf{multiplexer} \rrbracket]$. Intuitively the proof follows from the fact that the behaviors allowed by rule (iv) are a superset of those allowed by (iii) because in rule (iv), the clock of **x** is faster than the clock of either **u** or **v**.

1. From Benveniste and Le Guernic [64].

The same is true in rule (iii) where the clock of \mathbf{x} is faster than that of \mathbf{c} .

From the components *SYNCH* [*multiplexer*] and *VAL* [*multiplexer*] (written in the metalanguage) it is shown that the signal \mathbf{u} occurs as many times as the value of \mathbf{C} between occurrences of \mathbf{C} .¹ The performance of *multiplexer* is effectively illustrated by the chronogram of Figure 5-11. There are three significant points in that diagram. The first is that the signal \mathbf{u} is shown to occur the same number of times as the value of \mathbf{C} when \mathbf{C} occurs. Thus $u = \mathbf{multiplexer} \ C$ behaves the same as $u = \mathbf{mux} \ C$. Secondly, it can be observed by inspection here that time is linear and the clock of \mathbf{u} can be identified as the fastest clock, in fact it is exactly as fast as the basic clock. The third point is that \perp is used as a filler to denote the absence of a signal in an instant. The signals \mathbf{C} and \mathbf{raz} are shown to be often absent. This principle extends in the sense that when *multiplexer* is used in the context of a larger program an arbitrary number of \perp filler events may be inserted between occurrences of \mathbf{u} . In such a case some other clock would be the fastest clock.

signal	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}	t_{14}	t_{15}	t_{16}	t_{17}	t_{18}
\mathbf{u}	2	1	0	4	3	2	1	0	6	5	4	3	2	1	0	3	2	1	0
\mathbf{zu}	3	2	1	5	4	3	2	1	7	6	5	4	3	2	1	4	3	2	1
$\mathbf{past_u}$	\perp	2	1	0	4	3	2	1	0	6	5	4	3	2	1	0	3	2	1
\mathbf{raz}	3	\perp	\perp	5	\perp	\perp	\perp	\perp	7	\perp	\perp	\perp	\perp	\perp	\perp	4	\perp	\perp	\perp
$\mathbf{past_stop}$	t	f	f	t	f	f	f	f	t	f	f	f	f	f	f	t	f	f	f
\mathbf{stop}	f	f	t	f	f	f	f	t	f	f	f	f	f	f	t	f	f	f	t
\mathbf{C}	3	\perp	\perp	5	\perp	\perp	\perp	\perp	7	\perp	\perp	\perp	\perp	\perp	\perp	4	\perp	\perp	\perp

Figure 5-11. A Chronogram Showing *multiplexer* in Action

1. The exposition of the proof requires the presentation of the semantic metalanguage of clocks which has not been developed here. The mechanics of the proof is not material to the presentation here, however the proof can be found in Benveniste and Le Guernic [64], Section 4.3.1.

5.7.2.8 Summary

A subtle shift has occurred between SL_3 and SL_4 . In the former case, the baseline time units are macro instants T_i whereas in the latter case the baseline unit is the microtime instant t_i . In SL_3 , the **mux** instruction produces clocks that are faster than the macrostep identity clock that is present at every T_i . These faster clocks are finitary and were given the label δ_j in Figure 5-9. In the macrostep projection of time onto $\aleph \times \{0\}$ these transitory clocks are elided resulting in an incompletely observed process. In SL_4 every instruction produces clocks that are no faster than the microstep identity clock which is present at every t_i . In the context of Figure 5-9 time is linearized by following the causal paths through $\aleph \times \prod_{i=1}^k \aleph_i$ and mapping this timeline onto \aleph . Thus the example of the figure can be redrawn linearly as shown in Figure 5-12.

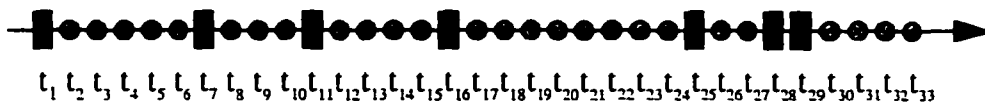


Figure 5-12. The Linearization of Time under Rule (iv)

The denotational theory of the Synchronous Languages hinges around this linear model of time. It is useful to think of macrotime and microtime as being “slow time” and “fast time” respectively. The difference between SL_3 and SL_4 can be summarized in terms of a choice between a baseline in slow macrotime with the fast microtime being derived and unobservable or a baseline in fast microtime with slow macrotime being derived using explicit markers denoting its passage.

5.7.3 Synchronous Programming Languages

The denotational theory of the Synchronous Languages one can describe the synchronous programming languages in terms of their overall attributes and an association with a

particular mini-language SL_k . The following paragraphs summarize the four Synchronous Languages within this framework.¹

Lustre

Lustre is a functional language. The basic construction in the language is the node which defines a set of dataflow equations mapping input sequences to output sequences. In this sense Lustre is a pure dataflow language. It is also purely reactive in the sense that every output can be directly traced to an input occurring within the same instant. No output sequence occurs faster than any input sequence.

Lustre is related to SL_2 with the modification that in rule (ii), for $y = x$ when b the clocks for x and b must be the same. The important effect of this modification is that the clock tree can be determined from a analysis of the program communication graph. This is the reason why Lustre is said to surpass the RMC Barrier through a static structural restriction to *RMC*. This restriction basically amounts to checking that the equation dependency graph is acyclic.

Signal

Signal is a language in the functional style. It is however a relational language in the sense that the context in which outputs are used can affect the rate at which inputs are consumed. The language designers view Signal as a sort of programming by constraint with the program behavior being the intersection of all the constraints. The abstract form of these constraints is *SYNCH* [[*program*]] and *VAL* [[*program*]]; their material form is given in *ALGCLOCK* [[*program*]]. Signal is not a reactive language in the sense that it

1. The characterization of Lustre, Signal and Esterel draws directly from the conclusions in Benveniste and Le Guernic [64], Chapter 5. The association of Argos with SL_1 is novel to this presentation.

Other relevant information about these languages such as their syntax, implementation strategies and application focus can be found either in addition to the original publications cited on page 239 or in tutorial presentations of these languages can be found in Halbwachs [320] or the *IEEE Special Section on Another Look at Real-Time Programming* [62] [101] [321] [465].

is possible to write programs in which the output occurs faster than the input.¹ It is however always possible to choose the fastest clock as the identity clock.

Signal corresponds almost directly to SL_4 . In fact, SL_4 can even be viewed as a sublanguage of Signal. Synchronization constraints in the language are derived from both the control and data dependencies in the equations. This gives Signal programs fine-grained control over the computation. This generality has a cost in that it is possible to write contradictory Signal programs which can only be detected from a dynamic analysis of the program behavior. Signal surpasses the RMC Barrier by a dynamic restriction to *RMC*. The *ALGCLOCK*[[*program*]] provides a formalism for this dynamic analysis.

Esterel

Esterel is an imperative language. The basic constructors in the language are sequential and parallel composition which are resolved in a deterministic fashion. The fundamental semantics of the language is given in terms of event derivatives on the language interpreted as process algebra. Esterel programs have finite state so there are be a finite number of such derivatives. The derivatives are associated with the states of an automaton thus giving direct interpretation of Esterel programs in terms of automata. Later developments have produced less obvious interpretations based on hierarchical abstract machines and synchronous circuits. The RMC Barrier is approached with restriction to *RMC* within the reachable transitions of the automaton.

Esterel programs map input sequences to output sequences as do all synchronous programming languages. They are purely reactive because and all synchronization constraints can be expressed in term of relationships between the input and output ports. No output event may occur without a direct input cause in the same instant. Though fully reactive, the interpretation of Esterel programs as causal processes has been shown to be

1. The *multiplexer* program of Figure 5-10 is one such example.

only partially observed. The behavior within the trans-microstep definition of a macrostep is hidden in the sense that observation is only allowed at the end of an instant and at the beginning of an instant (the end of the previous instant). For these reasons it is not possible to choose any fastest clock in an Esterel program. Esterel is said to be related to SL_3 because the *mux* of rule (iii) clearly has relation to the intra-instant serial composition operation.¹

Argos

Argos is a state-centric language with essentially the same set of notational features as StateCharts [330]. In contrast however with the *ad hoc* semantics shown in Example 4.3.4.6 the Argos semantics are synchronous. The RMC Barrier is addressed with a direct analysis of the transition structure of the implied automaton; only self-consistent descriptions are admitted. Argos has no support for data variables or a sequential constructor. Thus causal processes modeling the same behaviors have total observability. Argos programs are simply finite state machines with inputs, state and outputs, all sharing the same central clock. The direct association is with the mini-language SL_1 .

5.8 Communicating Reactive Processes (CRP)

The semantics of the Synchronous Languages offers wide possibilities for language design. Not only are there the dimensions of style, state-based versus imperative versus data-flow, but there are also choices between partial and total observability. The Synchronous Language semantics however are fundamentally $RM\bar{C}$ and do not provide direct support for the control of asynchronous, long-running or multi-instant operations. Such has been found to be useful in the control of long-running tasks from within the synchro-

1. Again, this is left as a conjecture *c.f.* Benveniste and Le Guernic [64], Chapter 5. There are other subtle differences between Esterel and SL_3 which are glossed over in that analysis as well. Among these is that data-centric computations in Esterel are unclocked and permanent across instants; variable declarations do not correspond directly to the register rule (ii) and there is no way to specify synchronization via variables using rule (i) and *synchro*. Such constraints can be expressed in Signal which is expressible in SL_4 .

nous framework [576]. Berry's Communicating Reactive Processes (CRP) [83] addresses this by mixing the synchronous reactive semantics of Esterel with the asynchronous trace-based semantics of CSP. The CRP semantics has two largely separable levels with $R\bar{M}\bar{C}$ holding at the lower level and $\bar{R}M\bar{C}$ holding at the upper level. This blending of CSP and Esterel is illustrated in Figure 5-13.

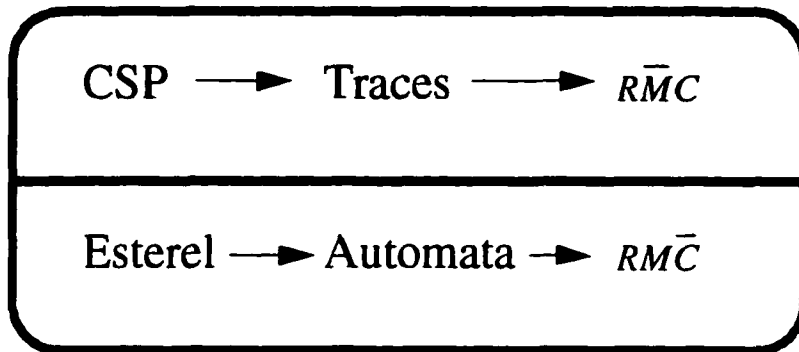


Figure 5-13. The Separated Semantics of CRP

The CRP semantics can be seen as an instance of Huizing and Gerth's separated semantics as presented in Section 4.4.2. They proposed addressing the RMC Barrier with a two-level scheme in which a different two-of-three choice among R , M and C is made at each level. Their particular preference however was for $R\bar{M}\bar{C}$ at the low level and $\bar{R}M\bar{C}$ at the high level.

In CRP the material and theoretical presentation of the language and terminology of CSP is introduced into the Esterel process algebra with a synchronizing **rendezvous** instruction operating over bidirectional channels. There are two parts to a rendezvous, the sending part and the receiving part. For synchronization and communication to occur, one process must be executing the sending phase of the rendezvous and at the same time a second process must be executing the receiving phase. Either process blocks until this condition holds and at that point the value is transferred between the two processes and they are

said to share an instant. This is exactly consistent with the notion of synchronizing communication in CSP. In CSP channels are not a broadcast medium. They are buffered so there is a some extra machinery which provides for the serialization as well as the ability to abort a pending rendezvous request based on a synchronously-supplied signal.

In CRP there is a distinction between small and large coordination schemes. “In the small,” synchrony and its determinism is both a direct benefit can be reasonably managed. The synchronous semantics does not come without some cost as a verification that the system description has the *C* property and the perfect synchrony hypothesis holds at the implementation level must be performed. At the upper level of the semantics, “in the large,” the asynchronous trace-based semantics of CSP applies. This structure has the benefit of allowing the Synchronous Language semantics to manage coordinating components which are tightly coupled, a task for which they are well suited, yet retaining the freedom afforded by the asynchronous interleaved semantics in managing loosely coupled executions.

5.9 Review

Each of the previous seven sections developed an example of applied semantics within the framework of the RMC Barrier and the means for surpassing it. The examples started from the simplest possible concurrency scheme, Asynchronous Shared Memory and progressively explored the possible ways of adding different features and kinds of structure to the semantics. The examples reached their height of expressive power in the Communicating Sequential Processes and the Clarke Languages of Section 5.5 and Section 5.6. There it was shown through certain refinements and restrictions of the two formalisms that there are language features which cannot be present in a semantics if it is to remain decidable. Decidability, or rather its lack, establishes a strong upper limit on the expressiveness that can be tolerated in a semantics oriented a formal methods of synthesis and

verification over and above the limits defined by RMC Barrier Theorem.

The semantics of Synchronous Languages, an RMC semantics which admits only systems which happen to have C , is shown to offer as much expressiveness as can be tolerated without crossing the line into undecidability. The microsemantic analysis of Example 4.3.4.4 showed that the Synchronous Language microsemantics was distinguished in Ex 4.3.4.4-2 by microstep enabling conditions $E_{\delta_i}^{(\Phi)}$ which are defined by the outputs produced across the (state-dependent) full path $\hat{\delta}$. An alternative characterization of Synchronous Language semantics was given in Section 5.7.2 with the Denotational Semantics of Synchronous Communicating Systems. That theory, which is completely independent of the RMC Barrier Theorem, highlighted the notion of trans-microstep observability and its relationship to the linearity of time as the key distinguishing feature of Synchronous Language semantics.

Finally Berry's Communicating Reactive Processes were presented as an instance of a separated semantics which potentially offers the best of both worlds. The predictability and determinism of RMC Synchronous Language semantics is appropriate for tightly coupled coordinating subsystem at the fine level. At the coarse level the trace-based RMC semantics of CSP naturally supports nondeterministic and asynchronous communications of loosely coupled coordination schemes.

6 System Description Languages

The previous chapters exposed the contributions and limitations of semantic models in the description of systems. There the focus was exclusively on the underlying mathematics with an emphasis on theoretical constructions and notions of expressibility or the lack thereof. It remains to relate these theoretical properties in semantic models to proposed and existing specification languages.¹ In the abstract, establishing this relationship in a proposed language is properly the design of the language itself. More concretely in the case of an existing language, establishing the semantic basis of the language in the foregoing theoretical framework is essentially a form of *post hoc* analysis or criticism given the clarity of hindsight. In its most ambitious form, language design and analysis requires tremendous breadth encompassing such issues as the specifics of the syntax, the mechanics of the type system, the practicalities of implementations and a host of other issues.² What is needed for the treatment here is quite a bit less aggressive and indeed more subtle: a general viewpoint which at once highlights the relationship of languages to their semantic

1. As explained in Chapter 1, the extra qualifier *specification* or *programming* modifying the noun *language* is adopted where necessary to distinguish this usage from that of the set-theoretic structure describing the set of possible behaviors of the system.

2. A useful survey of programming languages which includes a substantial bibliography is Salomon's taxonomy [635] which is based on the four dimensions of programming language independence: 1. the target machine (more generally the usage architecture), 2. the problem domain, 3. human aspect (user qualifications), 4. concepts of time (in the sense of program lifetime).

A treatise concerning trade-off issues in language design is Hilfinger's [348] analysis of Ada.

model while at the same time abstracting away the particular idiosyncracies of syntax and implementation. Additionally there should be some way to use the analysis developed to predict future developments in languages, or more pragmatically to highlight the deficiencies of current languages in such a way that new solutions can be proposed.

One way to understand language design in a very broad way without becoming bogged down in the actual details of any one language instance is to approach the analysis from an evolutionary perspective. In this light, a language is a manifestation of the application interests and implementation understanding available at its time of creation. It is a snapshot which captures one slice of a continuing stream of changes in application domains and implementation techniques. Taken in context therefore the evolutionary point of view is explicitly historical, focusing on the comparison and contrast of *between* languages rather than on the specific features in any one instance. Conveniently it allows the strengths and weaknesses of languages to be analyzed across broadly-defined generations. Strengths are shown to be replicated to succeeding generations while weaknesses are seen to die out as solutions are adopted or as the underlying application domain changes to avoid them. Finally the evolutionary perspective gives some sense of the future direction of languages by highlighting both trends over time and also persistent problems.

The application area relevant to this work is the specification of synchronous finite-state systems specified by software-defined models. From the executable model, implementations are derived in either in hardware or software either automatically or by manual reimplementation. The following sections examine the evolution of system description languages concentrating on hardware description languages (HDLs) as executable specifications of system behavior.¹ There are two main thrusts to the argument presented. The first is a historical one which explains how HDLs arrived at their current state and in fact

1. Though the treatment here is on hardware description languages, there has been interest in using HDL specifications to describe embedded software systems as well, *c.f.* Gajski *et al.* [277] or Chiodo *et al.* [169].

what that state is relative to the theoretical framework of the RMC Barrier Theorem and microsemantic analysis. The second is a projective one which identifies the weaknesses in the current generation of HDLs and offers a proposal for how the next generation of languages could better take advantage of the increased understanding of semantics provided by the theoretical framework. Weaknesses are characterized in terms of an intuitive notion the *distance* between the semantics of present-day HDLs standards and the semantics of the synchronous languages. The proposal aspect explores the opportunities and costs of minimizing and even removing this distance.

6.1 An Evolutionary View of Specification Languages

The uses of system descriptions have changed in important ways over the years. This in turn has led to changes in the sorts of features which are considered appropriate in the languages used to describe them. From the integrated circuit perspective, there has been the trend away from explicit modeling of the underlying physical effects like drive strength and charge-sharing, however, even within the realm of synchronous digital hardware descriptions, there has been evolution in the features and focus of system description languages. The ever-increasing complexity of system being designed has inexorably forced the adoption of more abstract models as specifications of system behavior. Too, with the advent of techniques such as technology-independent gate libraries and technology mapping there is much less of a need to describe what gates are in the design and more of a need to describe what it does.

The evolution of system description languages can thus be traced across three distinct application domains: simulation, synthesis and (formal) verification. The earliest and most basic use is in simulation which defines components operationally; the system is what the model executes. The second use is synthesis or resynthesis where the key issue is the determination of the mathematical function computed by the model. From that

abstract definition a different implementation of the description is created which optimizes some resource according to some given constraints. Most recently, there has been a resurgence of interest in formal verification techniques, driven largely by the development of implicit OBDD-based methods for the manipulation of large sets.¹ In turn this has spurred interest in the issues concerning the interfacing of system description language semantics and formal verification techniques. In fact, this work can be seen as a contribution in this last area.

The following sections present the argument that each of these domains has placed a largely orthogonal set of demands on system description languages. This has caused evolution both in language semantics but also in the problem domain itself. Further and more significantly though, this evolution process has progressed to the point where there is now nearly universal agreement that the needs of ultra high speed simulation, synthesis and formal verification are largely the same.

6.1.1 Simulation Orientation

The system description languages can be classified into two major classes, low level languages and high-level ones. The low level languages are simple single level descriptions used mainly for design interchange or directly for simulation. They are generally referred to by the term *internal representations* and are typically thought of as being the target of a higher-level compilation or synthesis process. One of the most common is a form of three-address code [10]. They have a form of syntax though it tends to have the non-textual form of a data structure. As well they have some notion of semantics so it is useful to describe them as simple languages.

The higher level consists of languages which are more reasonably characterized as sim-

1. An overview of the relevant results was presented in Section 3.3.4.

ulation control languages. Examples in this class are languages such as ADLIB [351], BDS [229], HARPA [707], Helix [202], HSL-FX [561] and iHDL [390] or the industry standard simulation languages VHDL [384] [387], Verilog [570] and UDL/I [404].¹ The common thread among all these languages is that they are all based around the activities of a set of simulator data structures such as an event queue, time wheel or delayed assignment block which are presumed to exist in any conforming implementation. The behavior of models described in these languages is operationally defined by the interaction of the model with the simulator event control.

Probably the most well accepted simulation scheme is the discrete-event paradigm. There the execution of model components cause events which then trigger other components to execute. A model of time is involved so that events can be triggered at points in the future. The power and generality of the discrete event model comes from the ability to support data-dependent delays. Conceptually it is also very simple. VHDL and Verilog are both discrete event simulation languages.

The advantage of the discrete-event paradigm lies in its generality. In the case of VHDL, the generality has allowed VHDL to be used for applications ranging from performance analysis [639] to device-level modeling at the switch level [201]. The more reasonable and mainstream uses involve the representation of combinational logic, sequential circuits such as pipelines and interacting finite state machines and simple netlists of blocks and library elements. The power of the discrete-event paradigm allows models written at various levels of abstraction and detail to be simulated through one extremely general event queue data structure and simulation cycle. The disadvantage of the discrete-event paradigm is that there is no presumed level of description so in a strong sense the

1. Formally, the semantics of UDL/I is defined by the NES model presented in Section 2.3.1. The NES model however has not been shown to lend itself to practical implementations. For realistic implementations therefore, the UDL/I language semantics, is defined by the traditional simulation algorithms.

meaning of the model as a specification is in the eye of the beholder.

6.1.2 Synthesis Orientation

From a simulation perspective what is important about a language is the features that it supports and how those features interact with the prescribed implementation. This perspective became problematic when the application domain widened to include not only simulation but also synthesis. From a synthesis perspective the focus is on determining what a model *means* in a mathematical model so that a better implementation of it can be selected. The rise of synthesis as an important application of languages was the first time when language semantics became truly significant in the design process.

As with the simulation-oriented languages one can distinguish both high and low levels in synthesis-oriented languages. As with the previous case, there has been little controversy over the form of languages at the lower level. Languages at this level are largely oriented at the description of logic networks at the gate-level including registers. Examples include the Berkeley Logic Interchange Format (BLIF) [649] or the Structural Logic Interchange Format (SLIF) [493]. There are other instances such as the Software/Hardware Intermediate Format (SHIFT) [167] or the Stanford University Intermediate Format (SIUF) [720] which are oriented at software synthesis and simulation respectively.

6.1.2.1 The Hardware Semantics

In contrast with the low level, there has been quite a lot of evolution at the high level. The original use of high level languages as specifications started with software programming languages (SPLs).¹ This original use could be called the search for *the hardware semantics* of SPLs because the research attempted to identify an interpretation for sequen-

1. Girczyc [286] made a distinction between Hardware Description Languages (HDLs) and Software Programming Languages (SPLs) in his use of Ada as a vehicle for specifying VLSI-level hardware.

tial programming language in terms of synchronous digital hardware. The hypothesis was that there was some vantage point from which an analogy between the execution of a program written in an SPL and executing *on* a general purpose computer and such a program executing *in* concurrent coordinating components of VLSI-level hardware. While one must be careful to distinguish research which was oriented at defining the high-level synthesis problem itself or at evaluating a specific approach to one of its subproblems there are a number of projects that explored the use of SPLs as specifications for synthesis. With this in mind one can identify attempts to use Pascal [691] [692], C [616] [145] and Ada [286] [287] [288]. One can even view Ku's Hardware C [445] [446] [447] as being part of this evolution.

Various reasons were proposed for pursuing the hardware semantics analogy approach. Among these reasons can be cited that no new language needed to be learned by designers, the development environments for SPLs were mature and actively maintained, on a related note, the development of a simulator for the "hardware"-interpretation of an SPL is trivial (*c.f.* Barbacci *et al.* [52]) and finally it might be convenient to develop the hardware specification and the software which runs on it in the same language. Given the wisdom afforded by hindsight, the identification of a hardware semantics analogy for SPLs can be seen as being fairly simplistic. Despite the alleged advantages, the fact remained that SPLs lack an intuitive fit for the system specification problem. Today it is commonly agreed the three factors which are missing are a model of concurrency, a model of time and the exposition of constraints.

On counterpoint to the SPLs, functional languages were proposed as appropriate for high-level system specifications. Here the analogy between the software model and the hardware could be argued to be more direct or even obvious: when the hardware implementation in combinational logic is functional then so should the software model specifying that implementation be functional. Languages in this class included Daisy [407], Ella

[540] [541] [542], μ FP [651], FHDL [524], STRICT [151], Silage [349], recurrence equations [562] and HML [466] [467].

6.1.2.2 Representations and Data Structures

It became obvious after a time that neither SPLs nor functional languages were the definitive solution to the specification problem. In response, the search changed from attempting to define a convincing analogy between the software specification and the hardware implementation to one of finding *the appropriate internal representation*. This representation was a hypothesized data structure, typically a graph or tableau, which could abstract away the superficial peculiarities of the description language and provide a suitable basis for the key applications of the era: both simulation and synthesis. The goal was to define a data structure which would organize the information from the high level language in a way that enabled the relevant scheduling, allocation and binding algorithms. Additionally, if the data structure could be shown to be canonical then the claim could be made that structurally different but behaviorally equivalent descriptions would turn out the same after the optimization steps. Too, by representing behavior in a neutral format such as a graph, it could be convincingly stated that the synthesis procedure was general enough to support multiple languages.

This change moved the focus away from the language level where the basic units are statements, expressions and control flow to a finer level of detail which exposed dependencies and allowed for the expression of various sorts of constraints. The internal representation approach to hardware semantics in a sense predates the analogic approach. It was first used by Snow [664] to define the meaning of ISPS [51] [53] programs in terms of the Value Trace (VT) [512]. Even so, the VT representation did not so much define a semantics so much as it enabled a certain class synthesis activities. In fact one claim of that early work was that it was reasonable and proper to approach the high level synthesis

scheduling and allocation problem purely in terms of algebraic constraint relations among the uninterpreted operation nodes in the graph-based representation [318] [319]. The constraints are solved by techniques such as mixed-integer linear programming and the result can be claimed to be optimal relative to the constraint conditions.

After this early work there was a gap with the bulk of the internal representation proposals being presented after the language-driven approach had substantially run its course. Among the many proposals were such representations as the Design Data Structure (DDS) [436] or the Behavioral Intermediate Format (BIF) [242] [244]. These representations featured not only the means for organizing the operations and operation ordering which is extracted from the raw parse tree, but they also addressed the issue of the bookkeeping needs of the scheduling, allocation and binding phases of high-level synthesis.

Schemes for high-level synthesis of VLSI-level hardware had always been patterned at some level after similar schemes for software compilation. It was therefore natural to adopt and adapt internal representations from software compilation into the hardware context. In the software compilation domain, the static single assignment (SSA) form [15] [625] and the program control dependence graph [261] came to be seen as the natural and appropriate internal representation for programs. This appropriateness was justified by the optimization and code generation algorithms which were enabled by the representation. The natural analogs of these forms for hardware synthesis domain were defined as well. Among these it can be mentioned that the Assignment Decision Diagram (ADD) [160] [161] and its more recent extension with the Condition Graph (CG) [412] [413] generalized the SSA form.¹ The Control/Data-Flow Graph (CDFG) [152] or the Control Flow Graph (CFG) [274] generalized the control dependence graph. Here too, the justification

1. An earlier use of single assignment occurred in Segal's [647] combinational logic synthesis from a subset of BDS [229]. The ADD form however makes a direct connection to SSA in the explicit representation of the ϕ -functions of SSA in the Condition Graphs.

for these forms was the optimization algorithms which were enabled by the representation.

6.1.2.3 Underspecification

One flaw seen in the graph representations was that they encoded what *must* occur in an implementation but failed allow for the expression of constraints or for underspecification. Proposals such as Operation/Event Graphs [16] [17] and Dataflow/Event Graphs [719] attempted to patch the flow graph model to address this deficiency. Other proposals for underspecification, the representation of timing slack and other optimization opportunities approached the problem from an automata-theoretic basis. The FSM Network Model [724], the p-Automata [235], Production-Based Specifications [645] [646], the Behavioral Finite State Machine (BFSM) [679] [725] and the High-Level Finite State Machine (HLFSM) [71] can be mentioned as examples of this approach. These last two, the BFSM and the HLFSM representations were ultimately adopted as part of a commercial policy-of-use for high level synthesis which governs the identification of degrees of scheduling freedom available in a given Verilog and VHDL model [437].

6.1.3 Verification Orientation

The common thread running through all of the synthesis-oriented internal representations is the concentration on the algorithms which are enabled by the representation. This is their justification. More recently the application domain of languages has been expanded yet again with the renewed interest in formal verification techniques. This interest is driven by new symbolic algorithms and representation techniques [137] which avoid the state explosion problem which plagued older methods. These methods are predicated on being able to characterize the system's execution in terms of discrete steps; macrosteps in the parlance of the previous chapters. The verification problem is defined in terms of properties which must hold across sequences of macrosteps. The languages designed for

formal verification must be related to a relevant mathematical structure such as the Kripke structure or one of the various ω -automata. The definition of this relationship is exactly the definition of the language's semantics. What is critical in a verification-oriented language is not any purported fit or flexibility relative to a simulation scheduling scheme or whether one can express degrees of freedom on resources or timing, rather it is that there is a clear definition of how programs in the language execute.

As with the simulation and synthesis cases, one can distinguish two levels of verification-oriented languages: a high-level and a lower-level. The low-level languages tend to have few features and there is little controversy about their suitability. Without much effort one can make a determination that the language either has features appropriate to the task at hand or it does not. The low-level languages tend to be fine-grained, based around a few simple primitives that can be connected together in large networks, possibly with some features for structural abstraction. Typically they are simple systems such as the network of multi-level logic and latches. Examples in this class of language include BEAVER [381], BLIF-MV [107] and even SMV [518] which all describe a form of abstract netlist.

This leads to a very simple two-phase scheme for verification from high-level languages. The first step is a synthesis procedure which takes the high-level language description as a source and produces an implementation in the low-level. Verification then occurs directly on the low-level description.

One must be careful at this stage to distinguish between cases where the high level language is *interpreted* in terms of some lower level *mathematical model* like a Petri net, a trace set or a network of communicating finite state machines and the cases where the high level language is *embedded* into the lower level language through an explicit synthesis step. The first situation is a case where the high level language's semantics is being

defined. It is a theoretical construction. In contrast, the second situation is one where the well-understood semantics of the low level language is exploited to give concreteness to the verification process. This synthetic embedding is a practical procedure with the result having a material form and being subject to measurable optimality criteria. The subtle distinction between the two cases is that in the first, different interpretations produce different semantic definitions while in the second different synthesized embeddings produce the same semantics but vary according to the optimality criteria.

One instance of this scheme is the HSIS [37] verification system which compiles [164] extended Verilog descriptions [49] into a network of logic and latches represented in BLIF-MV [107]. The verification is performed at the logic network level. There are numerous other examples in this vein that can be named. There is the CROCOS verifier [523] in which SDL [158] descriptions are compiled into the E-FCS mini-language which in turn has a semantics defined in terms of Dijkstra's *wp*-calculus [237]. Beer *et al.* [59] report a design flow under which descriptions in VHDL and a proprietary IBM HDL are synthesized to the gate level. The gate-level descriptions are written out in SMV [518] and the verification occurs at the SMV level. Courcoubetis *et al.* [214] posit¹ a design flow under which arbitrary VHDL descriptions including timing are transformed into Kursan's S/R [424] language which has been extended with certain real-time operators.

There are other more exotic examples of low-level verification-oriented languages with different emphases and levels of generality that can be named as well. Milne's CIRCAL [526] is a process algebra formulation of digital hardware behavior. It can be viewed as both a simple HDL and as a process algebra. Dill *et al.*'s Ever [239] [370] supports a non-deterministic interleaved style of semantics and is essentially a metalanguage type of notation for expressing verification problems.

1. In Courcoubetis *et al.* [214] the transformation procedure is defined. Also reported was the fact that, as of the time of publication, the procedure had yet to be implemented.

High level languages with formal semantic definitions can be distinguished as well. These have evolved in a number of ways over the years. On the one hand there were experiments with languages having exotic semantics, these varied widely as different approaches were tried over time. Examples in this realm include the imperative-style languages which have semantics defined by interval temporal logic such as Tempura [326] [546] [547] and Tokio [25] [440] [551]. Chandy and Misra's UNITY [159] and its derivative for hardware system description Murø [239] [370] used a nondeterministic interleaving semantics.

On the more traditional end of the spectrum were the proposals for languages which explored various synchronous alternatives. These languages are typically characterized by having finite state, static inter-process communication patterns and the macrostep cycle-level attributes necessary for placement within the framework of the RMC Barrier Theorem. This class includes Kurshan's S/R [424]¹ which was originally an $\bar{R}MC$ language but was later extended to be $RM\bar{C}$. Browne and Clarke's SML [122] [123] and CSML [193] [195] which is $\bar{R}MC$. Finally there is the family of Synchronous Languages, Esterel, Argos, Lustre and Signal which are surveyed in Halbwachs [320] and whose $RM\bar{C}$ semantics was presented in some detail in Section 5.7.

6.2 Hardware Description Language Standards

Despite the deep exploration of language possibilities shown in the preceding section, the standardized hardware description languages (HDLs), Verilog [570] and VHDL [384], have become accepted and even explicitly required in design flows with implementations becoming widely available in the early part of this decade. In the face of this, rather than proposing new languages or representations of behavior, more recent work has turned to finding ways to work *with* descriptions written in these languages, flaws and all. Unfortu-

1. An analysis of the S/R semantics was accomplished in Section 5.2.

nately, the discrete-event basis of these HDLs makes it very difficult to analyze the properties of arbitrary descriptions. On the one hand, this problem can be blamed on history as the standardization efforts came at a time when discrete event simulation of gate-level descriptions including timing and drive effects was essentially the only application domain. On the other hand, the problem has not been solved in recent years because there has been little understanding of what other sort of semantics to offer. As a partial solution to this, the more recent interest in cycle-level simulation, hardware synthesis, even software synthesis and the current interest in formal verification has led to proposals for HDL subsets and extensions that are more amenable to the techniques used in these domains. Before addressing the development of HDL subsets or extensions there must be some understanding of *why* these languages in their current state are problematic.

6.2.1 Discrete-Event Semantics

The fundamental problem with the standard HDLs is that they are based on the discrete-event paradigm. The intuitive perception is that their discrete event basis makes it very difficult to analyze a given system description and extract its cycle-to-cycle behavior. The difficulty stems from the complexity of an operational semantics defined by a simulator event loop. To date there have been few attempts at explaining this perception at a deeper theoretical level. The formal addenda [385] [386] and critiques have concentrated mainly on type inference issues and internal consistency. Having an explanation of why the discrete event model is problematic at the operational level would offer the possibility of proposing a different simulator event loop which did not suffer from the problems or of demolishing the idea of the event loop altogether.

Two attempts to analyze the operational bases of these languages can be distinguished. The first is the annotation language approach of the VHDL Annotation Language (VAL) [35]. In that work, the aspect of preemption was identified as the key flaw in the discrete

event semantics of VHDL. In response an anticipatory semantics which avoids preemption semantics was proposed. This approach is an instance of a more general scheme for identifying a non event-driven semantics to VHDL. A more in-depth presentation of VAL is deferred until Section 6.3.1. The second analysis of VHDL investigated its time model and simulation cycle through an information model formulation [304] [305]. Both these latter analyses however were oriented more at producing a precise specification of the standard; the critique they offered was at best a derived notion. Here, with the benefit of the RMC Barrier Theorem the theoretical framework can finally be given. Before that presentation however, the level of commonality between the semantics of Verilog and VHDL must be established.

6.2.2 VHDL and Verilog

The superficial differences between the two HDL standards has colored much of the debate on their utility and suitability. The distinctions between HDLs have come to be seen as being based on a vague notion of intuitive fit with a prototypical designer's needs. Arguments have been put forth that the C-style syntax of Verilog is more suitable for an HDL than the a Ada-style syntax of Verilog. The substantive differences between the two languages stem more from industry investment issues such as the availability and quality of simulator implementations, third-party model libraries and a user base familiar with the language. In short, these arguments tend to have primarily a cultural and business thrust being based more on feelings of personal preference with the latent justification being the sunk cost in the existing infrastructure. As well there are engineering-level differences which are surfaced at the user level. These included the fact that VHDL is has a very rich but strong type system whereas Verilog has a very weak but weak type system. This allows very detailed, intricate and incompatible logic modeling packages to be developed in VHDL whereas the four-valued strength model of **0**, **1**, **X** and **Z** is essentially fixed in Verilog. Such incompatibility caused by excess flexibility must be solved through *post*

hoc standardization efforts such as the IEEE's **Logic_System** standard for representing multi-valued logic in VHDL [89].

6.2.2.1 The Event Queue Paradigm

Fortunately it is fairly easy to see, within fairly broad limits, that at the semantic level the discrete event model used in Verilog and VHDL are essentially the same. Both simulator event queues are predicated upon the execution of processes which in turn cause events to be scheduled for the current or a future time point. In such a scheme there are explicitly two sorts of time. There is a macro level which is measured in terms of discrete units such as seconds, or more commonly nanoseconds. There is also a finer level, commonly referred to as δ -time, which expresses the ordering relationship of computations and events within a single time point.

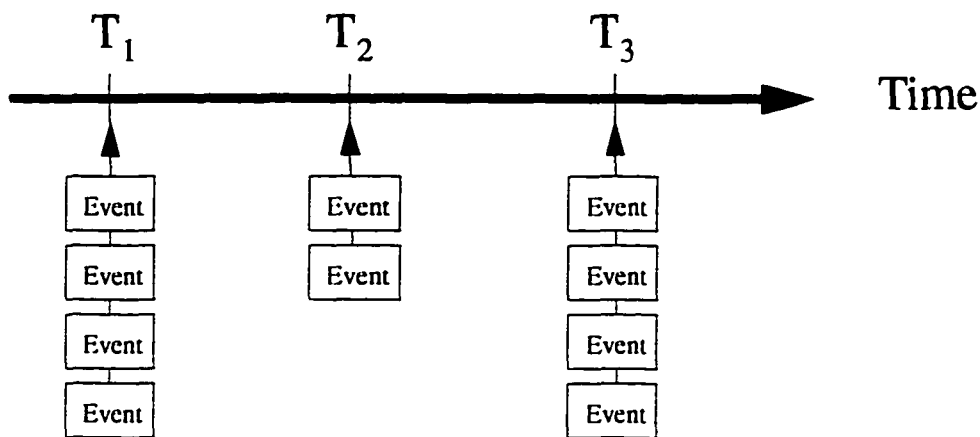


Figure 6-1. The Simulator Event Queues of the Discrete Event

Operationally, the simulator event loop for both VHDL and Verilog are defined in terms of event queues which organize the events scheduled for the time points. The event loop processes events in the event queue for the earliest time point. The events trigger the invocation of processes in the model which are sensitive to those events. The execution of the triggered processes in turn causes more events to be enqueued either for the same time

point or for a future time point. Because the invocation of a process may schedule events for the current time point as well as future time points, there is an iterative nature to the processing of a time point through the event loop: events are processed until there are no more events queued at the current time step. The structure to time and its associated time queues as illustrated in Figure 6-1.

- A simulation cycle consists of the following steps:
1. If no driver is active, then simulation time advances to the next time at which a driver becomes active or a process resumes. Simulation is complete when time advances to *TIME'High*.
 2. Each active explicit signal in the model is updated. (Events may occur on signals as a result).
 3. Each implicit signal in the model is updated. (Events may occur on signals as a result).
 4. For each process *P*, if *P* is currently sensitive to a signal *S*, and an event has occurred on *S* in this simulation cycle, then *P* resumes.
 5. Each process that has just resumed is executed until it suspends.

Figure 6-2. The VHDL Simulation Cycle

In the case of VHDL, the IEEE standard formally defines the fine structure within a single time step by specifying the simulation cycle event loop explicitly. The VHDL simulation cycle is shown in Figure 6-2.¹ There the fine structure is the δ -step which consists the execution of the *set* of runnable processes that are resumed in step 4. The salient point about the δ -step formulation is that all the processes at a particular δ -step have the same view of the simulator state. Signal assignments executed in the current δ -step only become effective at the start of the next δ -step, in steps 2 and 3 of the simulation cycle. From the viewpoint of the simulation cycle therefore the macrostep consists of an arbitrary number of δ -steps. The number is arbitrary because the current macrotime point is

1. From IEEE Std 1076-1987 [384], Section 12.6.3, page 12-14.

only updated when there can be no more δ -steps in the current macrotime point. The traditional depiction of VHDL's time model is illustrated in Figure 6-3 with the implicit understanding that at each δ_i consists of the invocation of a *set* of processes that were resumed for that δ_i . Specifically, each process which is resumed in the δ_i sees the same simulator state. This view is made precise in Section 6.2.3.1 and Figure 6-3.

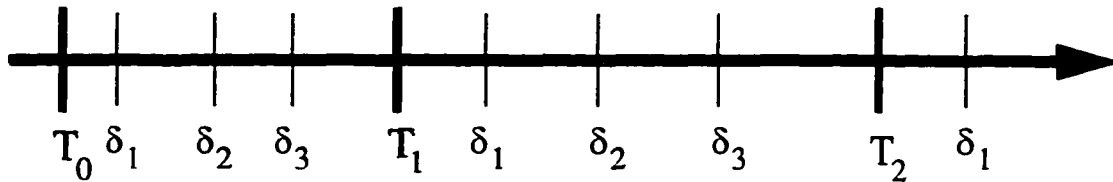


Figure 6-3. The Traditional View of Two-Level Time in VHDL

6.2.2.2 Differences in Event Queue Management

Between the two languages there are some minor differences in the conceptual location of the event queue. In particular, in VHDL the event queue is conceptually associated with the *driver* of the signal and a signal driver is typically associated with a particular process. In the case where a signal is driven by multiple processes as in the case of a bus or register signal, a distinction is made between the *driving value* of the signal and the *effective value* of the signal. The driving value is the value which is assigned to the signal in a step whereas the effective value is the value which actually appears on the signal as a result of the assignment. The driving value and the effective value may be different in the case of bus or register signal kinds or when there are implicit type conversions defined for the signal.

In Verilog on the other hand, the event queue is conceptually associated with the current lexically enclosing scope. Normally the two schemes produce the same results, but there is a subtle interaction between the Verilog **task** construct and the non-blocking assignment.¹ A task is similar to a procedure in VHDL. The non-blocking assignment of Verilog

and the signal assignment of VHDL are substantially the same. Both affect the value of the signal after the current invocation of the process is finished. In Verilog the nonblocking assignments take place when the enclosing process suspends at a delay statement such as an `@(posedge . . .)`. The interaction is illustrated in Figure 6-4 where a nonblocking assignment occurs within a task but the task exits before the enclosing process, the `initial` block, suspends. The new value of 1 to be assigned to `VALUE` at the end of the invocation is associated with the lexically enclosing block in which the assignment occurs. Yet when the task exits that lexical scope is exited. The signal `VALUE` remains at 0 instead of taking on the new value of 1.

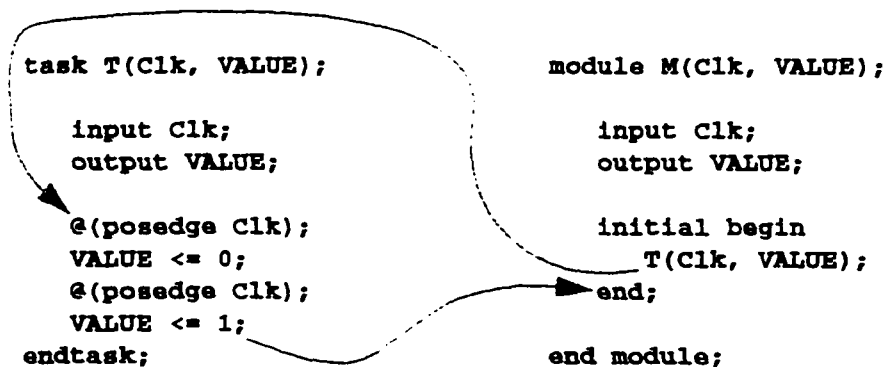


Figure 6-4. Interaction of a Verilog Task and Non-Blocking Assignment

6.2.2.3 Differences in Signal Assignment

On the same point, the non-blocking assignment of Verilog is subtly different than the VHDL signal assignment in the case where multiple non-blocking assignments occur in a time step. The semantics of the non-blocking assignment is defined as follows: the right-hand side of the assignment statement is evaluated and the value is scheduled for later update. However, if there is more than one non-blocking assignment to the signal in the time step then the value actually selected is nondeterministic. In contrast, such a case in

1. This example was pointed out to me by Gary York of Cadence Design Systems Inc.

VHDL would always result in the last-assigned value being the value of the signal. For example, in Figure 6-5 shows two processes, one in Verilog and one in VHDL. The Verilog example may result in either a **0** or a **1** on the signal **OUT**. In VHDL, the result is always **1**. This underspecification was originally installed in the language definition to allow implementation freedom in simulator implementations. Recently though the peculiarity has been exploited in a proposal for a policy-of-use for describing nondeterministic computations in Verilog [49]. Declared nondeterminism is a powerful abstraction mechanism in verification.

Verilog	VHDL
<pre>output OUT;</pre>	<pre>signal OUT: bit;</pre>
<pre>initial begin</pre>	<pre>process</pre>
<pre> OUT <= 0;</pre>	<pre>begin</pre>
<pre> OUT <= 1;</pre>	<pre> OUT <= 0;</pre>
<pre>end;</pre>	<pre> OUT <= 1;</pre>
	<pre> wait;</pre>
	<pre>end process;</pre>

Figure 6-5. Verilog and VHDL Signal Assignment

6.2.2.4 Overriding Similarities

There are a number of differences between the two standardized HDLs over and above the superficial contrast between their respective syntactic styles. At the deepest semantic level however, Verilog and VHDL are fundamentally the same. They both are defined in terms of a core simulation cycle which executes processes in the event driven fashion. In fact it can be argued that the simulation cycle for both languages are the same with the admission that the VHDL cycle is possibly more concretely detailed at the language level. Indeed, the current industry trend towards mixed Verilog/VHDL simulators is another form of evidence that the two languages have the same fundamental semantics. As such it is sufficient, for the purposes of the semantic analysis attempted in this work, to speak of

either Verilog or VHDL. For convenience, the remainder of this chapter describes only VHDL, its subsets and proposed extensions. The obvious analog in the case of Verilog can be inferred directly. It remains to establish where the standard HDLs fit in the semantic framework of the RMC Barrier Theorem and microsemantics.

6.2.3 Event-Driven Semantics

As stated, the discrete event model defines semantics through the manipulations performed on the simulator event queues. For the purposes here, the VHDL simulation cycle shown in Figure 6-2 can be used as a prototypical discrete event execution loop. In order to fit its semantics within the framework of the RMC Barrier Theorem and microsemantics some analysis is necessary. In particular that simulation cycle, as stated, is purely operational being defined by (very) high-level pseudocode acting in an *individual* simulator state vector. What is necessary is a formulation of that simulation cycle in as an image semantics where *sets* of simulator state vectors are $Q(c)$ transformed by a macrostep transition relation $T(c, i, o, n)$. That being done the analysis of the discrete event model in terms of the RMC Barrier Theorem and microsemantics will be fairly straightforward.

Fortunately almost all of the necessary analysis has already been performed in the example microsemantics of Example 4.3.4.6. That example, which actually is the original event-driven semantics of StateCharts [330], fits the situation at hand almost exactly. To use that result directly, it suffices to show that the simulation cycle of VHDL has the same attributes as Example 4.3.4.6. The results will then follow through directly. There are three salient attributes to that example. The first and most obvious is that the example has a three-level temporal structure with a macrotime, a δ -time and a η -time. In contrast, the traditional view is that VHDL has a two-level time with a macrotime and a δ -time. That VHDL has a three-level time in its relational μ -calculus formulation must be shown.

Secondly \bar{R}_δ holds in the example semantics. This means events emitted during the η -steps of δ_i do not become visible until δ_{i+1} . Though it is intuitively clear that this is the case in the VHDL simulation cycle, \bar{R}_δ must be shown in the context of the three-levels of time established previously.

Finally, the semantics of Example 4.3.4.6 is reactive. This means that signal emissions relate to the current macrostep instant not to future instants. In the example, this aspect is governed by Ex 4.3.4.6-1. The unrestricted semantics of VHDL is not reactive. This means that signal emissions created one macro instant can appear and affect future macro instants. To be reactive, the admissible programs must have the property that events consumed in a macro instant must have been produced in that macro instant, produced either by the system as a reaction in some δ -step or spontaneously by the environment at the start of the macro instant. This condition can be violated when either inertial or transport delay is used. Therefore some limits must be established on the use the nontrivial delay operators in order to place the VHDL simulation cycle within the framework of the RMC Barrier Theorem.

After establishing these similarities and restrictions it is straightforward to show that the semantics defined by VHDL's discrete event simulation cycle is $\bar{R}\bar{M}\bar{C}$. By analogy the same result must hold for the simulation cycle of Verilog.

6.2.3.1 The Discrete-Event Model has a Three-Level Time

An examination of the VHDL simulation cycle shown in Figure 6-2 revealed the traditional two-level time shown in Figure 6-3. That two-level view of VHDL semantics however elided an important facet of the semantics: that within a δ -step a *set* of processes were run in some order. With the understanding that every runnable process runs for only a finite number of steps, the computation produced by the resumption of a process can be exactly characterized via a transition relation. Let this transition relation be named T_η .

Elements of T_η relate the program state before the resumption of the process to the state after the invocation. In this light, the picture of the simulation cycle shown in Figure 6-3 can then be rewritten to illustrate the resumptions of individual processes. There, the set of processes resumed in a δ -step appear as η -steps between the δ -steps.

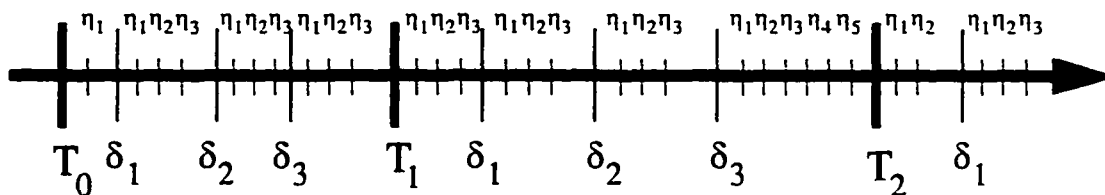


Figure 6-6. The Three-Levels of Time in VHDL Discrete Event Semantics

From this analysis and the depiction in Figure 6-3 it is clear that VHDL has a three-level structure of time with exactly the same structure as Example 4.3.4.6.

6.2.3.2 The Discrete Event Model is \bar{R}_δ

The view of VHDL simulation shown in Figure 6-3 can be related to the five steps of the VHDL simulation cycle from Figure 6-2. The procedure described in step 1 governs the number of δ -steps in the macrostep. In the middle are steps 2 and 3 which describe how the values on signals are updated for the next δ -step. The resumption of processes and their execution described in steps 4 and 5 clearly corresponds to the η -steps.

By analogy with Ex 4.3.4.6-1, steps 2 and 3 that the outputs O_i become defined after being emitted in the η -steps of the previous δ -step. These two steps update the simulator state so that all the process resumption η -steps compute against the same set of outputs. Thus \bar{R} holds over δ -steps. This condition is exactly the definition of the \bar{R}_δ from Example 4.3.4.6 therefore the semantics of the discrete event model is \bar{R}_δ .

6.2.3.3 The Synchronous Subset of the Discrete-Event Semantics

The major difference between the semantics of Example 4.3.4.6 and that supported by VHDL's simulation cycle is the property of reactivity [498] wherein, at the macrostep level, outputs are produced only in response to inputs. Outputs do not occur spontaneously with respect to inputs. The example semantics is reactive whereas full VHDL allows for non-reactive programs. This is not particularly due to the simulation cycle *per se*, but rather to the ability of a running process to emit signals with nontrivial delay through the **after** delay modifier. The **after** clause in a signal assignment allows a process to emit events which will occur in future macro instants.

A trivial example showing how nontrivial delay in signal emission produces non-reactive semantics is shown in Figure 6-7. In that case, the output **O** follows the input **I** when **STOP** is **TRUE** but freely oscillates when **STOP** is **FALSE**. The free oscillation in this second mode is non-reactive. This distinction between reactivity and non-reactivity was formalized in Section 5.7.2 in the distinction between SL_3 and SL_4 .

```
entity INV_LOOP is
  port(I: in bit; STOP: in bit;
        O: out bit);
end INV_LOOP;

architecture OOPS of INV_LOOP is
  signal INTERNAL: bit := '0';
begin
  with STOP select
    INTERNAL <= not INTERNAL after 10 ns  when FALSE,
              I after 10 ns              when TRUE;
  O <= INTERNAL;
end OOPS;
```

Figure 6-7. An Example of a Non-Reactive VHDL Program

The principle of reactivity requires that events produced in a macro instant must be causally traceable to an input event in that same instant. By extension, events consumed in the current instant must be produced in the current instant. They may have been produced either by the environment or internally by another process. In VHDL this principle plays out in the context of the δ -steps: the principle of reactivity requires that events consumed within a macrostep must be generated either on an input signal or within the macrostep but in a previous δ -step. Specifically, this means that the use of inertial and transport delay, the two forms of nontrivial delay in VHDL, must be severely limited.

In any VHDL model there are three classes of signals. There are *input signals* which communicate events and values from the environment into the model, there are *internal signals* which transfer events and values among subparts of the design and there are *output signals* which communicate from inside the model to the environment. This classification of signals is illustrated in Figure 6-8. In general, the interconnection of the processes by internal signals can be arbitrary and in particular there may be feedback loops among the processes. The resolved signal kinds **bus** and **register** are supported in the manner prescribed by the standard through what amounts to a distributed multiplexor arrangement. They naturally fall into this classification scheme. Too, without loss of generality one can assume that signals are either input or output but not both and that input signals are read-only while output signals are write-only.

These three classes of signals directly relate to the issue of reactivity and the minimal limits on the uses of nontrivial delay in a reactive VHDL model. The input signals are by definition read-only so signal assignment cannot occur on them. For this class of signal, nontrivial delay semantics is a non-issue. This leaves the use of internal signals and the output signals as the sites where reactivity must be established. In the case of internal signals, reactivity requires that events consumed in the macrostep be produced either on an input signal or on an internal signal in a previous δ -step of that same macrostep. This

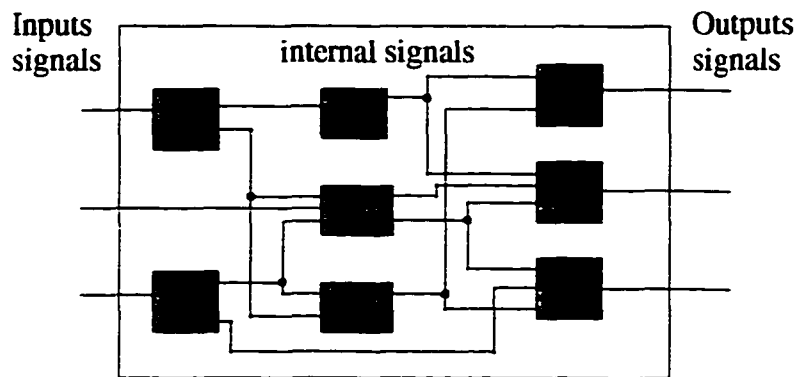


Figure 6-8. The Classes of Signals in a VHDL Program

implies that reactivity is preserved only in case unit delay is used on internal signals. The use of either inertial delay or transport delay on these internal signals would essentially allow for spontaneously produced responses, thereby violating the reactivity property.

The case of output signals is a bit more subtle. Output signals cannot be read so events occurring on them are guaranteed never to cause any further δ -steps within the instant. Thus from a semantic viewpoint it does not matter whether their emission is delayed in macrotime or it occurs in the same instant. From an external perspective therefore there is a subtle distinction between when an output signal is *emitted* and when it actually *occurs*. Signal emission is a notion that is relative to an internal view of the system whereas signal occurrence is one which is relative to an external or global view of the system. Signal emission is associated directly with the execution of the signal assignment statement. In contrast, signal occurrence is the macrostep point at which the value change registered by the signal assignment actually is observed externally.

The key insight for output signals in a reactive context is that from a semantic point of view nontrivial delay between their emission and occurrence is irrelevant. In a practical implementation, the environment might care about this delay, however since an output signal cannot trigger further δ -steps within the model, any nontrivial delay is transparent to

cause-effect analysis. What is not transparent are two potential side features of delayed occurrence which must be constrained in order to preserve reactivity. The first is a potential reordering of delayed reactions due to varying delays between reactions. The constraints on delays in this regard are the same constraints established for the “Codesign” Finite State Machine (CFSM), presented in Example 4.3.4.5. In particular the constraints on reaction order shown in Figure 4-9 must be observed. The second constraint is the involability of the connection between a signal emission and its occurrence. This is the issue of preemption, namely that a signal emission once executed entails the occurrence of the event; it cannot be preempted by subsequent signal emissions. Constraints in this regard were established for the CFSM as well. Preemption has also been investigated in the context of VHDL with the VHDL Annotation Language which is presented Section 6.3.1.

In the general case with arbitrary nontrivial delays, establishing the non-reordering or non-preemption properties can be quite difficult. It is definitely beyond the scope of the work here. A sufficient condition for reactivity is that internal and output signal assignments use only unit delay.

6.2.3.4 The Discrete-Event Semantics is $\overline{RM}\overline{C}$

With this background it is now possible to show that the semantics of VHDL is $\overline{RM}\overline{C}$. As with the analysis used in the examples of Section 4.3, the focus here is on what the discrete event semantics of VHDL *can* allow rather than what is a reasonable and proper use of that semantics.

Consider the properties of the example VHDL program shown in Figure 6-7. It has the interesting property that when an odd number is given at the input \mathbf{I} then the output \mathbf{O} becomes $\mathbf{I}+6$ at the end of the macrostep. When an even number is given at \mathbf{I} then the program never finishes the end of the macrostep; it oscillates forever in δ -time. These two scenarios are illustrated in Figure 6-10 and Figure 6-11 respectively. In those figures

```

entity Weird is
  port(I: in integer; O: out integer);
end Weird;

architecture Ripple of Weird is
  function Resolve(sources: in integer_vector)
    return integer;
  subtype resolved_integer is Resolve integer;
  signal i0: resolved_integer bus := -1;
  signal i1, i2, i3: integer := -1;
begin

  P0: i0 <= I;

  P1: process
  begin
    wait on i0;
    i1 <= i0 + 2;
  end process;

  P2: process
  begin
    wait on i1;
    i2 <= i1 + 2;
  end process;

  P3: process
    variable var: integer := 0;
  begin
    wait on i2;
    var := i2 + 2;
    if 0 = var mod 2 then
      i0 <= var mod 8;
    else
      i3 <= var;
    end if;
  end process;

  P4: O <= i3;

end Ripple;

```

Figure 6-9. An VHDL Program Illustrating \overline{RMC} Behavior
the highlighted values indicate that the corresponding driving process was executed in the δ -step and that the driven signal changed value. This example is interesting because it is not an entirely unreasonable one. It also illustrates all three properties.

Signal	Driver	T_0	δ_1	δ_2	δ_3	δ_4	δ_5	δ_6
I	env	1	1	1	1	1	1	1
i0	P0	-1	1	1	1	1	1	1
i1	P1	-1	-1	3	3	3	3	3
i2	P2	-1	-1	-1	5	5	5	5
i3	P3	-1	-1	-1	-1	7	7	7
O	P4	-1	-1	-1	-1	-1	7	7

Figure 6-10. The Execution of Figure 6-7 on an Odd-Valued Input

Signal	Driver	T_0	δ_1	δ_2	δ_3	δ_4	δ_6	δ_7	δ_8	δ_9	δ_{10}	δ_{11}	δ_{12}	δ_{13}	etc
I	env	0	0	0	0	0	0	0	0	0	0	0	0	0	...
i0	P0	-1	0	0	0	4	4	4	0	0	0	4	4	4	...
i1	P1	-1	-1	2	2	2	6	6	6	2	2	2	6	6	...
i2	P2	-1	-1	-1	4	4	4	8	8	8	4	4	4	8	...
i3	P3	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	...
o	P4	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	...

Figure 6-11. The Execution of Figure 6-7 on an Even-Valued Input

Responsiveness

Responsiveness requires that there exist at least one output which is dependent upon a particular input value. The example of Figure 6-10 shows one such example: **o** is **7** just when **i** is given as **1** at the start of the macrostep. The semantics of VHDL is R .

Modularity

Modularity requires that the execution of any one subpart of the design, take process **P1** for example, can be determined solely from the inputs and the outputs produced by all the other components. These would be processes **P0**, **P2**, **P3** and **P4**. An consequence of this principle is that there can be but a single value for any output in a macrostep. The scenario of Figure 6-11 shows that the driven outputs of processes **P0**, **P1** and **P2** are multi-valued and δ -step order dependent. The semantics of VHDL is therefore \bar{M} .

Causality

Causality requires that there be a partial order relating inputs to outputs which exists at the component level and is also preserved at the aggregate level. In the example scenario of Figure 6-11 it can be observed by tracing across the δ -steps that there is no partial order among the signals. For example, **i0** both causes **i1** and is caused by **i1**. In the forward direction, **i0** causes **i1** directly in **P0**. In the reverse direction, **i1** causes **i2** in **P1**, **i2** then causes **i0** in **P2**. Any system-level partial order would have both $i0 \leq i1$ and

$i1 \leq i0$. Yet because $i1$ and $i0$ are distinct signals it is the case that $i0 \neq i1$. No causal partial order exists. The semantics of VHDL is \bar{C} .

6.3 Beyond Discrete-Event Semantics

There are been several proposals to move beyond the discrete event basis of hardware description languages such as VHDL or Verilog. These efforts are predicated on the understanding that, for practical purposes, the discrete event semantics of these languages cannot be changed. The investment in them is simply too great to presume that a newly proposed language would have any hope of competing in the marketplace of ideas, let alone be commercially successful. So while language subsets may be admitted in this view, there is no place for a reinterpretation or redefinition of the standard. As an example, in the case of VHDL, the simulation cycle which must be used is defined by the standard and to change that definition even in minor ways changes invites the charge that the new operational definition constitutes a new language. To claim to support VHDL, the language must be treated as-is with the analysis scheme adapting to the task not vice versa. Two contrasting approaches to using standard VHDL in this vein are the annotation language approach of the VHDL Annotation Language and the Synchronous VHDL subset.

6.3.1 The Annotation Language Approach

The fundamental insight in the annotation approach is that a different denotation with desirable analytic properties can be *associated* with a discrete event model through the use of a carefully constructed annotation language. The trade-offs in such a scheme are the properties of the analytic domain itself and how the analytic domain is mapped down to the discrete event domain. In the case of the VHDL Annotation Language (VAL) [33] [34] [35], the analytical domain is the Waveform Algebra [31] where the denotations of VAL statements are interpreted as infinite waveforms on a discrete time line and the association to the $\bar{R}\bar{M}\bar{C}$ semantics of the VHDL model is declared by assertions and validated opera-

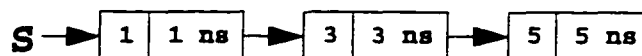
tionally through simulation.

6.3.1.1 The VAL Semantics of Time

The VHDL language supports two sorts of macrotime delay models: inertial delay and transport delay. Both of these delay models have the property that they are preemptive. This refers to the effect of a signal assignment which destroys the previously-projected values planned for the future value of the signal. The preemptive effect of inertial and transport delay signal assignments are illustrated in Figure 6-12 and Figure 6-14 respectively.¹ In the preemptive semantics, events projected by one signal assignment operation are later canceled when more information is available. In contrast to the preemptive semantics of VHDL, the VAL timing model is called *anticipatory* because any event, once projected, is guaranteed to take place. The declaration of the event's future existence anticipates its actual occurrence.

```
signal S: integer := 0;  
  
P1:  
process  
begin  
  S <= 1 after 1 ns, 3 after 3 ns, 5 after 5 ns;  
  S <= 4 after 4 ns;  
end process;
```

Driver of S after statement #1



Driver of S after statement #2

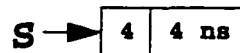


Figure 6-12. The Preemptive Aspect of VHDL Inertial Delay

1. The figures are adapted from Lipsett et al. [478]. More detailed examples of preemption are given in Augustin et al. [35].

To facilitate the declaration of event occurrences a new expression construct, the timing operator is introduced in VAL. It takes the form of an array-like temporal offset index on a value carrier such as a signal or state name: **NAME[offset]**. Unlike VHDL where all references to time must be positive, thereby referencing points strictly in the future, the VAL timing operator can reference points either in the future or the past. An example declaring that **STATE** adopts the value of **X** in three nanoseconds under the condition that **Y** was true two nanoseconds ago is shown in Figure 6-13.¹

```
when Y[-2 ns] then  
    STATE <- X[3 ns];  
end when;
```

Figure 6-13. A VAL Annotation Specifying a Buffer with Delay

The VAL language provides various statements for declaring constraints on behavior in this anticipatory fashion. There are statements such as **when**, illustrated above, and **select** which express the instantiation of conditional guards. The guards can be nested arbitrarily deeply with nesting interpreted in the obvious manner as the conjunction of the enclosing guard conditions. The **drive** statement, also illustrated above, declares the anticipatory state transition subject to the enclosing guards.

The effect of the VAL statements is the instantiation of guard conditions around drives statements. This describes state transitions predicated on the conditions of the guards and defines an abstraction of the VHDL model which has anticipatory semantics and is thus formally analyzable in terms of expressions in the Waveform Algebra. This analysis is accomplished outside of the scope of the annotation language.

1. Adapted from Augustin *et al.* [35], page 193.

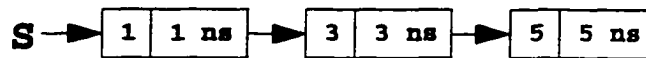
```

signal S: integer := 0;

P1:
process
begin
    S <= transport 1 after 1 ns,
        3 after 3 ns, 5 after 5 ns;
    S <= transport 4 after 4 ns;
end process;

```

Driver of S after statement #1



Driver of S after statement #2

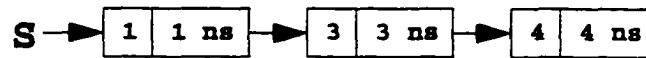


Figure 6-14. The Preemptive Aspect of VHDL Transport Delay

6.3.1.2 Mapping to the Discrete-Event Semantics

The VAL abstraction is associated with the underlying discrete event VHDL model by means of assertions which are said to map state transitions in the analysis domain to events occurring at the discrete event level. The validity of these assertions is established operationally by simulation runs. The claim is that if no assertion fails then the preemptive semantics of the discrete event execution is said to implement the anticipatory semantics of the VAL abstraction. The important aspect is that the mapping between the VAL abstraction and the VHDL model is established empirically through execution rather than formally through an analysis of the discrete event semantics.

Of interest here are the range of assertion types supported in the mapping part of the VAL language. The language designers chose to expose the δ -time level of VHDL in assertion expressions in order to give finer control over the specified behavior. The four flavors of assertions and their effect are shown in Figure 6-15:¹

- the **assert** flavor is the same as the **assert** statement in VHDL.
- **eventually** enforces a monotonicity property within a macrostep that once the expression becomes true it remains so,
- **finally** enforces that the stated condition is true at the very last δ -step,
- **sometimes** enforces an existential property that the stated condition is true for at least one δ -step.

Macrotime	T_{i-1}	T							T_{i+1}	T_{i+2}						
δ -time	1	1	2	3	4	5	6	7	1	2	3	4				
Guard	t	t	f	f	t	t	f	t	t	f	f	t	t	f	f	t
Assertion	t	f	f	f	t	f	f	t	t	f	f	f	t	t	f	
assert	-	v	-	-	-	v	-	-	-	-	-	v	v	-	-	v
eventually	-	-	-	-	-	v	-	-	-	-	-	v	-	-	-	v
finally	-	-	-	-	-	-	-	-	-	-	-	v	-	-	-	v
sometime	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	v

v - indicates violation of the assertion

Figure 6-15. Violation Conditions for the VAL Assertion Flavors

The assertions and model declaration statements may be arbitrarily intermixed. An example of an annotated model is shown in Figure 6-16.¹

6.3.2 The Synchronous VHDL Subset²

A second approach to establishing an analyzable subset of a discrete event based language starts from its \overline{RMC} semantics and identifies a different semantics within the fully general one. As with many of the previously studied semantics, a per-system restriction is used to disallow descriptions that do not have the target property. This subset of the semantics can then be said to define a subset of the language. The language subset is

1. Adapted from Augustin *et al.* [35], Figure 3.5, page 56.

1. Adapted from Augustin *et al.* [35], page 92.

2. This section revisits the original presentation of Synchronous VHDL [47] and places that proposal in the framework of the RMC Barrier Theorem and microsemantics developed here.

```

entity DFF is
  generic (SETUP, HOLD, DELAY: TIME);
  port (Clk: in BIT;      -- Clock input
        D: in BIT;       -- Data input
        Q: out BIT;      -- Output
        Qbar: out BIT);  -- and its complement
  --| assume DELAY >= HOLD
  --|   report "Error in generic constant";
  --| state model is BIT; -- A single bit of memory
begin
  --| when Clk'Changed('0') then
  --|   when D'Stable during [-SETUP, HOLD] then
  --|     D -> State[DELAY];
  --|   else
  --|     assert false report "SETUP-HOLD violation";
  --|   end when;
  --| end when;
  --| finally (State = Q) and ((not State) = Qbar)
  --|   report "State does not agree with output";
end DFF;

```

Figure 6-16. An Entity with VAL Annotations defined by the target subsemantics rather than vice versa. For the $RM\bar{C}$ semantics of discrete events, an obvious choice is a subsemantics that is M resulting in an $RM\bar{C}$ semantics as per the Synchronous Language semantics. Sublanguage identification in this manner is a fundamentally different approach than the analogical one used for the hardware semantics of Section 6.1.2.1.

The Synchronous VHDL [47] subset is defined in exactly this manner, arriving at an $RM\bar{C}$ semantics that fits within the $RM\bar{C}$ semantics of the full language. The key aspect is that the restriction on the semantics drives the restrictions on the allowable language constructs. The restricted semantics therefore defines a subset of the standard language which is synchronous thereby reestablishing the semantic foundation of the language away from its original discrete event basis but in a way which is guaranteed to be consistent with it. Such a proposal however does little to address conceptual omissions in the original language such as the potential confusion between state variables and output variables or the lack of a hierarchical behavioral constructors analogous to the existing hierar-

chical structural constructors of the *instance* and the *guarded block*. A proposal which specifically addresses this last deficiency is presented in Section 6.4.2.

The original development of Synchronous VHDL was undertaken as a proof of concept to determine if such a subset existed and whether such a semantics-defined language subset would be practicable, convenient and expressive enough for design description. These last judgements are ultimately as subjective as are all claims about language suitability.¹ With some lack of objectivity what can be described is the expressiveness and intuitive identification of legal programs. Synchronous VHDL is a highly restricted subset of a much richer computational model. Of interest to any user therefore is the ease with which admissible and inadmissible programs can be identified. Unfortunately, in practice, the subset is difficult to program to for reasons which are outlined in the ensuing sections. As the ensuing sections show, there is more than a little ambiguity surrounding how one might algorithmically check whether or not a given program instance is admissible in the synchronous subset. So while an $RM\bar{C}$ subset clearly exists, it is often difficult to determine *a priori* whether a given program instance has this property. Section 6.3.2.5 summarizes these reasons in the form of lessons learned in semantics and language design.

6.3.2.1 The Modularity Conditions

The definition of Synchronous VHDL starts from the semantic level and moves forward to the syntactic level. At the semantic level, unrestricted VHDL was shown to be explicitly non-modular, as $RM\bar{C}$, in Section 6.2.3. From this situation some per-system restrictions must be used to establish a modular subsemantics. This subsemantics would of course be $RM\bar{C}$ and once modularity is established then the theory of the Synchronous Languages can be applied. In particular the notions of causality and surpassing the RMC Barrier

1. Stoustrup's [672] introductory comments about language adoption being essentially a "life-style issue" are directly relevant in this regard.

through per-system causality checks, whether they be static or dynamic checks, are directly applicable.

The primary focus in Synchronous VHDL therefore is on the set of conditions which establish M . The analysis in Section 6.2 showed that the reason VHDL is \bar{M} is that, the three-level model of time in conjunction with \bar{R}_δ , allows signals to have multiple values in a macrostep. The M -establishing conditions must ensure that a single value is seen for all signals across all microsteps. The necessary conditions are:¹

1. single assignment across all δ -steps in a macrostep,
2. no write after read across all δ -steps in a macrostep.

The first condition ensures that a signal does not take on more than one value in a δ -series by any explicit act. That is, by multiple assignment operations. The second criterion ensures that the rest of the system does not make a decision based on any but the single emitted value of a signal in a δ -step series. These conditions are necessary; they must always hold for any system to have the M property. A simpler but not fully general set of sufficient conditions is given in Section 6.3.2.4.

The restriction to system descriptions which has M is one of the two per-system restrictions imposed on arbitrary VHDL descriptions to define the Synchronous VHDL subset. The check that M is obeyed is called *modularity checking* and is similar in spirit to the causality checking done in the Synchronous Languages. In that case, the $RM\bar{C}$ semantics does not guarantee causality, the semantics is \bar{C} , so the compiler must check that the system instance has C . Here the $R\bar{M}\bar{C}$ semantics does not guarantee modularity or causality so the compiler must check that the program instance has M as well as C .

1. These necessary conditions, also called *must conditions*, and other more practical sufficient conditions, also called *may conditions*, were originally proposed by Gonthier [295] for the denotational semantics of Esterel.

6.3.2.2 Differing Models of Time

The foregoing semantic constraints are relatively straightforward but they do not provide an effective means of checking the admissibility of an arbitrary VHDL description in the synchronous subset. Any practical modularity checking problem is complicated by the different models of time used in the two regimes. What must occur in a modularity checking procedure is the definition of a correspondence between the \overline{RMC} interpretation of discrete events and the \overline{RMC} interpretation of the synchronous semantics. This kind of correspondence is illustrated in Figure 6-17.

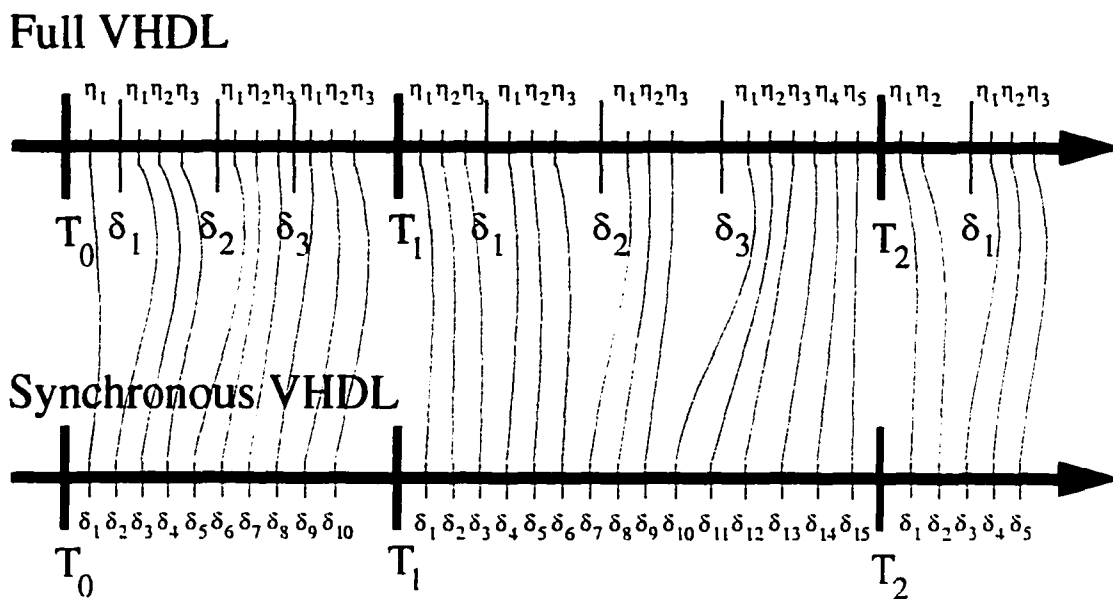


Figure 6-17. Extracting Synchronous Time from Discrete Event Time

There are three salient points in VHDL semantics which are relevant here. The first is that the three-level time of full VHDL is \overline{R}_δ . Secondly it can be noted that the forward and backward image computations $F_\eta\{Q\}$ and $B_\eta\{Q\}$ at the η -time level are (trivially) monotonic in the information ordering. Finally, at the δ -level the computations $F_\delta\{Q\}$ and $B_\delta\{Q\}$ are not monotonic in this information ordering. In contrast the \overline{RMC} synchronous semantics only has two levels and the forward and backward image computations

$F_{\delta}\{Q\}$ and $B_{\delta}\{Q\}$ are monotonic in the information ordering.

The major effect which distinguishes the discrete event semantics from the synchronous semantics is the \bar{R}_{δ} property of the discrete event semantics. In the discrete event world, the values seen on signals are the values assigned there from some previous δ -step. The \bar{R}_{δ} property implies that any new value assigned in step δ_i does not take effect until step δ_{i+1} . This means that even under a restriction to single assignment in a macrostep (condition 1 in Section 6.3.2.1), that it is possible for a signal to be used with two different values if it is read before it is written. This is the case when a signal is read in step δ_i but is assigned in δ_j and $j \geq i$. This case is the reason for condition 2.

Operationally the effect of \bar{R}_{δ} on modularity is subtle. This is because the granularity of process invocation interacts with the prohibition against read-before-write across the η -steps. Examining the VHDL simulation cycle shown in Figure 6-2, it can be observed that when a process becomes runnable, the process runs until it explicitly suspends because it executes a *wait* statement.¹ This suspension at a *wait* statement may or may not be permanent for the macrostep depending on whether the sensitivity condition of the *wait* is triggered again in a subsequent δ -step of the macrostep. In contrast, in the synchronous semantics there is no *explicit* notion of intra-macrostep suspension. Instead, Gonthier's principle of causal execution provides an *implicit* form of synchronization through the requirement that a signal can only be read after it is written in the macrostep.²

1. Without loss of generality, this presentation as the earlier presentation of Synchronous VHDL [47] assumes that the VHDL description has been reduced down to the core language. The presumed level of description is a control-dominated coordination scheme of communicating processes. For any semantic analysis, it is sufficient to examine the properties of the core language alone.

The core language consists of a flat network of processes each with one or more explicit *wait* statements. Other aspects of the language such as concurrent statements, concurrent signal assignments (the so-called "dataflow" sublanguage), or processes with static sensitivity lists can all be expressed in terms of the core language. The core language is described in Section 9 of the VHDL '87 [384] or VHDL '93 [387] standards documents.

6.3.2.3 An Examples of Differences and Correspondences

Differences

The example shown in Figure 6-18 is a VHDL entity containing two processes **P1** and **P2**. The presumption is that the process **P1** is invoked by some event on the input signal **M**. The execution profile of this system in this situation is shown in Figure 6-19 under the two different semantics. The execution profile gives a trace of the model's execution at the statement-level. This highlights the granularity of the η -steps and the δ -steps.

```

entity E is
  port(M: in bit);
end E;

signal Q, R: bit := '1';

P1:
process
  variable b: bit := '0';
begin
  (w) wait on M;
  (1) Q <= b;
  (2) if R then
  (3)   b := 1;
  (4) else
       b := 0;
  end if;
end process P1;

P2:
process
  variable Z: integer := 1;
begin
  (w) wait on Q
  (1) if Z then
  (2)   R <= '0';
  (3) else
       Z := next(Z);
  end if;
  (4) ... other stuff ...
end process P2;

```

Figure 6-18. An Example of the Effect of \bar{R}_δ in VHDL

The \overline{RMC} discrete event semantics of VHDL are shown on the top bar of the figure. There, a non-modular condition occurs on **R** which has two different values across the single- δ macrostep. In the \overline{RMC} case the macrostep consists of δ_1 alone. At the start of δ_1 the value of **R** is '1' because that is the value declared in the signal initializer. This is also the value on the signal in η_1 and η_2 . However, at the end of the macrostep, after η_2 is complete, the value becomes '0'. In particular in η_1 when **P1** is executing at statement

2. Specifically, Gonthier [295], part III. In the practical implementation of that theory a simplification is made which postpones reads until after there is no *potential* to execute a write on the signal. This is an instance of a state-insensitive *may* computation is substituted for a state-specific *must* computation.

②, the value read for **R** is '1' which explains why the next statement executed is ③. Later in the trace the value '0' is assigned to **R** at ② in **P2** and this value becomes the final value of **R** at the end of δ_1 and for the macrostep.

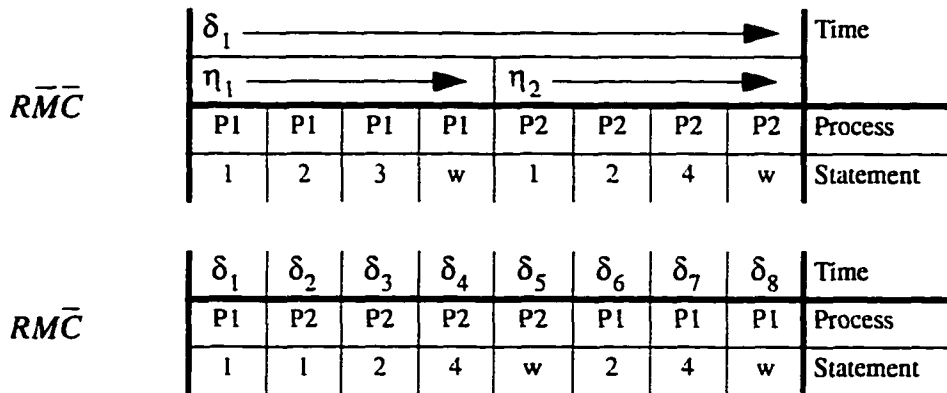


Figure 6-19. Execution where $\overline{RM\bar{C}}$ and \overline{RMC} Semantics Differ

The trace profile of the VHDL under \overline{RMC} semantics is shown in the lower frame of Figure 6-19. That case is subtly different because there is no explicit intra-macrostep suspension operation. Instead the semantics enforces synchronization internally by requiring that any statement which uses the value of a signal must execute *after* the value of the signal has been assigned. Operationally this means that a process runs until it cannot progress due to the lack of information. This is in contrast to VHDL's execution rule which runs a process until an explicit suspension statement is executed.

Synchronous VHDL defines the subset of the language where the \overline{RMC} subsemantics is consistent with the original $\overline{RM\bar{C}}$ semantics. The two frames of Figure 6-19 differ, therefore the example of Figure 6-18 is not part of the Synchronous VHDL subset. This is because the discrete event interpretation of the program exhibits non-modularity which is not exhibited in the synchronous interpretation. The differences between the two frames of Figure 6-19 illustrate two issues that be dealt with in the definition of Synchronous VHDL. The first is that \bar{R}_δ in the general case causes \bar{M} at the macrostep level. The second is that even when both conditions 1 and 2 from Section 6.3.2.1 hold, the execution

granularity of the explicit suspension mode prevents the occurrence of the implicit context switches that occur in the synchronous semantics.

Correspondences

The example of Figure 6-18 can be modified so that it is M and therefore is a part of the Synchronous VHDL subset. That modification is the introduction of an explicit synchronization operation in the form of a `wait` statement which is sensitive to an internally-driven signal, `Q2`. The modified example is shown in Figure 6-18. With this slight mod-

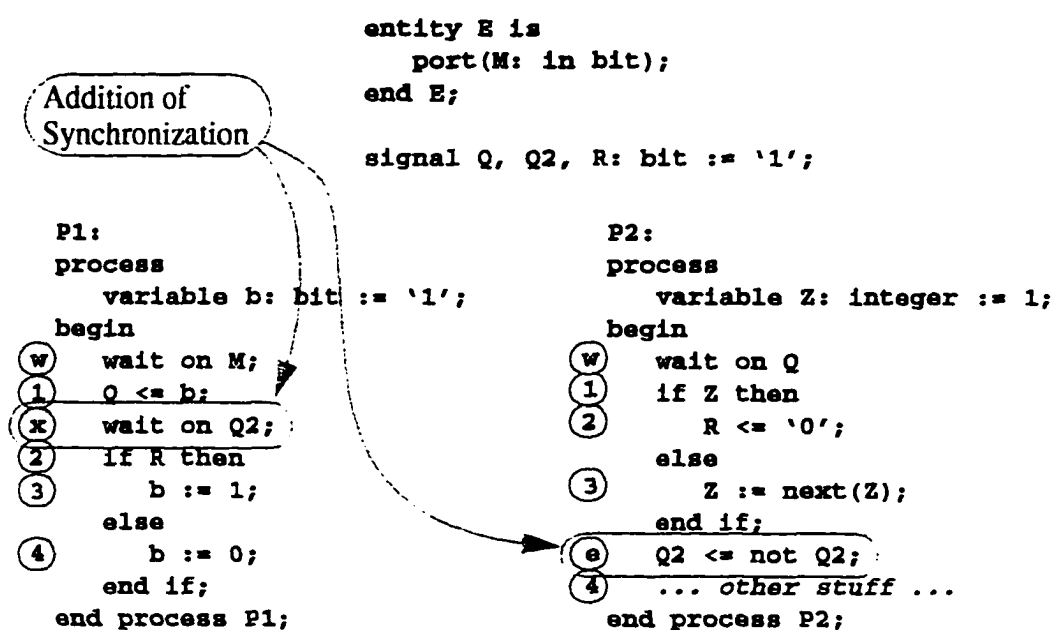


Figure 6-20. A Subtle Modification of Figure 6-18 to Avoid \bar{M}

ification. the execution trace for the \overline{RMC} interpretation and the $R\overline{MC}$ interpretation coincide.

6.3.2.4 Modularity Checking

There are two ways to approach the modularity checking problem. The first is to establish, by examination, that the necessary conditions for modularity hold for all possible execution traces of the program. This is the fully general approach but has the disadvan-

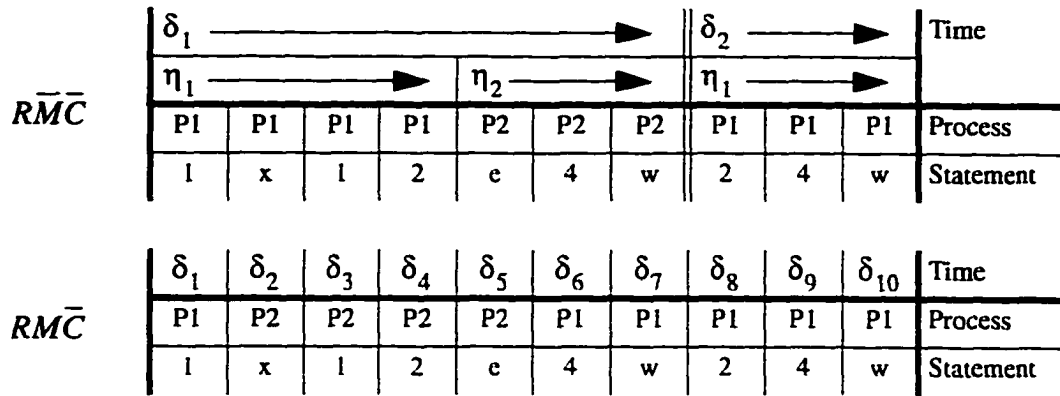


Figure 6-21. Execution where RMC and RMC Semantics Match

tage that it requires showing that modularity holds for all possible executions. This may require a prohibitive amount of analysis. A second approach adopts the view that there exists a reasonable set of *sufficient* conditions which guarantee that the program is modular. In particular the sufficient conditions are evaluated relative to their expressiveness and the analysis effort required by them.

General Modularity Checking

In the general case, the necessary conditions for modularity in discrete event semantics must be established for all feasible executions of the program. The modularity check must address the two necessary features for modularity. Unless otherwise restricted, VHDL allows within a single macrostep,

1. signals to be multiply emitted with distinct values.
2. any signal to be read before being written.

A general modularity check is a dynamic analysis which verifies that neither of these situations occur in any macrostep transition. The general case therefore entails a sort of reachability analysis to establish M in every transition from every state Q in the reachable state space of the model. Exactly this sort of dynamic checking is used in Esterel to establish C . There, the existence of the state-dependent partial order \leq_Q , or at least conservative approximation to it [295], is established for every reachable state Q . In fact, for the particular sort of non-modularity found in VHDL's discrete event semantics, any algorithm

which establishes C also establishes M as a side effect.

That VHDL's \bar{M} property is related to its \bar{C} property is intuitive but not entirely obvious. The relationship between \bar{M} and \bar{C} in discrete events can be readily seen from an analysis of two cases which address conditions 1 and 2 of the necessary conditions for M from Section 6.2.3.1. The first addresses condition 1 and is an analysis of how \bar{C} was established in Section 6.2.3.4. In the example of that section, a situation was constructed in a valid VHDL program where certain signals had multiple emissions in the macrostep. The case of, Figure 6-11 showed the signals $i0$, $i1$, and $i2$ having an infinite number of emissions in a macrostep. This meant that any potential partial order \leq_Q would have to obey the following relationship: $i0 \leq_Q i1 \leq_Q i2 \leq_Q i0$ for every state Q . This relationship is cyclic for any state Q so \leq_Q is never a partial order and the description is \bar{M} . This situation illustrates how multiple output emission in a macrostep introduces cycles in \leq_Q and prevents it from being a partial order. It offers the opportunity that causality analysis might be used to identify violations of condition 1: a modular description cannot have multiple emissions in a macrostep and neither can a causal description, though for subtly different reasons.

The second case addresses condition 2 which ensures that every signal has a single value across the macrostep by preventing any read preceding a write. A VHDL program where a read occurs before a write was shown in Figure 6-18 with the specific scenario being the $\bar{R}\bar{M}\bar{C}$ trace shown in Figure 6-19. In that case the discrete event semantics forces the signal R to be read in step η_1 of δ_1 though it is also written in η_2 . This leaves R with a different value at the end of the macrostep than was observed at η_1 . Letting the state Q_w be the state of the system at the start of that trace, with $P1$ at its \textcircled{w} and $P2$ at its \textcircled{w} as well. The chain of dependency in the trace runs from the read of signal R at $\textcircled{1}$ of $P1$ to the write of R at $\textcircled{2}$ of $P2$. This is denoted as $P1.1 \leq_{Q_w} P2.2$. However, condition 2 of modularity requires that the read on R occur after the write which is denoted

P2.2 \leq_Q P1.1.

The presentation of a compilation procedure that effects these modularity checks is deferred until Section 7.2. Some of the not insubstantial difficulties involved with applying that method to VHDL are described in Section 6.3.2.5.

A Sufficient Condition for Modularity

A specific case where modularity is clearly preserved is the case where each process is run at most once in the macrostep. In that case any non-resolved signal can be defined at most once when its driving process is invoked. All that is required is a further condition which prevents the not-yet-driven signal from being read. Though complex to describe in the abstract, the situation is actually quite simple and extremely common. The syntactic conditions which must hold are:

1. all processes have a fixed sensitivity list,
2. there are no cycles in the communication graph among the processes.

Such a situation is illustrated in Figure 6-22. That case is obviously a contrived example which is *almost* combinational but clearly exhibits sequential behavior because the value on the output **O** depends on the number of times the inputs **A**, **B**, **C** or **D** have changed. In fact, a description of combinational logic is a special case of this class of program because logic networks are by definition acyclic. Thus (acyclic) descriptions in the dataflow subset of VHDL fall into this category as well. In these cases, modularity can be determined from a structural examination of the network and there is no need for any deep semantic analysis.

6.3.2.5 Lessons Learned

From this study which identifies a synchronous subset of VHDL's discrete event semantics, several observations about the interaction between language features and semantics can be drawn. These observations offer both insight into why Synchronous VHDL is

```

entity E is
  port(A, B, C, D: in bit;
        O: out bit);
end E;

architecture A of E is
  signal i1, i2: bit;
begin

  P1:
  process
  begin
    i1 <= A or B;
    wait on A, B;
    i1 <= A and B;
    wait on A, B;
  end process P1;

  P2:
  process
  begin
    i2 <= C xor D;
    wait on C, D;
    i2 <= C nand D;
    wait on C, D;
  end process P2;

  P3: O <= i1 = i2;

end A;

```

Figure 6-22. A VHDL Program where Modularity Checking is Simple indeed somewhat of a stretch as well as prefiguring some of the language evolution proposals reviewed in Section 6.4.

Granularity of Execution

Operationally, VHDL's granularity of process execution under its $RM\bar{C}$ semantics is distinctly different than that of Esterel's synchronous $RM\bar{C}$ semantics. In particular, the discrete event semantics has an *explicit* suspension operator in the **wait** instruction. This contributes to the \bar{M} aspect of the discrete event semantics. In contrast, the synchronous semantics has a notion of *implicit* suspension under which a process suspends, within a macrostep, until the values that it requires become available. A case where explicit and implicit suspension semantics differ is the case shown in Figure 6-18. The salient aspect of that example is that the first process, process **P1**, "overcommits" and reads a signal, signal **R** in the example, before that signal is assigned in the macrostep. This operational

“overcommit” behavior is a heavy contributor to the \bar{M} -ness of VHDL and additionally makes it difficult to write synchronous programs in VHDL; witness the subtle analysis needed to identify the synchronization needed to convert the \bar{M} example of Figure 6-18 to the M example shown in Figure 6-21.

VHDL Event versus Transaction

One issue which has not been explicitly presented here¹ but which has been present in the whole analysis of VHDL’s discrete event semantics is the subtle distinction between the *event* and the *transaction*. In VHDL an event is a change in value on a signal whereas a transaction corresponds to merely a signal assignment independent of whether that assignment changes the value. From the VHDL Language Reference Manual glossary of terms, *event* is defined as:²

Event. An event is said to occur on a signal when the current value of the signal changes as a result of the updating of that signal with its effective value. (Section 12.6.1)

In contrast, a *transaction* is said to occur on a signal whenever its driver becomes active, and the activity of a signal driver is governed by the execution of signal assignments against the driver:³

Active Driver. A driver is said to be active during a simulation cycle in which it acquires a new value, regardless of whether the new value is different from the previous value. (Section 12.6.1)

A trivial example illustrating this distinction is shown in Figure 6-23. Every **Clock1** the value of **OUT** toggles, every **Clock2** the value of **OUT** is reset to ‘0’, when **Clock1** and **Clock2** occur together the result is determined by the bus resolution function for **ResolvedBit** (not shown). The signal **OUT** has an event on it under **Clock1** if and only if its value is different than the value that had been previously assigned to it.

1. The distinction between *event* and *transaction* has also been investigated in previous work [47].

2. From Appendix B of IEEE Std 1076-1987 [384], page B-5.

3. From Appendix B of IEEE Std 1076-1987 [384], page B-1.

```

entity E is
  port(OUT: out ResolvedBit;
        Clock1, Clock2: in Bit);
end E;

architecture A of E is
begin

  process
    variable value: bit := '0';
  begin
    wait on Clock1;
    OUT <= value;
    value := not value;
  end process;

  process
    variable value: bit := '0';
  begin
    wait on Clock2;
    OUT <= '0';
  end process;

end A;

```

Figure 6-23. An Example Distinguishing the Event and the Transaction

In the context of discrete event simulation, the distinction between event and transaction is important for implementation reasons in simulators. The notion of event as a change in value reflects the preoccupation, in simulation, with damping out repeated computations. This is purely a performance issue. Clearly if a signal assignment is made which does not change the signal's value there is no need to recompute and ripple the (non-)change through the process network.

In contrast, the use of the event abstraction in semantic analysis is highly problematic because it obfuscates the information ordering on the signal value domains. In Scott's theory of computable functions, the execution of microsteps represents the progressive refinement of an approximation to the information defined in the macrostep transition relation. A signal assignment which *does cause* an event corresponds to an indication that a better approximation of the information for the microstep can be attained through the execution of more microsteps. An assignment which does *not cause* an event corresponds to having reaching a fixed point in this approximation. This means that the information ordering fixed point for a macrostep is path-dependent in addition to being dependent on

the system state at the start of the microstep. In short, while Scott's theory of computable functions states that a minimum fixed point which is constructible by successive approximations necessarily exists for any finitely computable function, exactly what that information ordering relationship actually is in a discrete event semantics is entirely non-obvious.

An obvious and much more reasonable information ordering and macrostep approximation scheme is one which is tied directly to the output assignment operator. This is the notion of assignment introduced for the synchronous semantics of Example 4.3.4.4, namely that in a macrostep, the output emission operator directly affects the information content of a value carrier: before an output has been assigned, the signal is *undefined*, after the signal has been assigned it is *present*, when it is never assigned in the macrostep then it is *absent*. The synchronous semantics' output assignment always defines information, there can be no duplicate definitions in a macrostep, and a value can be consumed only once it is not *undefined*. Unfortunately this definition coincides with VHDL's notion of transaction rather than event.

The Subtleties of the δ -time Model

Intimately associated with the distinction between event and transition are the subtleties of the δ -time model. These differences were presented in Section 6.3.2.2 and can be characterized in a number of different though related ways. There, the time model of discrete events was characterized as having three levels with a macrotime, a δ -time and a η -time. Between the three different levels drastically different relationships pertaining to how multiple lower-level transitions were aggregated into the upper-level step held. In particular, the η -time level induced \bar{R}_δ yet R held at the macrotime level. When examined from the perspective of the image computations the difference appeared as $F_\eta\{Q\}$ and $B_\eta\{Q\}$ being (trivially) monotonic because of \bar{R}_δ , while the same could not be said of the microstep image computations $F_\delta\{Q\}$ and $B_\delta\{Q\}$.

States versus Outputs

Moving from the semantic attributes of Synchronous VHDL upwards to the language level, some comments on design description style can be offered. One of the most common situations which arise in the use of the Synchronous VHDL subset is the desire and the need to confuse the internal state of a process with its outputs. It is many times convenient in VHDL to treat a signal driven by a process as an output of that process. However, since it is also possible, courtesy of \bar{R}_S , to read the value of such a signal in the same macrostep where it written it is convenient to write descriptions which are \bar{M} . This leads to a confusion in the mind of the programmer between whether a signal is truly an output and whether a signal is actually an implicit state variable of a process. In an M semantics, output by definition cannot hold state across the macrostep boundary. Yet VHDL's \bar{R}_S property allows such state to be held and thereby induces \bar{M} . The example of Figure 6-24 illustrates this condition with two possible implementations of an oscillator, one which is M and one which is \bar{M} .

```
entity Oscillate is
    port(Clock: in bit; OUT: inout bit);
end Oscillate;

architecture DES of Oscillate is
begin
    process
    begin
        wait on Clock;
        OUT <= not OUT;
    end process;
end DES;

architecture SYNC of Oscillate is
begin
    process
        variable value: bit;
    begin
        wait on Clock;
        OUT <= value;
        value := not value;
    end process;
end SYNC;
```

Figure 6-24. An Example of the Confusion of State and Output

The architecture **DES** offers a natural implementation of the oscillator under a discrete event semantics. In that case the signal **OUT** is both the carrier for the output of the oscilla-

tor *and* at the same time the state variable remembering the phase of the oscillator. The example is not \overline{M} because the signal **OUT** is read before it is written in each macrostep; **OUT** necessarily has two values in the macrostep. In contrast the architecture **SYNC** maintains the phase of the oscillator separately and distinctly from the output.

The Flat versus Nested Process Model

Finally, there is one language level attribute missing from VHDL which is absolutely crucial in practical situations. VHDL describes systems in terms of a flat network of processes interconnected by signals which are interpreted as the outputs of their driving processes. One can view the language as consisting of two separable parts: the structural composition constructs and the individual processes themselves. The structural composition aspects don't contribute to the semantics, rather it is the networks of processes which are actually operated upon by simulators.

Processes under this view, are for practical purposes, finite state and the system description is a network of communicating finite state processes. While the communicating finite-state process model is a general enough describe any finite state system, it is not *convenient for programmers*. In particular, while a structural unit can contain other structural units in an arbitrarily deep hierarchy, the behavioral unit, the process, is atomic.¹ What would be convenient is the ability to arbitrarily nest structural and behavioral components and to have behavioral components which govern more than one thread of control. Such capabilities exist in the synchronous language Esterel with its nested process model or in the discrete event language StateCharts with its nested states. There is yet no officially-sanctioned extension to VHDL which supports these features.

1. This had been previously observed by Narayan *et al.* [555] and has since formed the basis for the SpecCharts extension to VHDL.

6.4 Extending the Standards

There have been other attempts at extending the discrete event languages to make them more amenable to various uses. These are cases of language evolution as described in Section 6.1. They do not however represent unmitigated progress in the sense that the proposed changes make for a better semantic definition of the language. The two cases described here are the recent revision of the VHDL standard, and the SpecCharts proposal for a hierarchical behavioral construct in the flat process model of VHDL.

6.4.1 VHDL '93

From the perspective of this work the new VHDL standard, VHDL '93¹ [387], does not represent a positive advance in the evolution of the language. The changes to the language involved some modifications to the original standard [384] as well as some new capabilities. The modifications involved the clarification and resolution of various inconsistencies and ambiguities which had been observed in the original standard [385] [386]. The new capabilities added include groups, shared variables, hierarchical path-names and the ability to define foreign-language models (*e.g.* compiled from C), shift and rotate operators and an enhanced inertial delay modifier. As well, the standard document itself was recast to highlight the structure of the language itself in terms of a static design hierarchy aspect and a dynamic execution aspect which was further distinguished into a core language and syntactic conveniences defined in terms of this core. The increased emphasis on the core language made the standard more concrete as well as clarified the orthogonality of the many kinds of declarations and statements in the language.

The clarifications to the dynamic aspect of the original standard are substantially transparent relative to the semantic theory developed in this work. The same can be said of the

1. The name *VHDL '93* is used here as that is the date-modifier adopted by the IEEE in the name of the new revision of the standard: IEEE Std 1076-1993. This revision is also variously referred to as *VHDL 92* as 1992 was the original target date for the first five-year revision of IEEE Std 1076-1987 [384].

new extensions designed to support design hierarchy compilation and management. What is of interest are some of the new behavioral constructs. These are significant because they provide new capabilities that must either be incorporated into existing language subset definitions or be found to be incompatible with non-simulation uses of the VHDL language such as synthesis and formal verification. Fortunately there are only a few of these and their definition and effect can be dealt with succinctly.

6.4.1.1 The Unaffected Waveform Constant

A new waveform constant **unaffected** has been added. The waveform causes the driver of the signal to be unchanged rather than disconnected. This makes the **unaffected** waveform different than the **null** waveform which does cause a driver disconnect. An example of this waveform is shown in Figure 6-25.¹

```
S <= unaffected when Input_pin = S'DrivingValue else
      Input_pin after Buffer_Delay;
```

Figure 6-25. An Example of the **unaffected** Waveform

The **unaffected** waveform constant is designed to allow for more efficient simulation. The basic idea is that **unaffected** indicates that no change should occur on the signal driver in that case. The new constant is necessary because in the dataflow style there is no other way to express the *lack* of an assignment.

6.4.1.2 Pure and Impure Functions

A formalization of the notion of pure and impure functions was introduced in the revised standard. Whereas the original standard required that all functions be pure, the revised standard allows for functions that have side-effects. Usage restrictions are defined so that impure functions cannot be used where a pure function is expected. In particular an

1. From IEEE Std 1076-1993 [387], page 133.

impure function cannot be used as a bus resolution function.

6.4.1.3 Pulse-Width Rejection Inertial Delay

A new modifier for inertial delay signal assignment was added which allows for a specified pulse rejection width to be used in the signal assignment. This allows for a continuum of delay models to be defined ranging from transport delay on the one extreme to inertial delay on the other extreme. The example shown in Figure 6-26¹ illustrates how transport delay and inertial delay from VHDL '87 are but instances of the more general pulse-width rejection delay. Any policy for VHDL use that proscribed the old inertial and transport delay constructs would necessarily prohibit the use of the more general pulse-width rejection delay.

```
-- Assignments using inertial delay

-- The following three assignments are equivalent to each other:
Output_pin <= Input_pin after 10 ns;
Output_pin <= inertial Input_pin after 10 ns;
Output_pin <= reject 10 ns inertial Input_pin after 10 ns;

-- Assignments with a pulse rejection limit less
  than the time expression
Output_pin <= reject 5 ns inertial Input_pin after 10 ns;
Output_pin <= reject 5 ns inertial Input_pin after 10 ns,
              not Input_pin after 20 ns;

-- Assignments using transport delay
Output_pin <= transport Input_pin after 10 ns;
Output_pin <= transport Input_pin after 10 ns,
              not Input_pin after 20 ns;

-- Their equivalent assignments
Output_pin <= reject 0 ns inertial Input_pin after 10 ns;
Output_pin <= reject 0 ns inertial Input_pin after 10 ns,
              not Input_pin after 20 ns;
```

Figure 6-26. Examples of the “Pulse-Width Rejection” Inertial Delay

1. From IEEE Std 1076-1993 [387], page 116.

6.4.1.4 Shared Variables

In the original standard, VHDL '87, the only vehicle for inter-process communication was the signal. In VHDL '93 it is possible to transmit information from one process to another by means of a shared variable. The variable is typically declared at the level of the architecture and is then read and written by more than one process. There are very few restrictions on their use:¹

More than one process may access a given shared variable; however if more than one process accesses a given shared variable during the same simulation cycle (see 12.6.4), neither the value of the shared variable after the access nor the value read from the shared variable is defined by the language. A description is erroneous if it depends on whether or how an implementation sequentializes access to shared variables.

The example shown in Figure 6-27² illustrates how ill-defined the semantics of shared variables are, relative to the standard (consider when **PROC2** runs before **PROC1** in the first δ -step).

Shared variables are clearly designed for high-performance simulation where the perceived cost of inter-process communication through signals is considered prohibitive. Alternatively they may find application in modeling situations where the partition of the description into processes and signals is at best forced. In either case, the expected application is clearly in the simulation and modeling arena and not that of the specification for synthesis or formal verification.

6.4.1.5 The Postponed Process Class

In VHDL '87, all processes were of the same class in that any process was runnable whenever the signals on its sensitivity list had events. There was a need however for a second class of processes which were guaranteed to be run only after all other processes had

1. From IEEE Std 1076-1993 [387], page 56.

2. From IEEE Std 1076-1993 [387], page 57.

```

architecture UseSharedVariables of SomeEntity is
  subtype ShortRange is INTEGER range 0 to 1;
  shared variable Counter: ShortRange := 0;
begin
  PROC1: process
  begin
    Counter := Counter + 1; -- the subtype check may fail
    wait;
  end process PROC1;

  PROC2: process
  begin
    Counter := Counter - 1; -- the subtype check may fail
    wait;
  end process PROC2
end architecture UseSharedVariables;

```

Figure 6-27. The Nondeterminism of a Shared Variable
 been run. The requirement in modeling was to ensure that some processes only saw the stable values on signals. In VHDL '93 this second class of processes is called postponed processes. A postponed processes is run only after the last δ -step of the macrostep.

Syntactically, a postponed process is indicated with the **postponed** keyword which can decorate both the **process** construct itself as well as the concurrent variants of signal assignments, procedure invocations and assertions. In allowing arbitrary activity to be deferred to the end of the macrostep, the postponed process generalizes the **finally** assertion of VAL.

Other than being executed at a different stage of the simulation, the semantics of the postponed process is exactly that of the non-postponed variety. The deferred execution is the supported by an extended definition of the original simulation cycle.¹ The new VHDL '93 simulation cycle is shown in Figure 6-28.² That six-part definition extends the original

1. From IEEE Std 1076-1987 [384], Section 12.6.3, page 12-13.

2. From IEEE Std 1076-1993 [387], Section 12.6.4, page 169.

simulation cycle definition with the addition of Step (g). That step runs the postponed processes which have become activated during the previous delta steps.

- A simulation cycle consists of the following steps:
- a) The current time T_c is set equal to T_n . Simulation is complete when $T_n = TIME'HIGH$ and there are no active drivers or process resumptions at T_n .
 - b) Each active explicit signal in the model is updated. (Events may occur on signals as a result).
 - c) Each implicit signal in the model is updated. (Events may occur on signals as a result)
 - d) For each process P , if P is currently sensitive to a signal S and if an event has occurred on S in this simulation cycle, then P resumes.
 - e) Each non-postponed process that has resumed in the current simulation cycle is executed until it suspends.
 - f) The time of the next simulation cycle, T_n , is determined by setting it to the earliest of
 1. $TIME'HIGH$,
 2. The next time at which a driver becomes active or,
 3. The next time at which a process resumes.If $T_n = T_c$, then the next simulation cycle (if any) will be a *delta cycle*.
 - g) If the next simulation cycle will be a delta cycle, the remainder of this step is skipped. Otherwise each postponed process that has resumed but has not been executed since its last resumption is executed until it suspends. Then T_n is recalculated according to the rules of step f. It is an error of the execution of any postponed process causes a delta cycle to occur immediately after the current simulation cycle.

Figure 6-28. The VHDL '93 Simulation Cycle

6.4.1.6 Conclusions

The previous five examples highlight the semantic additions in the new standard. They

illustrate how the new revisions have fixed discrete event semantics of the language even more tightly to the simulation cycle. In short, these additions show that while the language has *evolved*, it has not *progressed* in the sense of being more semantically reasonable. In fact, the previous five examples show that little has been done to reorient VHDL towards something which can be interpreted as a specification. Of the specification-oriented problems in VHDL, that the semantics is \overline{RMC} still exists as does lack of a hierarchical behavioral primitive.

6.4.2 SpecCharts

A proposal that does address the lack of a hierarchical behavioral construct in VHDL is the SpecCharts of Gajski, Gong, Narayan and Vahid.¹ The SpecCharts are a synthesis of the graphical state-based description formalism of Harel's StateCharts [330] [331] with the programmatic sequential code-fragment description of behavior found in VHDL.

At a primary level the SpecCharts formalism allows for the attribution of the hierarchical states of StateCharts with program fragments defined in the VHDL language. At a deeper level however, the proposal goes beyond that in giving material (that is textual) form to the graphical formalism and thereby addressing the problem of VHDL's lack of a hierarchical behavioral construct. Additionally, the method by which the new syntactic structures are grafted onto the existing standard language offers some insights both into the ways in which new language features can be tested experimentally using existing implementations as well as into the subtle constraining effects imposed by microstep semantics on these exploratory ventures.

1. The SpecCharts have evolved substantially since their first publication [552] [553] [696] [697] [554] [555]. The presentation here draws from more recent work [275] [277] which has produced the textual representation of the language as well as formalized the semantic model underneath the mixed textual-graphical notation.

6.4.2.1 The Program State Machine Model

The model of computation in standard VHDL consists of a flat space of processes interconnected by signals. Within a process there is only one sort of behavioral description: sequential code. External to processes there is no way for one process to exert control over another process except by emitting a value on a signal; there is no notion of one process completing and causing another sibling to start or of one parent process “aborting” its children based on some watched-for condition. The SpecCharts are based on a computational model which provides all of these features. The authors call this model the Programmatic State Machine (PSM).

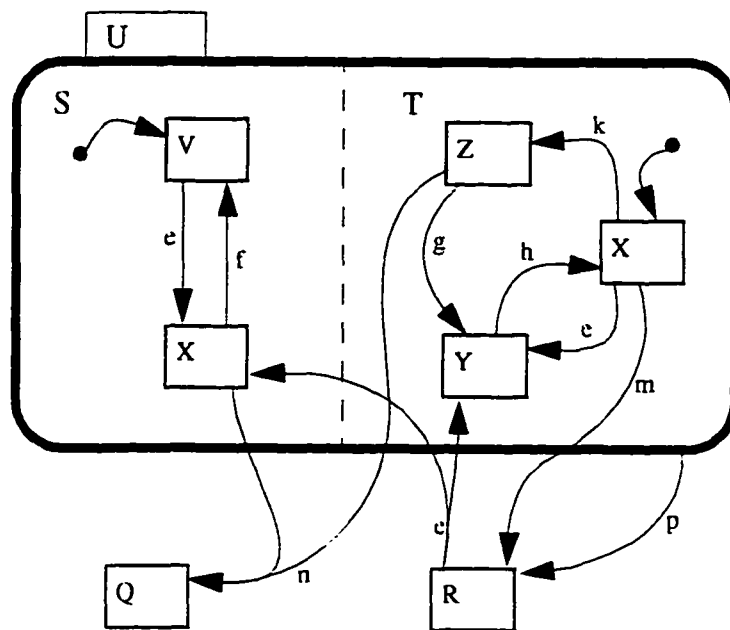
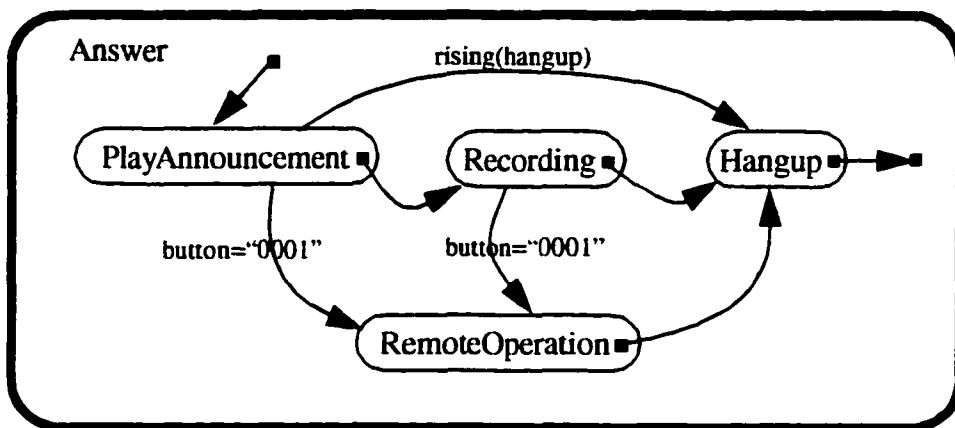


Figure 6-28. The AND and OR States of a StateChart

A PSM is made up of two components, a hierarchical concurrent finite state machine (HCFSM) and fragments of a sequential programming language. The HCFSM formalism is simply a generalization of the StateCharts notation with its hierarchical states. Within the hierarchy the levels are either be “or” levels indicating that the HCFSM state is the dis-

junction of the states at that level, alternatively the level can be an “and” level indicating that the HCFSM state is the conjunction of the states at that level. The example of Figure 6-28¹ illustrates such a decomposition: the state **U** is an “and” level consisting of **S** and **T**, the state **T** is an “or” level consisting of **X** or **Y** or **Z**. The incremental contribution of the PSM over the original HCFSM is the addition of program fragments in the states. This gives a formalism in the style of Figure 6-29² that associates code fragments with the states.



```

behavior PlayAnnouncement type code is
begin
  ann_play <= '1';
  wait until ann_done = '1';
  ann_play <= '0';
end;

```

```

behavior RecordMag type code is
begin
  ProduceBeep(1 s);
  if (hangup = '0') then
    tape_rec <= 1;
    wait until hangup = '1' for 100 s;
    ProduceBeep(1 s);
    num_msgs <= num_msgs + 1;
    tape_rec <= '0';
  end if;
end;

```

Figure 6-29. SpecCharts adds Program Fragments to StateCharts

In a PSM, the code fragments can be associated with any state, even hierarchical states. There are therefore two possibilities for when a state transition can occur in a PSM. The

1. Adapted from Harel et al. [334], Figure 6, page 407.

2. Adapted from Gajski et al. [277], Figure 4.8, page 127.

first is immediately when the condition on the edge occurs. This is called a *transition-immediately arc* (TI). The second time is upon completion of the code fragment for the state subject to the edge. This is called a *transition-on-completion arc* (TOC). Formally any arc in the PSM is represented as a triple (T, C, NS) where $T \in \{TOC, TI\}$ is the type, C is the condition governing when the transition is valid and NS is the next state.

6.4.2.2 The Duality of Graphical and Textual Representations

What is fascinating about the SpecCharts formalism is that it at once draws from the graphical aspects of StateCharts and the textual representation of VHDL. While earlier presentations of SpecCharts [554] [555] concentrated on the graphical aspects, more recent developments have produced an equivalent textual representation for the artifacts of the notation [277].

The new language structure is called the *behavior* and it supports the two kinds of hierarchy in HCFSM through its *type*. The leaf-level behavior carries the type *code* and follows the pattern of the *process* in traditional VHDL. That use is illustrated in Figure 6-29. For the hierarchical states of the HCFSM, the “or” level has the type of *sequential subbehaviors* while the “and” level has the type of *concurrent subbehaviors*. Additionally there is a notation for describing the two different kinds of edges, TOC and TI, between behaviors. These uses are illustrated in the partial example of Figure 6-30¹ which defines a hierarchy of concurrent and sequential behaviors ultimately terminating at the leaf level with behaviors of type code.

What is significant about the SpecCharts textual description is that it has an analogous graphical description which is immediately derivable from the textual form. Or alternatively, for every graphical representation of a SpecChart there is an analogous textual rep-

1. Adapted from Gajski *et al.* [277], Figure 3.17, page 106.

```

entity E is
  port (P: in integer; Q: out integer);
end E;

architecture A of E is
begin
  behavior B type concurrent subbehaviors is
    type int_array is array(natural range <>) of integer;
    signal M: int_array(15 downto 0);
  begin
    X: (TOC, true, complete);
    Y: (TOC, e3, complete);
    Z;

    behavior X type sequential subbehaviors is
    begin
      X1: (TI, e1, X2);
      X2: (TOC, e2, complete);

      behavior X1 type code is
        ...
      end X1;
      behavior X2 type code is
        ...
      end X2;
    end X;

    behavior Y type code is
      variable max: integer;
    begin
      max := 0;
      for j in 0 to 15 loop
        if (M(j) > max) then
          max := M(j);
        end if;
      end loop;
    end Y;

    behavior Z type code is
      ...
    end Z;
  end B;
end A;

```

Figure 6-30. A Sample SpecCharts Specification

resentation. The corresponding graphical representation for the example of Figure 6-30 is shown in Figure 6-31.¹

6.4.2.3 The Semantics of SpecCharts

The PSM model and by extension SpecCharts are but a syntactic extension of the under-

1. Adapted from Gajski *et al.* [277], Figure 3.21, page 111.

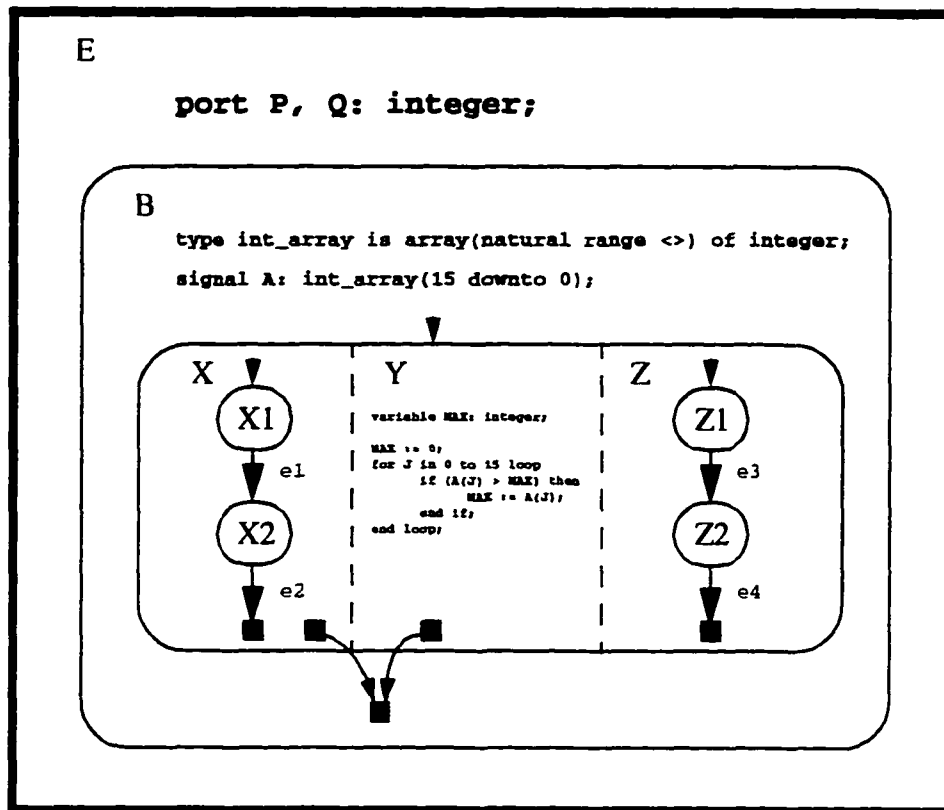


Figure 6-31. Equivalent SpecCharts Graphical Representation

lying semantics of VHDL. Underneath all of the notation, whether graphical or textual, the semantics is still the \overline{RMC} semantics of VHDL. This can be observed by noting that the PSM model is a structuring of the sequential aspect of VHDL in terms of the hierarchical state graphs of Harel's StateCharts. Thus the semantics of SpecCharts is some combination of the discrete event semantics of VHDL aggregated according to the semantics of StateCharts.

In Section 6.2.3, the semantics of VHDL were shown to be \overline{RMC} . Too, the semantics of StateCharts were shown in Example 4.3.4.6 to be \overline{RMC} . Interestingly, a close inspection of both systems shows that at a fundamental level, the semantics of StateCharts and unit-delay VHDL are exactly the same:

- both VHDL and StateCharts are R , additionally both are \bar{R}_δ ,
- both are \bar{M} with the macrostep inter-process communication being dependent upon the δ -step order in which outputs are produced,
- both are \bar{C} because there exist descriptions for which there is no finite series of δ -steps defining a macro step.

The strong conclusion that one must draw from this is that SpecChart's syntactic structuring of the design into hierarchical and concurrent finite state machines does nothing to enhance or restrict the semantics of the formalism. So while the expressiveness and convenience of the SpecChart formalism can be seen an important step forward in the evolution of system description languages, it offers little in the direction of a sound semantic basis for such descriptions.

6.4.2.4 The Future Evolution of SpecChart Semantics

That the SpecCharts adopts the semantics of VHDL and StateCharts directly offers a possibility for the future evolution of the formalism. Indeed, the semantics of StateCharts underwent some evolution after its original presentation and there is no reason that this evolution could not be reflected in a modified semantics for SpecCharts. The original semantics of StateCharts given by Harel [330] had the $\bar{R}\bar{M}\bar{C}$ property. Later modifications by Harel, Pnueli *et al.* gave an $\bar{R}\bar{M}\bar{C}$ semantics [333]. Finally Maraninchi developed in Argos [501] a StateCharts-like notation with the $\bar{R}\bar{M}\bar{C}$ property of the synchronous semantics. As well, an evolution of VHDL's $\bar{R}\bar{M}\bar{C}$ semantics to the synchronous $\bar{R}\bar{M}\bar{C}$ semantics was described in the Synchronous VHDL subset of Section 6.3.2. This offers the potential for a union of the $\bar{R}\bar{M}\bar{C}$ semantics of Argos and Synchronous VHDL using the hierarchical **behavior** construct. The definition of such a language is clearly feasible and would seem to be a very attractive, providing a synchronous semantics while at the same time addressing one of the major deficiencies of Synchronous VHDL: the constraint of the flat process model of VHDL '87.

6.5 Review

The difficulty in the analysis of discrete event semantics can be seen by placing the semantics of these languages within the framework of the RMC Barrier Theorem. This is accomplished in Section 6.2.3. This setting explains at the semantic level the problems that are inherent in the discrete-event basis of these languages. At the same time it sets up the proposals for moving beyond system specifications based on discrete event operational models. Two of these are the annotation language approach and the identification of a constrained discrete event regime that preserves desirable semantic properties. These two approaches are typified by the VHDL Annotation Language (VAL) [35] and the Synchronous VHDL subset [47] respectively.

From this historical perspective one can look to developments coming to fruition in the near future such as the new VHDL '93 standard [387] or the addition of hierarchical behavioral constructs into VHDL (SpecCharts) [277]. These developments do not directly add to the understanding of the semantics as they merely build upon the existing discrete event semantic model of VHDL '87. Their significance however is that they offer new language constructs that introduce known features into the standard language framework. In the sense used here, this integration is the essence of language design evolution in the form of the adoption of what were previously considered experimental features into mainstream use.

7 The Non-Deterministic Abstract Machine

The theoretical framework established in the early chapters coupled with the analysis of the later ones encapsulates a method of semantics-directed language design. The perspective so far has consistently been theoretical and observational; oriented at explaining, critiquing and classifying. It remains to put the ideas and philosophy developed in this work into a concrete form in a practical design setting. This chapter therefore is dedicated to an in-depth presentation of an internal representation (IR) which has been designed using the ideas presented in the previous chapters: computational semantics, microsemantic analysis and the RMC Barrier Theory. In contrast with the previous chapters, the focus here is explicitly illustrative and designed to give the reader a view of how the semantic theory described previous can be put into practical use.

The Non-Deterministic Abstract Machine (NDAM) is a semantics-directed machine architecture oriented at the Synchronous Languages. As an exercise in the design of a semantics-directed internal representation, the abstract machine draws from the elements highlighted in each of the previous chapters. In particular:

- From Chapter 2, the denotations of programs executing on the machine are transition relations. By extension, the instructions of the machine are instances of these transition relations in the small.
- From Chapter 3, the operational semantics of the machine is non-abstract and defines a sort of computational semantics. A microsemantic analysis shows that it is substantially similar to the δ -time presented in Section 3.4.3.

- From Chapter 4, the microsemantic analysis shows that the semantics is $RM\bar{C}$ and thus the RMC Barrier must be surpassed. This is done by disallowing non-causal programs. Therefore, the NDAM has a causality checking problem.
- From Chapter 5, the NDAM is an applied semantics in the sense that it is explicitly oriented at the domain of imperative-style system description languages. This in contrast to being oriented at dataflow networks or the hierarchical finite state machine.
- From Chapter 6, the specific system description languages of interest are synchronous subsets of the standardized HDLs and of course Esterel which was the original imperative synchronous programming language.

The result is an internal representation which is designed to be suitable for a number of languages as is depicted in Figure 7-1.¹

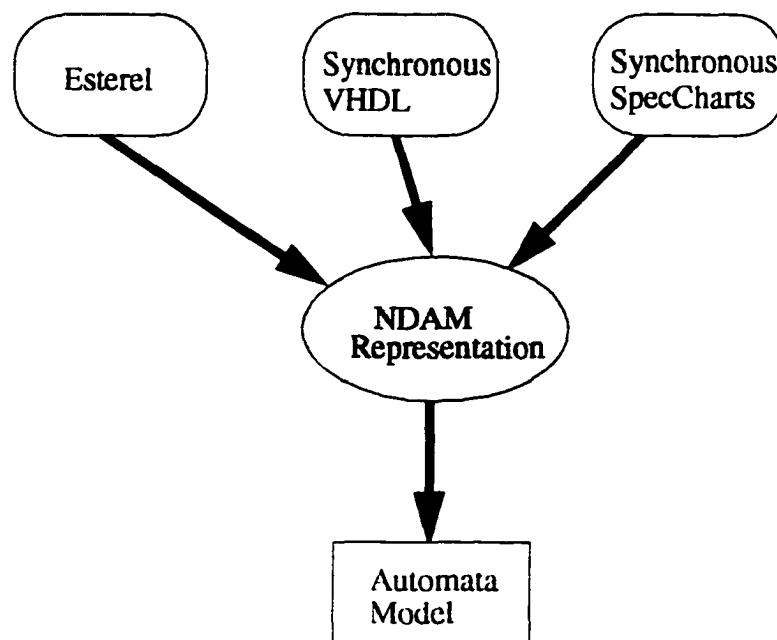


Figure 7-1. The NDAM Representation as a Non-Abstract Semantics

In fairness, it must be pointed out that, at this writing, there are a number of such internal representations. At this point, the NDAM is no means unique in its addressing the Synchronous Languages. Other semantics-directed internal representations for synchro-

1. With the appropriate modularity checking procedure, as was defined for Synchronous VHDL in Section 6.3.2.4, a sort of “Synchronous Verilog” could equally appear in that diagram.

nous languages that can be named are:

- The SL Languages [64] [65] were presented in Section 5.7.2 as an one such instance. One which is oriented at dataflow-type representations. These instructions have been formalized in the **gc** internal representation [577] used by the Lustre and Signal compilers.
- The **ic** and **oc** internal representations [577]¹ are used by Esterel v3 compiler [176]. These two internal representations have an instruction-set form and a tabular representation respectively.
- Finally, with the advent of “the hardware semantics” [78] [264] [525], any representation of combinational logic and latches (*e.g.* BLIF [109], BLIF-MV [107]) can be interpreted as a semantics-directed internal representation for the synchronous languages.

The following sections provide an overview of the NDAM concentrating on the design principles used, the causality checking problem, its computational semantics, and the application issues of high-level language compilation and a runtime generation. The presentation here highlights the interesting and distinguishing features. A detailed exposition of the assembly language form can be found in Appendix A.

7.1 The Abstract Machine

The abstract machine is based on the usual model of an instruction set processor with extremely simple instructions and an unbounded number of registers [10]. The intent is that the preliminary translation from the high-level language will exploit this feature, leaving to a later optimization phase the collapse of temporaries into a much smaller number of “hard” registers. The abstract machine is distinguished from other more traditional internal representations of compilers (*e.g.* PCCIR [345] or U-CODE [172]) in three ways. First, the notions of *process* and concurrency are explicit in the internal representation.

1. The design of the NDAM as the internal representation for Synchronous VHDL [47] somewhat predates the formal standardization of the definitions of **lc**, **oc** and **gc**. I am most grateful to Albert Benveniste and Gérard Berry for their invitation and support in the spring of 1994 during which I translated the standards document [577].

It is now clear that the NDAM, save for the detailed syntax, is substantially similar to **lc**. The sole substantial difference is the operational definition of the **TCWC** instruction (defined in Section 7.1.3.3) in terms of a dynamic fixed-point. This technique and its effect on implementations is detailed in Section 7.5.

Secondly, and related to concurrency, the representation of inter-process coordination and of synchronization are explicit as well, taking the form of a special class of variables called *signals* that are subject to two restrictions: within a step, they have the property of single assignment and they must always be written before they are read.¹ Finally there is an explicit notion of *exceptions*, whereby one portion of the design can preempt other portions and return control to a higher level in the nested process structure.

7.1.1 Design Constraints

In addition to these major features of the representation which are oriented directly at supporting the synchronous semantics within an imperative style, there are certain other aspects to the design as well. These relate to its sufficiency and generality and are related to the two uses of a non-abstract semantics. On the one hand, viewing the diagram of Figure 7-1 from the high-level language downward, there is an issue of expressive sufficiency: namely can the internal representation support, in a reasonable and cogent fashion, the conceptual structures from the source language. On the other hand, from the semantic model upward, there is the issue of the semantic characteristics: namely can the internal representation represent a reasonable number of the feasible mathematical structures of the denotational model.

7.1.1.1 Expressive Sufficiency

One view of the expressive sufficiency is defined by Gajski *et al.* [697] [277] [698] in the Program State Machine (PSM) model. A representation and even a specification language can be measured against how it supports the following seven attributes:

1. These restrictions ensure modularity. The theoretical basis for them was presented in Section 3.4.3.2 as the domain definitions of outputs in δ -time.

1. *Hierarchy*

Is there support for hierarchy in structure and behavioral aspects? Structural hierarchy would include features such as instances or macros while behavioral hierarchy would include nested states or processes and subprograms.

2. *State transitions*

Is there a notation for directly describing states and transitions between those states?

3. *Programming constructs*

Is there a notation for describing sequential blocks of program-type code?

4. *Concurrency*

Is there an explicit representation of concurrency? This includes the representation of so called “macro” concurrency between coordinating processes as well as the so called “micro” concurrency describing the independence of certain subexpressions in an otherwise sequential body of code.

5. *Exception handling*

Is there a way to express non-standard exit paths? For example, a desirable language feature is the ability to declare that under some circumstance, one portion of the (concurrent) design preempts the execution of the rest of the system and returns control to a higher authority.

6. *Completion*

Is there a way to express that some sub-portion of the design has completed its computation is now “done?” In particular, once the termination has occurred, can control be recovered by a higher authority. There exist representations such as ω -automata where termination is explicitly not a part of the definition.

7. *Equivalence of state transitions and code*

Is there some way to treat the state-transition representation mentioned in *item 2* in an equivalent manner as the programming language aspects of *item 3*.

These characteristics relate most directly to the specification language itself. Indeed, the argument of Gajski *et al.* [277] is that all seven of these attributes are necessary in a proper specification language. Further, they argue that only the SpecCharts extension of VHDL provides all of these attributes to the programmer. Their argument is directed exclusively at the properties of the specification language and explicitly avoids any analysis of semantics. As was pointed out in Section 6.4.2.3, the semantics of SpecCharts is fundamentally that of VHDL: it has three levels and is $R\bar{M}\bar{C}$ and $\bar{R}_8M_8C_8$. In contrast the focus of this work is exclusively with the conditions when synchronous semantics is well-defined to

the exclusion of language features. However, in designing an intermediate representation, both views must be taken into account.

7.1.1.2 Semantic Characteristics

From the perspective of the semantic model upwards, there are certain properties that must obtain in the semantics of the internal representation for it to properly represent a synchronous system. These properties were presented in Section 5.7.1 with explanation. They are reiterated here briefly:

1. Perfect Synchrony
2. Multiform Time
3. Projective Semantics
4. Concurrency
5. Determinism

Of these five, the only one which requires further mention is *Item 5* stating that synchronous semantics are deterministic. Yet the very title of NDAM contains the word *nondeterminism* in it, so how are these two views reconciled?

There is a subtle but important distinction which must be made in the different kinds of nondeterminism which can exist in a non-abstract model. The first kind is a sort of “declarative” sort of nondeterminism which states that there exist multiple possibilities that are not constrained from within the semantic model. This sort of nondeterminism has been called *selection nondeterminism*.¹ The second sort of nondeterminism is a sort of “imperative” nondeterminism which is used to model concurrency as the nondeterministic interleaving of multi-step paths in separate modules. This sort of nondeterminism has been called *ordering nondeterminism*. Of the two, selection nondeterminism is wholly consistent with the computational semantics of synchronous languages because it does not

1. Gajski *et al.* [277], page 83.

destroy modularity. In contrast, ordering nondeterminism is, on its face, inconsistent with synchronous semantics because it is non-modular. This can be seen in the very definition of modularity from Section 4.1.2 and from the two \bar{M} microsemantics in Eq 4.3.4.3, Eq 4.3.4.5 and Eq 4.3.4.6.

The nondeterminism supported by the non-abstract semantics of the NDAM is exclusively selection nondeterminism and never ordering nondeterminism.

7.1.2 Structural Constructors

An NDAM description consists of two sorts of constructions. There are structural constructors which define the physical hierarchy of the system and its interface to the outside world. The other sort of constructor are the behavioral constructors which define what the network does at the δ -time level, and by extension at the macrotime level. This section defines the structural constructors.

7.1.2.1 The Network

The network is defined in terms of the data types available within it, the signals it exports to the outside world, the signals available within it and its processes. The network's interface to the outside world is exclusively through the exported signals. This is depicted in the diagram of Figure 7-2.

The network declaration is as follows:

```
network N(NAME) is  
  { type-declaration  
    signal-declaration  
    input-or-output-declaration  
    process-tree-declaration }  
end network N(NAME)
```

The types and signals declared at the top level of the network are visible to all processes in the network. The *input-or-output-declaration* defines which declared signals

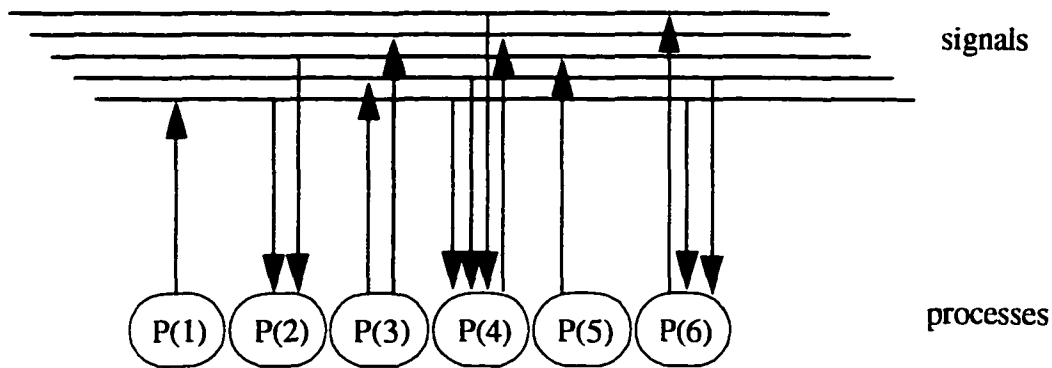


Figure 7-2. A Network of Processes Communicating via Signals

constitute the interface of the network. The remainder of the signals are thus considered internal to the network though are visible to all process.

7.1.2.2 The Process Tree

The network declaration is purely a structural artifact that aggregates and hides all internal structure. The major internal structure of the network is its processes. Processes are hierarchical entities, being able to contain other processes. These subprocesses are organized into a *process tree* that defines the call-callee relationship between the processes. The process tree is a static definition of the parent-child relationship between processes. However, while the process tree is static, the invocation of processes in the tree is a dynamic artifact of the behavior of a process. As such, all the process tree does is explain the behavioral and structural nesting of (possibly concurrent) control flows within the network.

A parent can only make calls to its immediate children in the process tree structure. Conversely, a process can only have a single parent in the process tree. These restrictions ensure that the control flow structure among the processes is fixed and therefore can be statically analyzed. The sufficient constraint is that the process call structure is not self-

recursive.

The network's process tree is declared in two parts. The first kind of declaration indicates which processes are at the top-level of the network. There may be multiple processes at the top level. This merely indicates that they are all executing synchronously and in parallel at all times. The second kind of declaration in the process tree declares the internal branches of the tree. The leaves of the tree are those processes that have only parents in the process tree. The two kinds of process tree declarations are:

```
top:          P(NAME-1), P(NAME-2), ... P(NAME-N)  
P(NAME-I): P(NAME-I-1), P(NAME-I-2), ... P(NAME-I-K)
```

An example network including its process tree declarations is given in Figure 7-3.

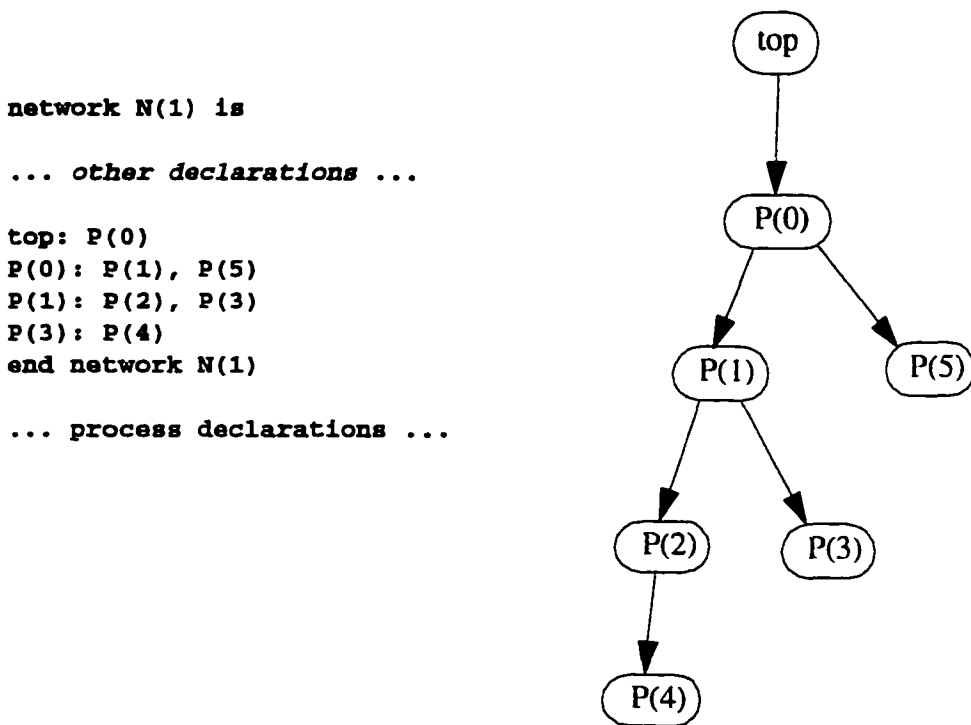


Figure 7-3. A Network and its Process Tree

7.1.2.3 A Process

The network and its process tree define the overall structural hierarchy of the system.

the instructions within each process define its behavior. At the boundary between these two is the interface to the process. The interfaces are the set of entry points from which the process may be called by its (unique) parent. The provision for this extra piece of information is critical for dataflow and control flow analysis used in the causality checking algorithm presented in Section 7.2 because the behavior of a process as seen by its parent on its first invocation can be summarized at its interfaces.

An interface declaration is:

```
interface I(start) L(0)
```

This declares an interface named '**start**' at label **0** in the instruction body of the process. There may be multiple interfaces in a process. Although a process may have multiple entry points, the structure of the process tree ensures that there is only one thread of control within the process.

A process therefore consists of declarations and instructions in the following form:

```
process P(NAME) is  
{ type-declaration  
  signal-declaration  
  register-declaration  
  exception-declaration  
  counter-declaration  
  interface-declaration  
  instruction }  
end process P(NAME)
```

The declarations within a process are visible both within the process, and to its child processes. The forms of the various declarations is similar to that of the signal in the network, except that a process can declare types, signals, registers, exceptions and counters:

```
type T(NAME) SIZE  
signal S(NAME) T(NAME)  
register R(NAME) T(NAME)  
temporary R(NAME) T(NAME)  
exception E(NAME) T(NAME)  
counter C(NAME) VALUE
```


The utility of these declarations is made clear in the following section which covers the behavioral constructors. In short: a *type* is used to mark all value carriers with the domain to which they belong; *signals* are the special sort of single-assignment register which is used to coordinate inter-process communication and ensure the modularity of the semantics; *registers* constitute the aspect of the state space of the process and are used to hold values both within δ -time and between macrosteps; a *temporary* is a special sort of register which is used only within δ -time; an *exception* is a special sort of register which is used to communicate the value thrown in an exception back to the parent's catcher; and a *counter* is a special sort of decrement-only register which is used to model the counted-signals aspect of Esterel.

7.1.3 Behavioral Constructors

The behavioral constructors are declarations and instructions which define the behavior of a network both in δ -time and by extension in at the macro level. This section defines the interesting and unique instructions of the NDAM and gives their informal semantics. A formal semantics which precisely specifies the operational semantics of the NDAM is deferred until Section 7.3.

The instruction body consists of NDAM instructions prefixed by labels of the form $\mathbf{L}(\mathbf{NAME})$. Presentation of instruction body examples is deferred to Section 7.1.4 at which point the various instructions will have been presented.

7.1.3.1 Signals

Signals are the only means of inter-process communication. They provide a means by which values are transferred between processes. Additionally they provide a synchronization mechanism by virtue of the fact that a signal cannot be referenced (in a reader) until its value is defined (by a driver). A signal may only be assigned once in a macrostep.

There are, in effect, by this distinction of reader and driver, two classes of signal-manipulation instructions: those that drive signals and those which operate based on the presence/absence or value on a signal. The **emit** instruction is the unique instruction used to assign a value to a signal:

emit S(LHS) R(RHS)

The **emit** assigns the value of the right-hand side onto signal **S(LHS)** in the instant. It is effectively a special class of assignment. It is unique in that it defines the presence of the signal **S(LHS)** henceforth in the instant.

The second class of signal-manipulation instructions are those that deal with the consumer side of the equation. There are two instructions that access the value and presence or absence of a signal. These are:

R(LHS) := presence S(RHS)

R(LHS) := selection S(RHS)

The **presence** instruction returns a single-bit result indicating whether the signal **S(RHS)** is present or absent in the current instant.

The **'selection'** instruction returns the value on the signal in the current instant. Some further comments on this terminology and its relevance to the particular form of selection nondeterminism used in the NDAM are deferred until Section 7.1.3.4. There is a particular form of the conditional that transfers control based on the presence or absence of a particular signal within an instant:

present [not] S(NAME) goto L(NAME)

This instruction branches to **L(NAME)** if the signal **S(NAME)** is present in the current instant. Otherwise it continues on to the successor instruction.

Of note here is that the **present** instruction could be just as easily derived from the following sequence of instructions:

```
type t(bit) 2
temporary R(tmp0) t(bit)
  R(tmp0) := presence S(NAME)
  if R(tmp0) goto L(NAME)
```

Conversely, the **'presence'** accessor instruction could be derived from the **present** branch test and a temporary. Both exist in the intermediate representation for conceptual efficiency in code generation from the high-level language.

Signals are the unique synchronizing vehicle within δ -time. They are required to present a single value and presence/absence status to all readers in the macrostep. Thus they must obey a restriction to single assignment across δ -time and no reader must access a signal in a δ before it has been written. To ensure that a signal is not read before it is written across δ -time, a barrier instruction **'require'** suspends the containing thread of execution until the mentioned signals have become known δ -time:

require S(NAME-1), S(NAME-2), ... S(NAME-K)

The **'require'** instruction is a barrier instruction that can be used to block a process within the instant until the presence/absence, and therefore the value, of the signals **S(NAME-i)** is defined.

This blocking occurs only *within* the macrostep. When the status of each **S(NAME-i)** is known then a **'require'** does nothing. A **'require'** must precede every signal accessor instruction (e.g. **'select'** or **'presence'**) on a control flow path. Its use is illustrated in Figure 7-4.

There is one further class of instruction that interacts with signals. It supports the notion of interrupts wherein the emission of occurrence a signal causes a blocked computation to proceed. Interrupts are handled by the try-call-watching-catching (**TCWC**) instruction which also manages exceptions and the concurrent execution of subprocesses. Its presentation is deferred until Section 7.1.3.3.

7.1.3.2 Exceptions

Signals are the one means by which processes can pass values and coordinate their δ -step execution. Exceptions on the other hand allow for one process to preempt all of its

```

type t(int) 65356
type t(pure) 1
signal S(1) t(int)
signal S(2) t(int)
signal S(3) t(pure)
signal S(OUT) t(int)

temporary R(t1) t(int)
temporary R(t2) t(int)
L(0):  require S(1), S(2)
        R(t1) := selection S(1)
        R(t2) := selection S(2)
        require S(3)
        present S(3) goto L(2)
        R(t3) := R(t1) + R(t2)
        goto L(2)
        R(t3) := R(t1) - R(t2)
L(2):  emit S(OUT) R(t3)
        exit -- we're done

```

Figure 7-4. The Use of 'require' to Ensure Definition Before Use

siblings (and their children) and dictate a return into the parent where the exception is 'handled.' This is all accomplished within a macrostep in a manner which is entirely consistent with synchronous semantics.

There are three exception handling instructions. The two that raise an exception and reference its thrown value are presented in this section. The presentation of the third *TCWC* is deferred until the next section, Section 7.1.3.3.

raise E(LHS) R(RHS)

This instruction raises the exception E(LHS) passing R(RHS) back to the exception handler in the parent. The execution of the 'raise' preempts all of the processes below the parent which is handling the exception in a *TCWC*. Control is returned into the parent's exception handler.

R(LHS) := raised E(RHS)

The '**raised**' instruction recovers the thrown exception value from the virtual register **E(RHS)** into a register in the parent. This instruction is only valid in the exception handler when there is a meaningful quantity in **E(RHS)**.

The instantiation of exception handlers and the invocation of the handler's instructions after a '**raise**' has occurred is dealt with in the **TCWC** instruction.

7.1.3.3 Try-Call-Watching-Catching (TCWC)

The instruction that is the most unique to the NDAM is also its most complicated. It is the "Try Call Watching Catching" (**TCWC**) instruction which manages three features:

1. the commencement and continued concurrent execution of subprocesses.
2. interrupts or guarding against a signal's presence in the instant.
3. exception handling and recovery from raised exceptions in the instant.

The fully general form of the instruction is shown in Figure 7-5. Either or both of the '**watching**' and '**catching**' subclauses may be absent. This indicates that no signals are guarded against and that no exception handlers are instantiated respectively.

```
try
  call P(CHILD-1) I(NAME);
  call P(CHILD-2) I(NAME);
  ...
  call P(CHILD-K) I(NAME);
watching
  when [ C(NAME-1) ] S(NAME-1) goto L(NAME-1);
  when [ C(NAME-2) ] S(NAME-2) goto L(NAME-2);
  ...
  when [ C(NAME-L) ] S(NAME-L) goto L(NAME-L);
catching
  handle E(NAME-1) goto L(NAME-1);
  handle E(NAME-2) goto L(NAME-2);
  ...
  handle E(NAME-M) goto L(NAME-M)
```

Figure 7-5. The Fully General Form of the **TCWC** Instruction

When the **TCWC** is executed, it starts its child process **P (CHILD-i)** starting each at its entry point **I (NAME)**. The children execute in synchronously and concurrently. There are two cases for the **TCWC**: the case when the **TCWC** is first executed in a macrostep and the case when it continues to be active in a successor macrostep.

There are three possibilities for **TCWC** during the first instant that it is executed:

1. If all the **P (CHILD-i)** processes exit in the current instant then the **TCWC** proceeds on to its successor instruction.
2. If any child raises an exception handled by this **TCWC** then all the children are terminated at the end of their δ -steps. Execution continues on at the label **L (NAME)** of the appropriate **'handle'** clause. This gives the "last wishes" flavor of exceptions. If any child raises an exception that is not handled by this **TCWC** then the calling **TCWC** with such a handler takes care of the exception. There will always be such a parent in a correct network.
3. If no child executes a **'raise'** but any child halts then the **TCWC** blocks for the macrostep.

In subsequent macrosteps the **TCWC** checks the signal guards in the **'when'** clauses and executes the child processes only if no signal counter has expired. Thus, the child processes of the **TCWC** execute only after the status of the guarding signals becomes known. There are four elements to execution of the **TCWC** in the second and subsequent instants of its invocation:

0. All signal guards are checked. If any signal guard's counter has expired (reached zero) then all the **P (CHILD-i)** are terminated and execution proceeds at the label **L (NAME)** of the appropriate **'when'** clause signal handler.

Cases 1, 2, and 3 are all same as for the first instant of invocation.

7.1.3.4 Selection Nondeterminism

The NDAM supports selection nondeterminism within the $RM\bar{C}$ synchronous semantics because it does not violate the modularity requirement. Nondeterminism of this sort is used as an abstraction mechanism in formal verification as was described in Section 5.7.1.5. On the NDAM, nondeterminism comes into play when emitting multiple values on a signal in a single δ -step or branching to more than one successor instruction. To this

end there is a nondeterministic variant of the 'emit' instruction. As well, the 'goto' branch instruction has a nondeterministic variant that has multiple branch targets. The key distinction in these instructions is that they do not destroy the *M* property; they do not introduce ordering nondeterminism.

The nondeterministic versions of these two instructions are:

emit *S(LHS)* *R(RHS-1)*, *R(RHS-2)*, ... *R(RHS-K)*

The 'emit' assigns one of the values of the *R(RHS-i)* onto the signal *S(LHS)* in the instant.

goto *L(NAME-1)*, *L(NAME-2)*, ... *L(NAME-N)*

The program counter of the process is assigned to one of the *L(NAME-i)*.

These are the only two nondeterministic instructions on the NDAM. With these nondeterministic forms, the signal accessor instructions are said to *resolve* the nondeterminism.

7.1.3.5 Other Instructions

The other instructions are as might be expected on a infinite-register reduced instruction set architecture. These include assignment operations, addition, subtraction, branch, test-and-branch and the like. The "traditional" instructions are as follows:

```
R(LHS) := R(RHS)  
R(LHS) := unary-op R(RHS)  
R(LHS) := R(RHS-1) binary-op R(RHS-2)  
if [not] R(NAME) goto L(NAME)  
goto L(NAME)  
null
```

These instructions are given in detail in Appendix A.

7.1.4 Examples

An example that illustrates the basic instructions, is shown in Figure 7-6. That figure shows a fragment of C code and the corresponding NDAM assembly code.¹

```

/* Euclid's algorithm */
{
  short a, b;          /* filled somehow*/

  /* presumably R(a) and R(b) are
     filled with values somehow*/

  while (b != 0) {
    int a0 = a;
    a = b;
    b = a0 % b;
  }

  /*a contains the result*/
}

type t(short) 65535  -- 2^16
register R(a) t(short)
register R(b) t(short)
constant R(0) t(short) := 0

type t(bool) 2
-- presumably R(a) and R(b)
-- are filled with values somehow
temporary R(tmp1) t(bool)
L(0): r(tmp1) := R(b) != R(0)
      if r(tmp1) goto L(1)
temporary R(a0) t(short)
      R(a0) := r(a)
      R(a) := R(b)
      R(b) := R(a0) mod R(b)
      goto L(0)
L(1): exit          -- result in R(a)

```

Figure 7-6. Euclid's Algorithm in C and in NDAM Assembly

A more detailed example that shows a wider range of instructions, including the network, several subprocesses and concurrent **TCWC** is shown in Figure 7-7a and Figure 7-7b respectively. That example is the classic Esterel stopwatch example.¹ The process tree for the network is shown in Figure 7-7a along with the original Esterel source. The presentation of the compilation recipe used to create that example is deferred until Section 7.4.1.

7.2 Causality Checking

Synchronous semantics is, by definition, $RM\bar{C}$. Thus for implementations which will execute in the real world, there is an obligation to ensure that the systems described in the semantics are causal: there must be a well-defined way to execute them forward in time. The goal of causality checking is the determination of whether a NDAM network has the correct state-dependent partial order \leq_Q (i, o) for every reachable state Q .

A causality checking procedure in the barest sense is decision procedure that returns *TRUE* or *FALSE* as to whether the network has the appropriate \leq_Q (i, o). Any implemen-

1. From Cormen *et al.* [206], page 810.

1. Adapted from Halbwegs [320], page 23 and 26.


```

module STOPWATCH_1:

input START_STOP, HS, RESET;
output TIME(integer);

loop
  var TIME := 0: integer in
    loop
      emit TIME(TIME);
      await START_STOP;
      do
        every HS do
          TIME := TIME+1;
          emit TIME(TIME)
        end every
      upto START_STOP
    end loop
  end var
each RESET
end module

```

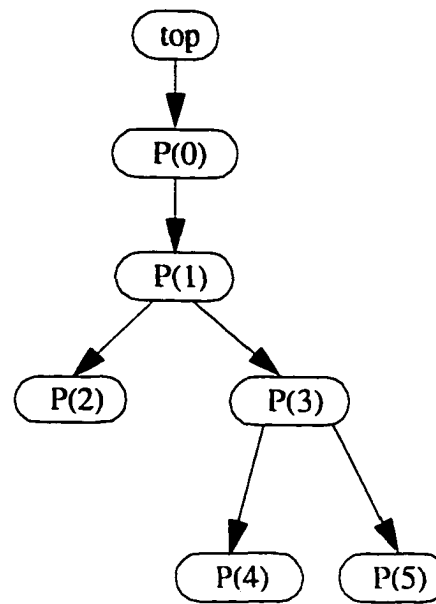


Figure 7-7a. The Classic Stopwatch in Esterel

tation of such a causality checking procedure should have other desirable properties as well. These properties include:

1. If the answer is *FALSE*, that the network is \bar{C} , then some sort of diagnostic trace should be produced that indicates *why* this is the case. Specifically, for which reachable state Q the causality failure occurred.
2. The decision procedure should be 'quick' in the sense that it can usefully be used in interactive or semi-interactive contexts the same way that a traditional compiler is used.

On the positive side, there are known approaches for designing diagnostic causality checking algorithms. This satisfies the first criterion. Unfortunately, as is pointed out in Section 7.2.2, the general causality checking problem is intrinsically difficult: having been shown to be NP-complete. Thus as the size of a description grows, exact causality checking must be dropped in favor of conservatively estimating the state-dependent causal order within a step. Such an exact and estimating scheme is defined here in Section 7.2.3 using a flattened representation of the hierarchical NDAM process tree.

```

network N(1) is
top: P(0)
P(0): P(1)
P(1): P(2), P(3)
P(3): P(4), P(5)
type t(unit) 1
type t(bool) 2
type t(int) 0 65536
signal s(0) t(unit)-- tick
signal s(1) t(unit)-- START_STOP
signal s(2) t(unit)-- HS
signal s(3) t(unit)-- RESET
signal s(4) t(int)-- TIME
input: s(0), s(1), s(2), s(3)
output: s(4)
end network N(1)

process P(0) is
constant R(unit) t(unit) := 0
constant R(false) t(bool) := 0
constant R(true) t(bool) := 1
constant r(int0) t(int) := 0
constant r(int1) t(int) := 1
temporary R(tmp0) t(int)
temporary R(tmp1) t(int)
register r(0) t(int) := 0-- TIME
L(0): try
    call P(1) I(0);
    watching
    when s(3) goto L(1)
L(1): goto L(0)
end process P(0)

process P(1) is
interface I(0) L(0)
L(0): r(0) := r(int0)
L(1): emit s(4) r(0)

L(2): try
    call P(2) I(0);
    watching
    when s(1) goto L(3)
L(3): try
    call P(3) I(0);
    watching
    when s(1) goto L(4)
L(4): goto L(1)
end process P(1)

process P(2) is
interface I(0) L(0)
L(0): halt
end process P(2)

process P(3) is
interface I(0) L(0)
L(0): try
    call P(4) I(0);
    watching
    when s(2) goto L(1)
L(1): try
    call P(5) I(0);
    watching
    when s(2) goto L(2)
L(2): goto L(1)
end process P(3)

process P(4) is
interface I(0) L(0)
L(0): halt
end process P(4)

process P(5) is
interface I(0) L(0)
L(0): r(tmp0) := r(0) + r(int1)
L(1): r(0) := r(tmp0)
L(2): emit s(4) r(0)
L(3): halt
end process P(5)

```

Figure 7-7b. The NDAM Network for the Stopwatch

7.2.1 Problem Definition

The definition of causality used in Section 4.1.3 was the existence of a state-dependent partial order relation $\leq_Q (i, o)$ which respects composition. In that context, the existence

of $\leq_Q (i, o)$ had the specific meaning that there existed pairs $I(i)$ and $O(o)$ which were causally related in the sense of the inputs must be available before the outputs could be computed. This relationship could depend on the state $Q(c)$ whence it was observed. The key element of causality was that the partial order respected composition; that for a state defined as two components $Q_1 \times Q_2$ that $\leq_{Q_1 \times Q_2} (i, o)$ respected the causal orders of both communicating components $\leq_{Q_1} (i, o)$ and $\leq_{Q_2} (i, o)$.

In the study of the microsemantics in Chapter 4 that definition of causality was seen, in practice, to be a property of the δ -path emanating from a particular state Q . Causality is a requirement that an output, be written in some δ -step before it is read. In the terminology used in the NDAM, this is stated as the condition that a **signal** must be defined by an **'emit'** before it is accessed (**'select,' 'presence'** or **'present'** test) and this must be verified along every δ -path emanating from every reachable state. This more intuitive view is depicted in Figure 7-8 with the intuitive view of a causality problem being illustrated in Figure 7-8.

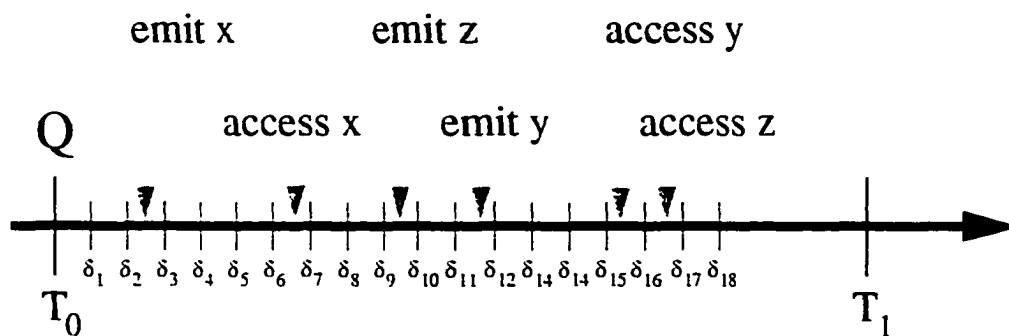


Figure 7-8. Causal Consistency on a δ -Path

There are two features which make this problem non-trivial in the general case. The first is that the causal ordering property must be verified for every δ -path out of every reachable state. This means that the size of the general causality checking problem grows with the size of the state space of the system. The second is that the generation of the δ -paths is

complicated by the distribution of the emission and access across the concurrently executing processes. This means that the reachable state space must be traversed in order to generate the δ -path.

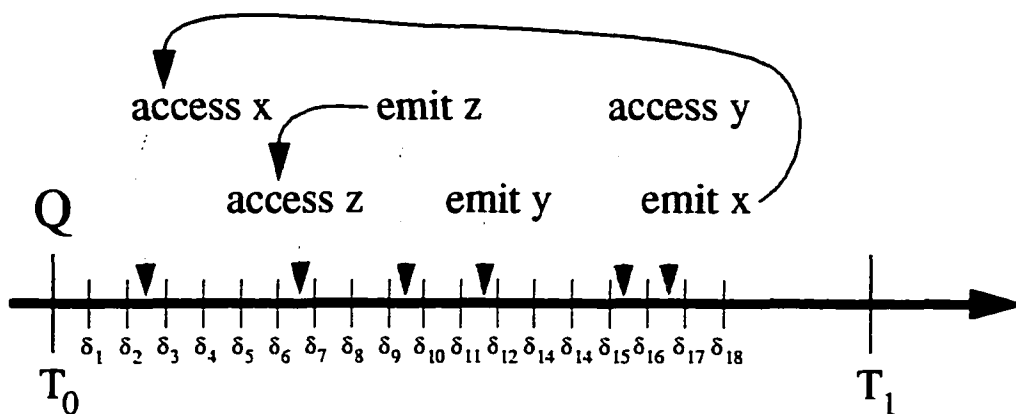


Figure 7-9. The Manifestation of Causality Problems on a δ -Path

The following sections describe known approaches to the general causality checking problem in a general setting (*i.e.* independent of the NDAM). Following that is the formulation of the causality checking problem for the specific case of the NDAM.

7.2.2 Known Approaches

There are two approaches to the causality checking problem. The first uses an explicit enumeration of the reachable state space and the δ -paths between the states. The second uses an implicit or symbolic representation of these items.

7.2.2.1 Explicit Enumeration

Conceptually the explicit enumeration technique is straightforward and is sketched in Figure 7-10. In practice, this procedure is extremely expensive because it visits every state and every edge of the state transition graph of the program. The set V ultimately grows to hold the reachable state set whose size potentially grows as the product of the number of concurrent processes. Additionally the inner loop explores every feasible transition out of

every state and the number of such input combinations grows as the product of the number of inputs as well.

```

explicit_causality_check(program, initial-states)
{
  let  $I = \text{initial-states}$  // a non-empty set of states
  let  $W = I$  // the work set (states to visit)
  let  $V = \emptyset$  // the set of visited states

  while  $|W| \geq 0$  {
    choose  $s \in W$ 
     $V = V \cup \{s\}$  // note that  $s$  has been visited
    foreach  $i \in \text{INPUT}$  { // inputs from the environment
      simulate  $s \rightarrow \delta_1 \rightarrow \delta_2 \rightarrow \dots \rightarrow \delta_k \rightarrow t$ 
      ensure that  $k < \infty$  by checking for  $\exists i, j. \delta_i = \delta_j$ 
      if  $k = \infty$  report an "instantaneous loop"
      check for causality property on the  $\delta$ -path
      if  $t \notin V$ 
         $W = W \cup \{t\}$ 
    }
  }
}

```

Figure 7-10. Causality Checking by Explicit Enumeration

In practice a less accurate version of this algorithm is used. In particular, a distinction is made between “data” states which are held in variables and “control” states which correspond to the positions of the program counters in active processes. These correspond, in the NDAM context to the **register** values and the locations of ‘halt’ instruction respectively. The convenient abstraction is to ignore the contribution of the data states and the instructions that process them. This leads to a much more abstract version of the “simulate” which can estimate if a causality problem *may* occur in the δ -path emanating from a reachable control state. Another simplification is to place restrictions on the environment in which the program will be placed. The environment guarantees that only a subset of the possible input event combinations will ever occur; equivalently, the unobservable

input event combinations are an input “don’t care” set for the program. This has the effect of making the set *INPUT* much smaller than it would otherwise be. Explicit causality checking under these restrictions is feasible for many programs.¹

7.2.2.2 Implicit Enumeration²

Experience has shown that implicit or symbolic formulations of explicit algorithms are often quite attractive. The benefit lies in the observation that the size of the symbolic representation is often entirely unrelated to the size of the state space being manipulated. The same can be said of explicit causality checking. In the formulation of causality checking from an implicit or symbolic perspective, the system is viewed from the perspective of approximator functional. This is depicted in Figure 7-11 with the sets Q_i and Q_{i+1} representing the current-state and the next-state of the network respectively. In turn, these states are defined in terms of intermediate δ -states which are denoted Q_{δ_0} and Q_{δ_1} respectively. The significant point in that diagram is that there are k feedback dimensions between Q_{δ_0} and Q_{δ_1} . The ability to identify or estimate k is crucial to the following development because it is the finite point at which the fixed point is reached. Either:

$$\bar{F} = \mathcal{F}\{\bar{F}\} = \mu_{F_\delta} \mathcal{F} = \bigsqcup_{i=0}^k \mathcal{F}\{F_\delta\} \quad (\text{Eq 7-1})$$

or there is no other finite point $k < \infty$, so necessarily:

$$\bar{F} = \mathcal{F}\{\bar{F}\} = \mu_{F_\delta} \mathcal{F} = \bigsqcup_{i=0}^{\infty} \mathcal{F}\{F_\delta\} \quad (\text{Eq 7-2})$$

The distinction between Eq 7-1 and Eq 7-2 is merely the number of iterations before the fixed point. The key to the implicit causality checking decision procedure is the use of the

1. Procedures like the one shown in Figure 7-10 are used in the early Esterel compilers. These are described in Section 7.5.3 distinguishing them along the lines of the runtime implementation that they use.

2. I thank Gérard Berry for pointing out the relevance of Malik’s analysis of cyclic combinational circuits [494] to the causality checking problem. This technique has since been incorporated into the causality checker of the Esterel v4 compiler [264] [525].

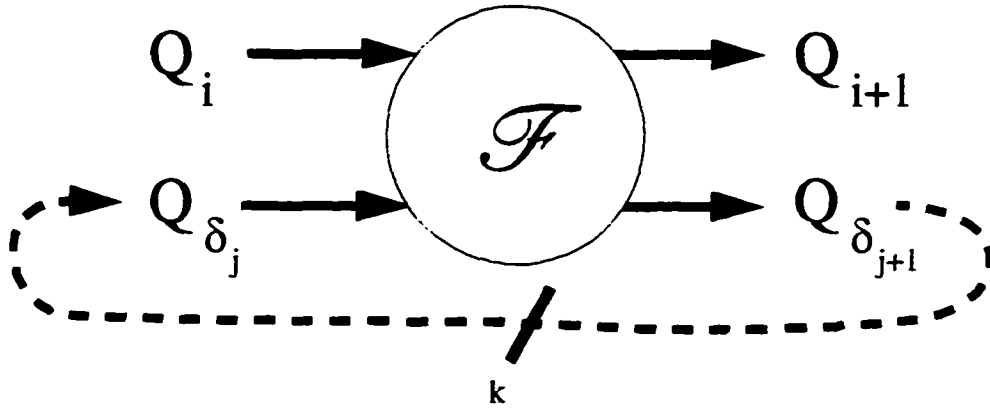


Figure 7-11. Causality from the Perspective of the Image Computation

structural metric k which is the number of dimensions of feedback, in a theorem which states that if the fixed point does not happen by a point $\Lambda(k)$ then it won't happen until infinity. Having a fixed point at infinity implies that there exists at least one δ -path which never terminates. Obviously such a path cannot be compressed to finite size so that it could be embedded in a single macrostep.

With this formulation in mind, the question of causality becomes one of whether Q_{i+1} is uniquely determined from Q_i and this is checked by checking that the following property holds:

$$\bar{F}_0 \subset \bar{F}_1 \subset \bar{F}_2 \subset \dots \subset \bar{F}_{\Lambda(k)} \quad (\text{Eq 7-3})$$

That is, it is necessary to check, for each step in $0 < i \leq \Lambda(k)$, that $\bar{F}_i \subset \bar{F}_{i+1}$; an equivalent check is that $\bar{F}_{\Lambda(k)}$ is a limit point for the domain $Q \rightarrow Q$.

As with the case of Eq 3-51, there need not be a material representation for the \bar{F}_i such that $\bar{F}_i \subset \bar{F}_{i+1}$ can be established. Conveniently, as with the explicit case of Section 7.2.2.1, the property of Eq 7-3 need not hold for all states Q_i . Let Q^* denote the reachable states of the system; Eq 7-3 need only hold for the $Q_i \subseteq Q^*$. Intuitively, the network can behave *arbitrarily* when executed from an unreachable state since no such state can ever be observed. Thus, Eq 7-3 can be specialized to Q^* , iterating until Q_δ stabilizes or

the limit of $i = \Lambda(k)$ is reached:

$$\bar{F}_0\{Q^*, Q_{\delta_0}\} \subseteq \bar{F}_1\{Q^*, Q_{\delta_1}\} \subseteq \bar{F}_2\{Q^*, Q_{\delta_2}\} \subseteq \dots \subseteq \bar{F}_{\Lambda(k)}\{Q^*, Q_{\delta_{\Lambda(k)}}\} \quad (\text{Eq 7-4})$$

That is, it is necessary to check, for each step in $0 < i \leq \Lambda(k)$, that $Q_{\delta_i} \subseteq Q_{\delta_{i+1}}$; an equivalent check is that $Q_{\delta_{\Lambda(k)}}$ is a limit point for the domain S_{δ} .

The following theorem guarantees that if $Q_{\delta_{\Lambda(k)}}$ is not a limit point then the fixed point must be at infinity. The theorem follows Sharad Malik's original formulation for the flat Boolean domain shown in Figure 3-3. The new aspect here is the generalization of that result to arbitrary finite domains (*i.e.* to non-flat domains). The iteration limit $\Lambda(k)$ is generalized and shown here to be the sum, over the k feedback paths, of the lengths Γ_i of the longest increasing chain of the feedback domain:

$$\Lambda(k) = \sum_{i=1}^k \Gamma_i \quad (\text{Eq 7-5})$$

Where the domains on every feedback paths is the same this reduces to the product of the number of feedback paths k and the length Γ of longest increasing chain in the domain:

$$\Lambda(k) = k \times \Gamma \quad (\text{Eq 7-6})$$

Of course, the original bound is a special case of this since the longest increasing path in the Boolean domain is $\Gamma = 1$: thus $\Lambda(k) = k$.

The Fixed Point Bound Theorem

Let the k internal variables of a non-abstract image semantics be identified. Let $\Lambda(k)$ be as defined in Eq 7-5. Then, either the fixed point, $\mu \mathcal{F}$, of the semantics' approximator functional, \mathcal{F} , occurs in $\Lambda(k)$ steps as per Eq 7-1 or there is no finite limit point for the semantics; and therefore the limit point is infinite as per Eq 7-2.

Proof Outline

The k internal variables of any non-abstract image semantics can be identified by examining the dimensions of the semantics that are projected away by the projection Π to full abstraction. These variables can be identified by an examination of structure of the microsemantics.

When $k = 0$ there is no internal feedback in a macrostep, the semantics is fully abstract and $Q_{i+1} = F\{Q_i\}$ is existential. Therefore the proof establishes the case of $k = 1$ and then generalizes this result to arbitrary k .

Proof

Observe from Figure 7-11 that a state in the non-abstract semantics consists of two components: the external component Q_i and the internal component Q_{δ_j} , which when associated as a tuple are written (Q_i, Q_{δ_j}) . Also observe that, on the iterations before the fixed point when $j < \Lambda(k)$, that Q_{i+1} may not be complete; the values appearing there are only approximations of the values which will finally appear there. To denote the incompleteness that precedes the fully complete Q_{i+1} , the notation ${}_{\delta_j}\bar{Q}_{i+1}$ is used. This quantity has the following two properties:

1. $\forall j, {}_{\delta_j}\bar{Q}_{i+1} \sqsubseteq Q_{i+1}$
2. $\forall j \geq \Lambda(k), {}_{\delta_j}\bar{Q}_{i+1} = Q_{i+1}$

Let F_{δ} be the primitive image functional for the non-abstract semantics as per Section 3.4. In application form and using the tuple notation defined above this gives:

$$\left({}_{\delta_{j+1}}\bar{Q}_{i+1}, Q_{\delta_{j+1}} \right) = F_{\delta} \left\{ \left(Q_i, Q_{\delta_j} \right) \right\} \quad (\text{Eq 7-7})$$

For the case of $k = 1$ in Figure 7-11, four sorts of dependencies can be distinguished in the microsteps of the semantics:

1. $Q_i \rightarrow_{\delta_{j+1}} \bar{Q}_{i+1}$
2. $Q_i \rightarrow Q_{\delta_{j+1}}$
3. $Q_{\delta_j} \rightarrow_{\delta_{j+1}} \bar{Q}_{i+1}$
4. $Q_{\delta_j} \rightarrow_{\delta_{j+1}} \bar{Q}_{i+1}$

A dependency is defined relative to the monotonicity of Eq 7-7 for a state $Q = (Q_i, Q_{\delta_j})$. A dependency is an implication that an increase, relative to \sqsubseteq , in the accuracy of the left-hand side implies a potential increase in the accuracy of the right-hand side. The fixed point occurs when there is no increase across Eq 7-7 and instead there is equality. Causality checking centers around exploiting the dependency structure in the verification that F_δ is in fact monotonic over Q .

Two cases can be identified in Eq 7-7:

1. $(\delta_{j+1} \bar{Q}_{i+1}, Q_{\delta_{j+1}}) \sqsubseteq F_\delta \{(Q_i, Q_{\delta_j})\}$
2. $(\delta_{j+1} \bar{Q}_{i+1}, Q_{\delta_{j+1}}) = F_\delta \{(Q_i, Q_{\delta_j})\}$

Thus either F_δ increases or it does not. When it does not, the fixed point has been reached. Examining the two components, Q_{δ_j} and $\delta_j \bar{Q}_{i+1}$ in light of the dependencies establishes iteration bound $\Lambda(k)$ until the fixed point. Consider:

1. $Q_i \rightarrow_{\delta_{j+1}} \bar{Q}_{i+1}$
 Q_i does not increase with j and hence cannot affect $\Lambda(k)$.
2. $Q_i \rightarrow Q_{\delta_{j+1}}$
 Q_i does not increase with j and hence cannot affect $\Lambda(k)$.
3. $Q_{\delta_j} \rightarrow_{\delta_{j+1}} \bar{Q}_{i+1}$
 Q_{δ_j} can increase with j ; the fixed point occurs one step after when it does not.
4. $Q_{\delta_j} \rightarrow_{\delta_{j+1}} \bar{Q}_{i+1}$
 $\delta_{j+1} \bar{Q}_{i+1}$ can increase with j but is solely dependent on doing so by an increase via case 3 in Q_{δ_j} from iteration $j-1$.

This case analysis shows that it is only *case 3* that directly affects the number of iterations of the approximator functional \mathcal{F} until the fixed point. Thus, any bound on the number of increases that *case 3* can cause necessarily implies a bound on the number of increases that can occur in F_δ . In turn, this gives the a bound on the number of iterations of \mathcal{F} until its fixed point.

For the case of $k = 1$, this iteration bound is the length of the longest increasing chain from \perp in the domain of Q_δ . Let the length of this longest increasing chain be given by Γ for the domain of Q_δ . The fixed point of \mathcal{F}_δ cannot be any longer than Γ because (a) F_δ is monotonic so there can be no decrease through *case 3* and (b) when equality occurs through *case 3*, no further increase is ever possible on any further iteration. At that point, $Q_{i+1} = \mu_{Q_\delta} F_\delta$ and so $\bar{F}_{Q_i} = \mathcal{F}_\delta\{\bar{F}_{Q_i}\}$ by Eq 3-51. Thus for the case of $k = 1$:

$$\Lambda(k) = \Gamma$$

Moving to the case of $k > 1$ follows the same line of reasoning: either no dimension of it increases or some subset of the dimensions of Q_δ increase. When no dimension increases, by the above analysis, the fixed point has been reached and no further increases can occur on any further iteration. When a dimension does increase, then it moves up one notch on its increasing chain. Let the length of the longest increasing chain of each domain D_i making up Q_δ be given by Γ_i . There are k such domains comprising the k feedback dimensions of \mathcal{F}_δ . In the increasing case, at least one feedback dimension D_i increases, and its increase is at least one notch relative to \subseteq_{D_i} . There can be at most Γ_i such increases. The same can be said for every other feedback dimension as well. The worst case being when every feedback dimension increases independently on a separate iteration. Therefore $\Lambda(k)$ is given by:

$$\Lambda(k) = \sum_{i=1}^k \Gamma_i \tag{Eq 7-8}$$

Where the domains D_i are all the same, the Γ_i are all equal and can be given as the unsubscripted Γ . In that case:

$$\Lambda(k) = k \times \Gamma$$

In the case where the domains are all the flat Boolean domain, the longest increasing chain is 1 so:

$$\Lambda(k) = k$$

Q.E.D.

With the aid of this theorem, implicit causality checking is straightforward. The algorithm is shown in. Fully general causality analysis by this method is NP-complete [494] as might be expected from the formulation in terms of image computations on the reachable state space of the program instance.

7.2.3 The Flattened Transition-Graph

The approach taken to causality checking of NDAM programs is not distinct from the approaches previously presented but rather draws from either the explicit or implicit enumeration technique as appropriate. The central problem in formulating the causality checking problem for NDAM networks is that the control flow paths in a step are obscured not only by concurrency but also by the hierarchy of the process tree. The construction of a flattened transition graph from the process tree addresses this problem and allows either of the approaches to causality checking be used directly.

The process tree is convenient and natural for the representation and execution of hierarchical control constructs. Indeed, the whole purpose of the multifunctional **TCWC** instruction is to isolate behavioral hierarchy in such a way that the operational interpretation of NDAM instructions is more or less direct.¹ The problem with the process tree for causality analysis is that this very hierarchy obscures the control flow paths between lev-

```

implicit_causality_check(program, initial-states)
{
  let  $I = \text{initial-states}$  // a non-empty set of states
  either
    let  $Q^* = \mu_I(\lambda Q.Q \vee \bar{F}\{Q\})$  // compute reachable states
  or // alternatively
    let  $Q^* = \text{est}\{\text{program}\}$  // estimate  $Q^*$  structurally
  // determine the  $k$  feedback dimensions from a structural
  // analysis of the concurrent control flow in the program
  let  $k = \text{dfs}(\text{program})$ 
  let  $\Gamma_i$  be determined for each  $1 < i \leq k$  feedback dimension
  let  $Q = F_\delta^{(\wedge(k))}\{Q^*\}$  // simulate for  $\wedge(k)$   $\delta$ -steps
  if  $Q$  is a limit point then
     $Q = F_\delta\{Q\}$  and  $\bar{F}_{Q^*} = \mathcal{F}_\delta\{\bar{F}_{Q^*}\}$ 
    the description is causal
    return OK
  else // some elements in  $Q$  are not limit points
    these correspond to noncausal loops
    show a  $\delta$ -path from  $q \in Q$  back to a previous macrostate
    return NOT OK
}

```

Figure 7-12. Causality Checking by Implicit Enumeration

els of the hierarchy. Since causality is intrinsically a δ -path property, not a δ -state property, a representation that makes such δ -paths explicit is a fundamental component of causality checking. The following sections define a flat transition-graph representation which is derived from the hierarchical process tree.

A flat transition-graph approach is particularly appealing because it naturally takes into account the obvious intra-process control flows engendered by the **TCWC** instruction: concurrent execution as well as the behavior of signal and exception handlers. The other ben-

1. The operational semantics of the NDAM is described in Section 7.3.1 and an actual implementation which is faithful to that semantics is described in Section 7.5.

efits of the graph-based formalism are that it allows for a δ -path error trace to be generated when an error is detected. It naturally supports the two other sorts of analyses that are necessary in a practical implementation. The first of these is the estimation of whether every individual control-thread is acyclic. This is a stronger condition than the necessary one which is that every δ -path be acyclic. This latter property is required if the domain S_δ is to preserve monotonicity, a point which is dealt with further in the domain definitions of the operational semantics in Section 7.3.1. The second sort of analysis is modularity checking as might be used in Synchronous VHDL.

7.2.3.1 The Estimate Graph

The fundamental property represented in the estimate graph is that each edge in the graph represents a micro-transition of the semantics. Thus the edges of the estimate graph have a one-to-one correspondence to elements of T_δ . The estimate graph flattens away the hierarchical control of the **TCWC** and replaces it with explicit edges in the flat graph representation. As such the **TCWC** represents more of a 'macro' representation for a whole set of transition edges out of the processes that it controls.

The Halting and Halted Estimates

Latent control flow occurs with the **halt** instruction in a child process. On the abstract machine, the **halt** instruction is defined to cease execution in the process containing it. Unstated are three other behaviors that must be explicitly accounted for in a flat representation. The first is that upon halting, if another sibling process executed a **raise** instruction then the halted process is killed and execution of the network continues with the relevant exception handler in that parent. The second is that in subsequent instants, if a signal occurs which is watched for in an enclosing process, then the halted process is killed and execution continues in the relevant signal handler of that parent. Thirdly, if in

any subsequent instants a sibling process raises an exception then the thread is killed and execution continues in the handler of the parent.

The estimate graph takes this into account by expressing the **halt** instruction in split-phase form. The halt is translated into of two aspects: **halting** and **halted**. The **halting** node models the arrival of control thread onto the **halt** instruction. The **halted** models the control thread remaining on the **halt** instruction after the first instant. Ignoring exceptions, the only possible edge out of a **halted** is one triggered by a watched-for signal. Figure 7-13 shows the basic **halting** and **halted** pair with edges leading away from the **halted** towards the **when** nodes header nodes of the signal handlers.

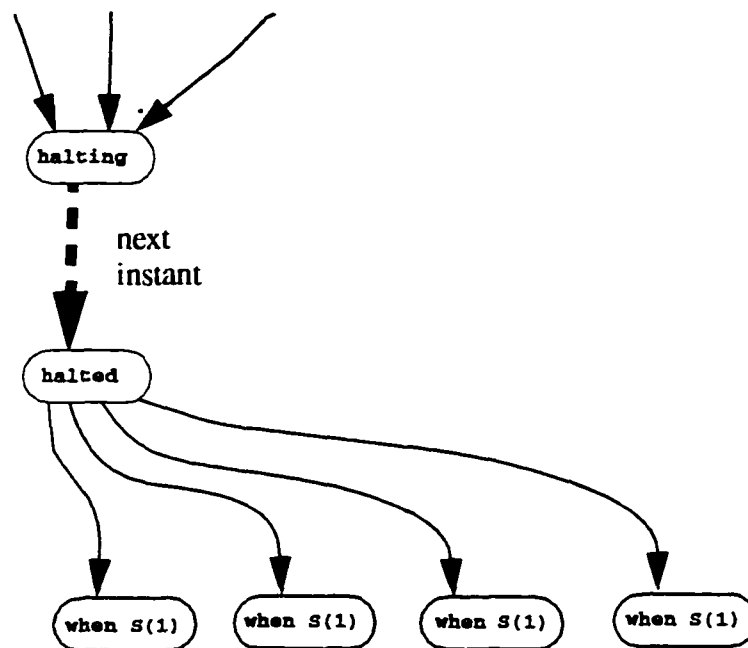


Figure 7-13. The Split-Phase Structure of **halting** / **halted**

Raised Set Clauses

The estimate nodes such as **halting**, **halted** and others that must take into account raised exceptions declare control flow to exception handlers with a set of *raised set* clauses. These clauses declare the control transfer that occurs if a sibling process executes

a **raise** in the instant. Each **raised** clause implicitly preserves the exception priority by declaring which exception it handles and which it does *not* handle.¹ Figure 7-14 shows the full **halting** / **halted** pair with the raised set clauses.

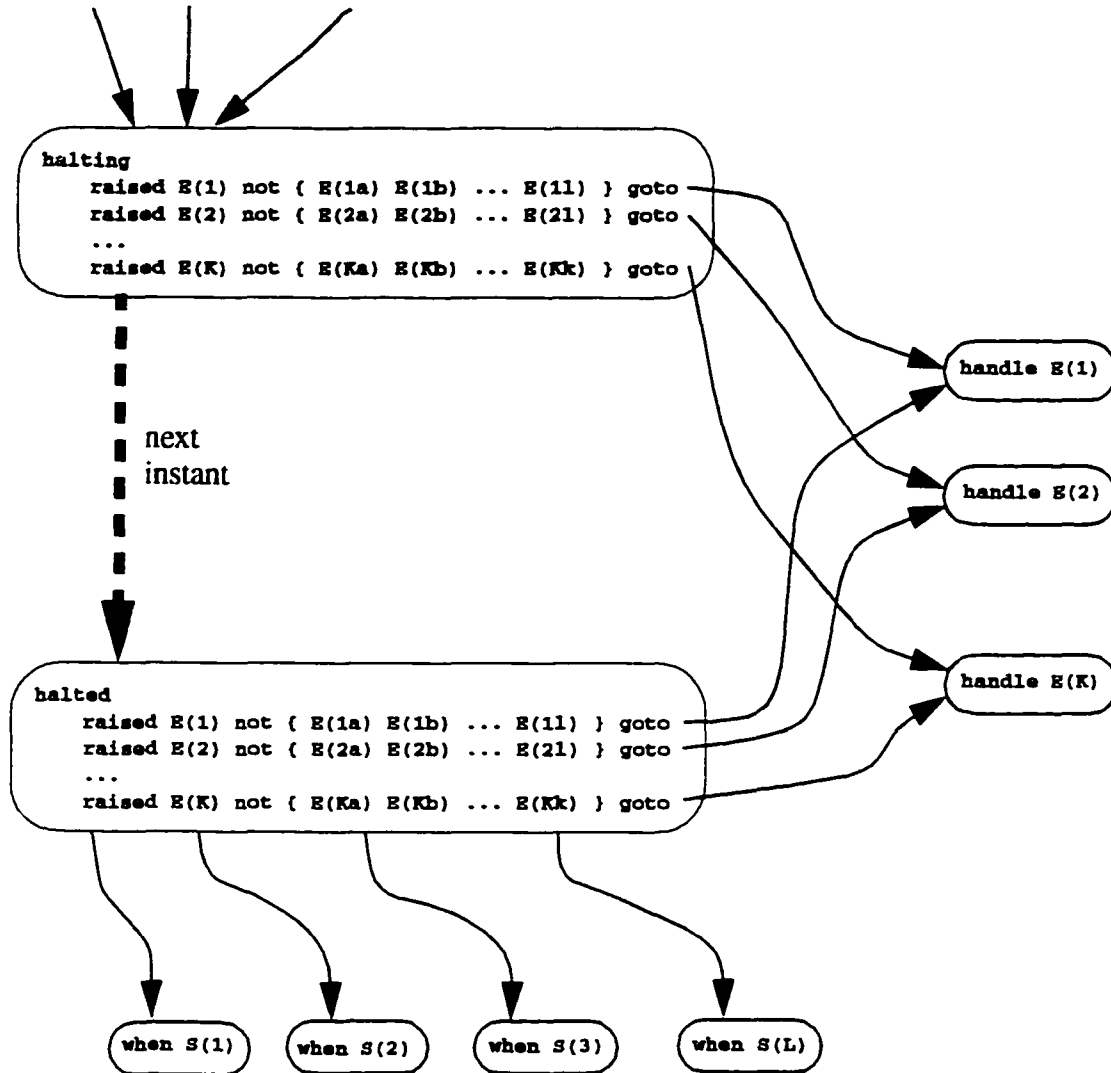


Figure 7-14. The Fully-Loaded Structure of a **halting** / **halted**

1. The flatten algorithm of Section 7.2.3.2 ensures that the *raise* clauses are consistently defined according to the priority of the enclosing **TCWC** instructions

Raise Estimate

The **raise** estimate corresponds to the NDAM **raise** instruction. It follows the form of the **halting** except that it also adds the information that the exception **E(i)** was actually raised. The raised set of the **raise** estimate is still present to resolve the priority of the raised exception **E(i)** against any others raised in the same instant. The form of the **raise** estimate node is shown in Figure 7-15.

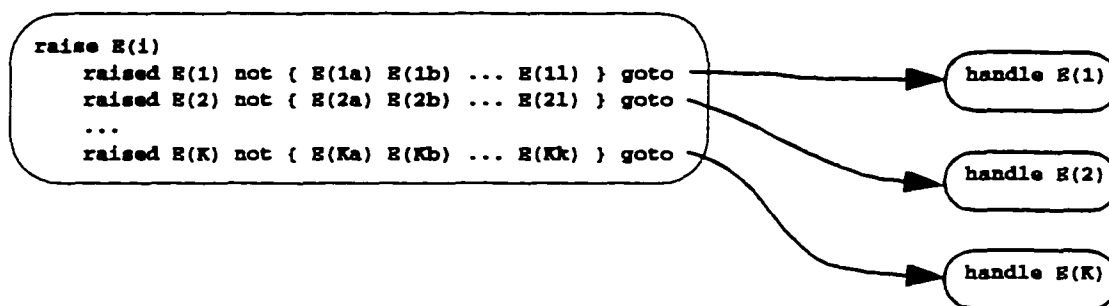


Figure 7-15. The Priority-Based Structure of the **raise** Estimate

The Fork and Collect Estimates

The control flow arising from the invocation and synchronization aspects of the **TCWC** are managed by **fork** and **collect** nodes as shown in Figure 7-16. The **fork** declares a set of control threads that all continue concurrently. Additionally it declares the exceptions that may be raised between the **fork** and the **collect**. The **collect** waits, synchronizing, until all of the concurrent threads have returned. Only then does control flow on to the successors. In this light, the collect is a sort of “halt” in the sense that some control threads will “park” at the **collect** until their siblings have completed or an exception has been raised. If an exception is raised by some child then the raised edge defines the control flow to the appropriate handler where the threads are merged back into a single thread. A more detailed explanation of how the **TCWC** is mapped onto this structure is given in the presentation of the flatten algorithm in Section 7.2.3.2.

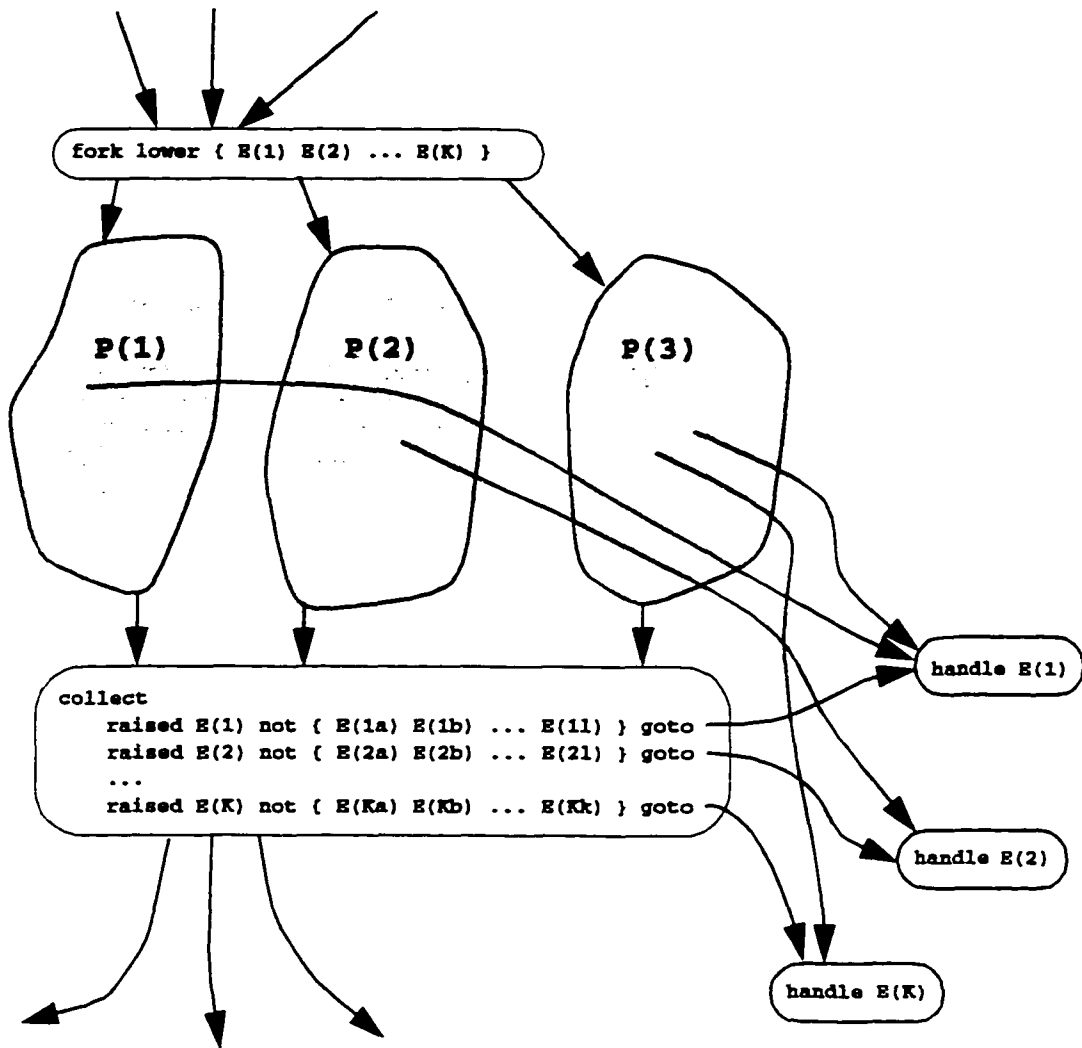


Figure 7-16. The Estimate Graph Template for a **TCWC** Instruction

Signal Estimates

There are three estimate nodes that summarize the signal manipulation instructions. These are the **emit**, **undef** and **require** as shown in Figure 7-17. They have the same function and meaning as the corresponding NDAM instruction. The estimate graph nodes differ only in that they allow a *set* of signals to be mentioned in one node, and in the case of the **emit** no value component is mentioned.

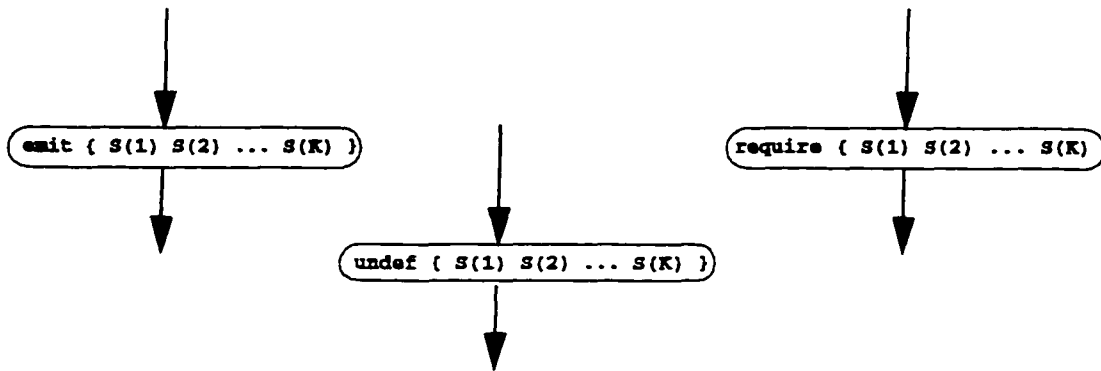


Figure 7-17. The Signal Handling Estimates

Decision Estimates

Control flow choice is treated in two different ways in the estimate graph. The **select** estimate summarizes control flow choice in a manner that is interpreted by the dataflow equations. On the other hand, the **null** estimate abstracts control flow choice that is not interpreted by the dataflow equations. In that case the control thread proceeds along one of the paths out of the **null** estimate node. Both nodes are illustrated in Figure 7-18.

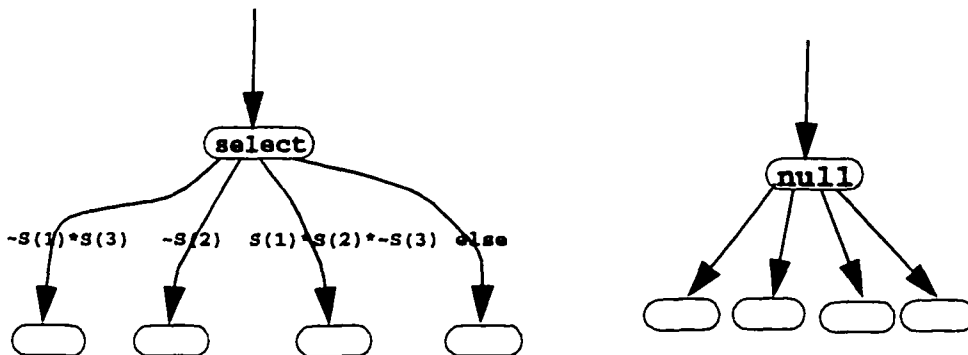


Figure 7-18. The Control-Flow Estimates

7.2.3.2 The Flatten Algorithm

The flatten algorithm, shown in Figure 7-19 constructs the flat estimate graph from the hierarchical process tree representation of the NDAM network. The construction of the graph by that algorithm centers around two features. The first is the reproduction of the

non-hierarchical portions of the process tree directly in the estimate graph. The second is the expansion of the *TCWC* and its implicit behaviors into explicit edges and node annotations in the estimate graph.

```

struct BuildState {
    graph    -- the estimate graph data structure
    visited  -- a table mapping a basic block to its generated estimate
    stack    -- stack of TCWC that have been called
};

flatten()
{
    BuildState state;
    state.graph = a new estimate graph
    add to state.graph a start node
    // Pretend there is single TCWC that calls all of
    // the top-level processes at once and halts if they return
    add to state.graph a fork node (with no lowered exceptions)
    add an edge from the start to the fork
    add to state.graph a collect node (with an empty raised set)

    add to state.graph a halting node (with an empty raised set)
    add an edge from collect to the halting
    add to state.graph a halted node (with an empty raised set)

    initialize tcwc_frame with no when and no handle clauses
    stack.push(tcwc_frame);

    foreach process in the top-level of the process-tree {
        BasicBlock bb = the first executable instruction in the process
        bb.generate(state, fork);
    }
    stack.pop();
}

```

The **flatten** algorithm is completed with the definition of **BasicBlock::generate** listed in Appendix B.

Figure 7-19. The Flatten Algorithm

The Expansion of the **TCWC**

The **TCWC** instruction is the only truly complex instruction. It uses subprocess invocation to encapsulate four distinct effects:

1. The concurrent invocation of child processes.
2. The synchronization of the parent awaiting termination of the children.
3. The interruption of the children by a signal.
4. The preemption of all the children by a raised exception in a child.

For each of the child processes of the **TCWC**, an estimate graph is generated. In the estimate graph this complexity disappears as the hierarchical behavior of the **TCWC** instruction is distributed over the of primitive actions of the estimate graphs of its subprocesses. Within each subgraph, the raised clauses take into account the signal and exception handlers instantiated by the **TCWC**. The conversion from the hierarchical process tree form to the flat estimate graph hinges on this “stacking” of signal and exception handlers. This stacking in turn reproduces the priority mechanisms of the instantiated signal and exception handlers.

7.2.4 Checks on the Flattened Estimate Graph

The flattened estimate graph directly gives the transitions of T_δ for the NDAM: each edge in the transition graph corresponds directly to some enabled transition of the δ -time semantics. Given an arbitrary NDAM program, the task at hand is the determination of whether the particular network has C . When C holds, the operational semantics of a network, presented in Section 7.3.1 is well-defined. There are three checks which establish C and, in sum, establish the causality of the particular network at hand.

Finiteness of δ -Time

A necessary condition for F_δ to be monotonic is that \sqsubseteq_Q be well-defined. This in turn, from Eq 3-23f requires that \sqsubseteq_{S_δ} be well-defined. The necessary and sufficient condition

for \subseteq_{s_δ} to be well-defined is that there be no cycles in T_δ . A necessary condition for *that* property to hold is that there be no cycles in the control flow of the network. The estimate graph exposes such cycles since the **halting/halted** separately effectively isolates control flows across the macrostep boundary. Any cycles still remaining within the estimate graph correspond to potentially excitable cycles in T_δ and vice versa. A simple data-flow computation of reaching definitions [10] is used to determine if there are any cycles: can any estimate graph node “reach” itself on some δ -path.

Modularity

Establishing that T_δ is acyclic establishes that F_δ is monotonic if and only if the domain 2^O is respected as well. This is the condition of modularity:

1. that no signal is read before it is written on any δ -path.
2. that there be only a single emission of any signal on any δ -path.

In practice either of the simulation-based causality checking algorithms is implemented to check for these two conditions. In the explicit case of Section 7.2.2.1 the modification is straightforward: a direct examination of the path $s \rightarrow \delta_1 \rightarrow \delta_2 \rightarrow \dots \rightarrow \delta_k \rightarrow t$ is made. The modifications necessary for the implicit case of Section 7.2.2.2 are a bit more subtle but merely involve defining the domains so that they contain an element T as per Figure 3-19. The T denotes “was redefined” when found on a signal at the end of a macrostep. The multiple assignment of a signal results in a value of T.

Causality

Finally, with the finiteness and modularity of δ -paths established the causality checking algorithms, either explicit or implicit ensure that there is a well-defined state-dependent partial order \leq_Q from all reachable states $Q \in Q^*$.

7.2.5 Focus

In this section, the causality checking problem was defined. Two approaches to causality checking were presented: the explicit and implicit algorithms respectively. These algorithms were presented for the case of a general image semantics where F_{δ} was given. Since F_{δ} for the NDAM is not directly obvious from the hierarchical process tree representation, a flattening algorithm was reported that produced a flat transition-graph representation from the hierarchical process tree of the NDAM network. This representation can support either the explicit or implicit causality checking schemes.

7.3 The Semantics of the Machine

The presentation of NDAM instructions has so far has been informal and intuitive in nature. The intent was to present the material form of the assembly code notation with the formal definition of their microstep behavior being deferred to a later point. Two different semantics are defined for the NDAM in this section. The first is an operational semantics which has the attribute that it defines one possible implementation scheme for interpreting NDAM instructions. The denotation of the operational semantics is a runtime system implementation. The second semantics is the transition relation semantics which has the attribute that its denotations are states and transition relations as per the δ -time microsemantics of Section 3.4.3.

7.3.1 The Operational Semantics of δ -Time

The operational semantics is defined by a particular runtime system implementation. This is depicted in the diagram of Figure 7-22. The non-abstract domain M_{op} is the set of all possible runtime instances with a certain style with the actual domain definitions remaining only intuitively defined for this presentation. As well the projection Π from the domain M_{op} of runtime system into the fully abstract semantics $M = (Q, T)$ is left intuitively defined. As might be expected, the definition of Π substantially amounts to ignor-

ing the implementation artifacts used to implement δ -steps control flow, synchronization and completion.

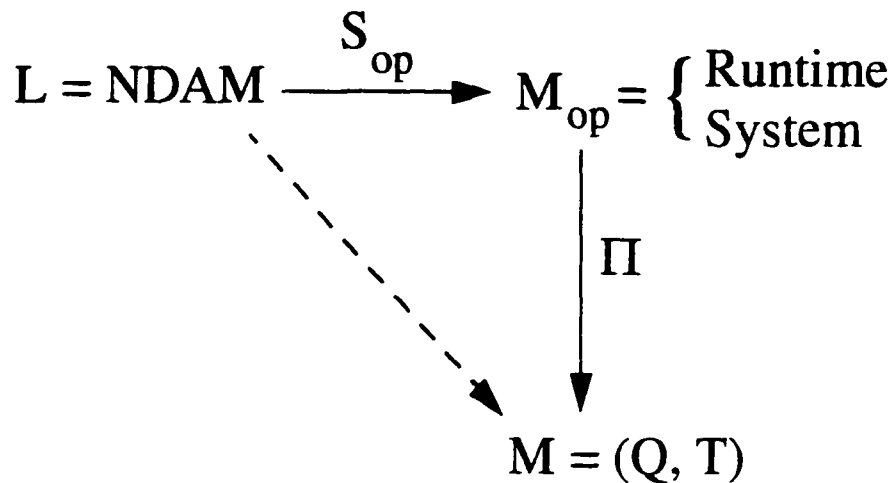


Figure 7-20. The Operational Semantics of the NDAM

7.3.1.1 The Literal Fixed Point Method

The key idea behind the runtime system is that the code body literally implements the approximator functional \mathcal{F}_δ of the NDAM network. The repeated execution of this code body converges after a number of steps at the relative least fixed $\mu\mathcal{F}_\delta$. This convergence is based on a principle of idempotence, namely that there are certain synchronization-type structures within it which if executed once can be executed an arbitrary number of times. All that is required of the implementation is some means of ensuring idempotence and of detecting when the point of idempotence has been reached. This method is here called the *literal fixed point method*. An implementation of this semantics is described in detail in Section 7.5 so the presentation here is restricted to describing its design from a semantic perspective: why it works, deferring the declaration of what it is to the later presentation.

The analogy between the NDAM network and the approximator functional analogy means that two properties must hold. The first is that the state of the NDAM network at

the end of the macrostep must be a fixed point: repeated evaluations of the network must not result in any further progress. Secondly, the state of the NDAM network within a macrostep represents an approximation of the completion towards this fixed point. Clearly the state of a network is defined by the state of its processes and the status of the signals so these properties must be supported on each process and the set of signals.

7.3.1.2 A Process' State

A process' state is summarized in its program counter and whether or not it is *done* or *complete* for the macrostep instant. A process which is done has clearly reached its fixed point; it need not be evaluated any further in the instant. A process may become done for the instant but be restarted by its parent process. These two invocations represent distinct instantiations of the child process within the parent's lifetime. So long as a child is never not done when a parent is done the uniqueness of the fixed point is guaranteed.

The second aspect of a process is the notion of its approximate completeness within the instant. This approximation is summarized in the process' program counter which is defined in a non-flat domain with the non-trivial internal ordering defined by Eq 3-22f. Thus a NDAM process is well-defined when its control flow paths in the context of its use in the network is faithful to the ordering of Eq 3-22f. The necessary condition for this to be true is that there be no cycles in the domain S_{δ} of Eq 3-21.

For the general case of Eq 3-22f, S_{δ} need not be a Cartesian product of S_{PC} and S_{DATA} ; it is existentially declared with its $\subseteq_{S_{\delta}}$. This assumption drastically simplified the development in Section 3.4.3. In practice however, S_{δ} is not existential, rather it is a Cartesian product, $S_{\delta} = S_{PC} \times S_{DATA}$, of the non-flat domain S_{PC} of program counter values and the Cartesian domain S_{DATA} of the register values. A sufficient condition for fidelity to Eq 3-22f is that $\subseteq_{S_{PC}}$ be well-defined. This is another way of saying that the uninterpreted¹ control flow paths are loop free. Since S_{PC} consists of the combination of

program counters of the concurrent processes, the determination of when the S_{PC} domain is well-defined is non-trivial. The causality-check test of Section 7.2 verifies that S_{PC} is well-defined for a given NDAM network instance.

7.3.1.3 Outputs

Outputs in the operational semantics are a Cartesian domain as defined in Eq 3-24. The extra requirement for the implementation is an explicit way to test for \perp on a signal. In the implementation of Section 7.5 this is accomplished with an extra bit for each signal recording whether the signal is defined (yet) or not.

It must be stressed that this extra bit is an implementation artifact of *this particular non-abstract semantics*, the literal fixed point method, which is expected to be implemented in software. The extra signal status bit is explicitly *not* a requirement of synchronous semantics. This should be clear from the presentation of δ -time in Section 3.4.3 and the microsemantics of synchronous languages in Section 4.3.4.4. As well, there exist non-abstract semantics that do not use an explicit representation for \perp are the lookup-table scheme and the hardware semantics used in other Esterel compilers. These are reviewed for completeness in Section 7.5.3.

The test for the non-definition of a signal allows for the synchronization of the **'require'** instruction operations to be defined. At a higher level, the semantics defines that a **'require'** instruction blocks just when a signal has the value \perp . This ensures that an undefined signal is never read and thus the semantics remains faithful to Kahn's data-flow conditions. This has the unexploited but significant effect of allowing **'require'** barrier instructions to be removed when it can be shown that signal read operations (*e.g.* a **'present,'** **'presence,'** or **'selection'**) never occur before a signal assignment

1. *i.e.* ignoring the contributions of data-flow. Thus whether the consequence of **1f** condition is invisible in sort of analysis.

(e.g. an **emit**) in any constructed scheduling of the network.

7.3.1.4 Synchronization

Each NDAM instruction manipulates the program counter in some way. Most merely compute some register-transfer operation and assign the program counter to the successor instruction. These instructions increase relative to $\equiv_{S_{PC}}$. In contrast, there is a class of instructions which increase relative to $\equiv_{S_{PC}}$ only when certain conditions about 2^0 obtain. These, by definition, are the class of synchronization instructions and they are distinguished by effectively *blocking* the execution of a process in δ -time until the definition of the mentioned signal becomes available. This class is typified by the **require** but there are aspects of it in the **wait** and **TCWC** as well. The key to the synchronization instructions is that they block in δ -time awaiting a signal's definition: *either* its presence or absence. Synchronization against the definitive presence or the definitive absence is then exclusively restricted to the macrostep level (e.g. Esterel's **do...watching**).

7.3.1.5 Completion

The other significant aspect of the operational semantics is the treatment of completion: the halting of a process, or the termination when it exits. The relevant instructions are **halt**, **exit** and **wait** (which is best thought of as a **halt** and **require** merged together). These instructions implement the idempotence aspect of the network. In the framework of domain theory, the program counters of completion instructions are elements of the domain S_T as used in Eq 3-23f. The halting instructions are operationally faithful to Eq 3-23f by *not* incrementing the program counter and at the same time declaring the process to be *done*. Further invocations of the process have no effect.

7.3.1.6 Focus

The previous sections highlighted the design of the a non-abstract semantics for the

NDAM. These elements are exploited in the runtime implementation described in Section 7.5. The presentation of that section explicitly enumerates the operational behavior of all the NDAM instructions. Additionally the single extra-network runtime routine, **EVAL**, which makes this scheme feasible is shown.

7.3.2 The Relational Semantics of δ -Time

The transition relation semantics of δ -time has denotations which are states and transition relations as illustrated in Figure 7-22. The construction of elements of this domain follows directly from the computational semantics of δ -time in Section 3.4.3 coupled with the process tree flattening procedure defined in Section 7.2.3.

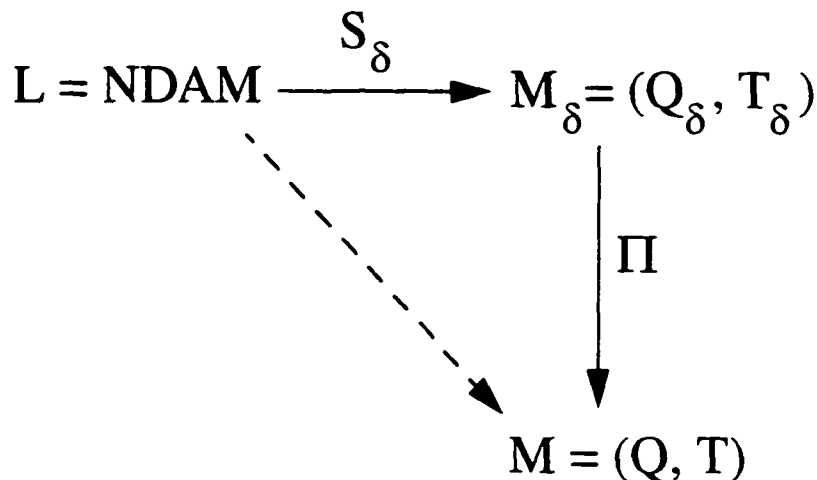


Figure 7-21. The Relational Semantics of the NDAM

The elements of the computational semantics are:¹

1. *Temporal Analysis*

By design, the δ -time of Section 3.4.2.1.

2. *Domain Analysis*

By construction, the domains of Section 3.4.2.2.

1. For simplicity, the backward case is elided from the following enumeration. The backwards case follows directly.

3. *The Primitive Transition relation T_δ*

The flattened flow graph from Section 7.2.3 is the transition relation T_δ .

4. *The Primitive Image Functional F_δ*

A NDAM instruction takes one δ -step: $F_\delta = \lambda Q. (\exists c. Q(c) \wedge T_\delta(c, n)) [n/c]$

5. *The Approximator Functional \mathcal{F}_δ*

The standard definition: $\mathcal{F}_\delta = \lambda F. \lambda Q. F_\delta \{Q\}$

6. *The Approximated Image Functional \bar{F}*

The fixed point relative to a single “first step:” $\bar{F} = \mu_{F_\delta} \mathcal{F}_\delta$,
more importantly with absorbing end conditions: $\bar{F}_{Q_i \{Q_{i+1}\}} = \mu_{F_\delta \{Q_i\}} \mathcal{F}_\delta$.

7. *The Projection Π to Full Abstraction*

As stated in Section 3.4.3.7.

8. *Observations*

Observations follow directly from those stated in Section 3.4.3.8.

The key element here is the extraction of the microtime transition relation T_δ from the operational definition. Having been extracted, the semantic objects can be manipulated directly and in their own right. For example, T_δ can be subjected to various forms of abstraction and optimization as suits the needs of its material representation. Abstraction and optimization opportunities can appear along the axes of control flow, synchronization and δ -step paths. The estimate graph of Section 7.2.3.1 can be seen as an instance of these first two sorts of abstraction since it abstracts away the data-centric operations leaving only a “may” estimate of the true control flow and inter-process synchronization patterns.

The extraordinarily fine granularity of T_δ can be addressed from this vantage point as well. T_δ represents the set of single steps that can be taken in δ -time. A transition relation $T_{2\delta}$ which consists of the two-step paths in δ -time is defined as:

$$T_{2\delta}(c, n) = \exists i. T_\delta(c, i) \wedge T_\delta(i, n)$$

In fact, this coalescing of δ -steps into δ -paths can be carried on arbitrarily many times subject only to the practical size restrictions of the material representation of the transition relation $T_{k\delta}$. Of course in the limit, this process results in “compiling away” all δ -steps

which is of course exactly the definition of the fully abstract transition relation. Thus there is some length n at which $T_{n\delta} = T$ and full abstraction is reached.¹ In fact, this sort of coalescing of microstep transition relations into the larger multi-step version is exactly what is being accomplished in the quantification ordering schedule of σ -time from Section 3.4.4.

7.3.3 Focus

The two previous sections have presented the semantics of the NDAM. The first semantics was operational and defined one particular implementation strategy. The second semantics was relational and defined the non-abstract macrostep image computation as a series of microstep image computations which were fully abstract in δ -time. This construction neatly sidestepped the problem of determining if the operational and relational semantics were the same: they are *necessarily* the same because one was derived from the other in a lossless transformation (the flattening procedure of Section 7.2.3).

Reduced to its essence, the claim here is that the method used in this section of extracting the transition relation semantics from the operational semantics is the only well-defined way to interface a high-level language to formal verification. This is the essence of Gordon's *WYSIWYV* aphorism: "*What You Simulate is What You Verify*" [299]. In turn, the relationship of a particular non-abstract semantics to the fully abstract case is defined by the projection Π . The RMC Barrier Theorem states the conditions that must apply for that projection to be well-defined.

In contrast, other methods of defining the "verification semantics" of a language necessarily introduce some syntax-based interpretation of the language, or arbitrarily-defined synthesis step based on a language subset.² Examples in this regard abound. The Synchron-

1. The length of a δ -path is state-dependent, however the absorbing end conditions of Figure 3-15 effectively stretch out all δ -paths to have infinite length so no information is lost in this construction.

nous VHDL subset is one example which is treated directly in this work, in Section 6.3.2. Another example is the syntactic subsetting of Verilog's $\overline{RMC}/\overline{R}_\delta M_\delta C_\delta$ semantics for embedding into the RMC /vacated semantics of the generalized gate model [49] [164].

7.4 High-Level Language Compilation

The features of NDAM make the compilation of the relevant source languages rather straightforward. This section presents the compilation recipes for several languages. All the compilation chains are as shown in Figure 7-22. A runtime that supports the operational semantics of δ -time is presented in Section 7.5.

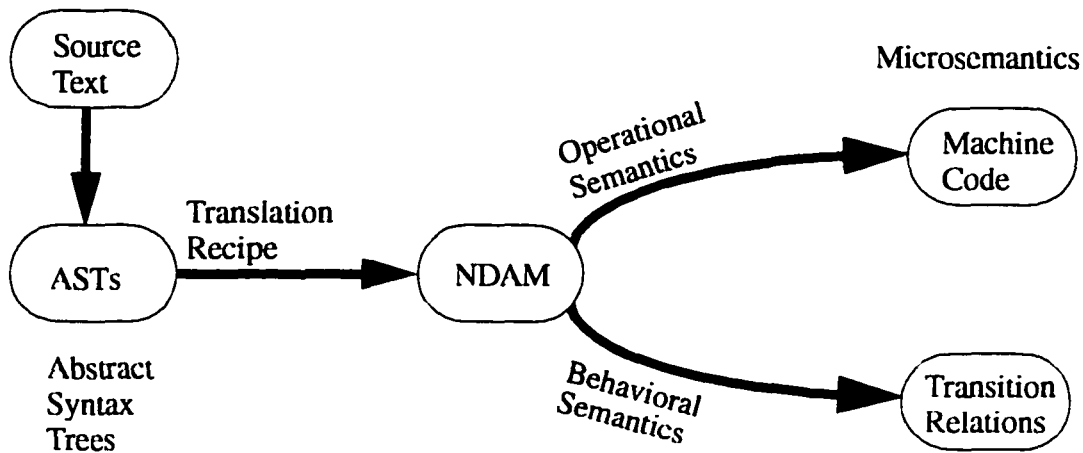


Figure 7-22. The Two-Level Approach with NDAM Assembly

7.4.1 Esterel

Given that the Esterel source code has been parsed then the recipe of Figure 7-23a. can be used to produce a NDAM representation of an Esterel program. The conversion according to the rewrite rules proceeds by recursive descent along the abstract syntax trees produced in the front-end of the compiler. No attempt at optimization or (hard) register

2. This has been referred to as the “...well, then we won't support that” method [93] [47] [49].

allocation is performed at this early stage and the Esterel statements copymodule or run are treated as purely syntactic objects (*i.e.* they are expended in-place) as per the formal semantics of Esterel. An implementation of this compilation scheme exists.

Esterel Statement	NDAM Instruction
halt	halt
nothing	nothing
exit	exit
exit TRAP	raise E(TRAP) R(pure)
exit TRAP(exp)	compute exp into temporary R(t0) raise E(TRAP) R(t0)
VAR := exp	compute exp into temporary R(t0) R(VAR) := R(t0)
emit SIG	emit S(SIG) R(pure)
if exp then statement1 else statement2 end if	compute exp into temporary R(t0) if not R(t0) goto L(s1) generate code for statement1 goto L(next) L(s2): generate code for statement2 L(next):
present SIG then statement1 else statement2 end if	require S(SIG) present not S(SIG) goto L(s1) generate code for statement1 goto L(next) L(s2): generate code for statement2 L(next):
await immediate SIG	require S(SIG) present S(SIG) goto L(around) wait S(SIG) L(around):
await S(SIG)	wait S(SIG)

Figure 7-23a. A Recipe for Compiling Esterel into NDAM Assembly

7.4.2 Synchronous VHDL

This same compilation scheme can be used for VHDL to implement the Synchronous VHDL subset. Such an implementation was described in previous work [47]. As with the Esterel compiler the translation scheme follows traditional intermediate code generation practices to transform the abstract syntax trees into NDAM assembly codes. Since that

Esterel Statement	NDAM Instruction
<pre>loop statement end loop</pre>	<pre>L(loop) : generate code for <i>statement</i> goto L(loop)</pre>
<pre>repeat N times statement end repeat</pre>	<pre>compute N into temporary R(t0) L(loop) : if not R(t0) goto L(done) generate code for <i>statement</i> R(t0) := dec R(t0) goto L(loop) L(end) :</pre>
<pre>statement1 ; statement2 ; ... statementN</pre>	<pre>generate code for <i>statement1</i> generate code for <i>statement2</i> ... generate code for <i>statementN</i></pre>
<pre>[statement1 statement2 ... statementN]</pre>	<pre>try call P(child-1) I(0); call P(child-2) I(0); ... call P(child-N) I(0) foreach <i>statement-i</i> add new child processes process P(child-1) is interface I(0) L(0) L(0) : generate code for <i>statement-i</i> end process P(child-1)</pre>
<pre>run M [<i>substitution-list</i>] copymodule M [<i>substitution-list</i>]</pre>	<pre>look up module M translate its sole instruction while performing the substitutions in <i>substitution-list</i></pre>

Figure 7-23b. A Recipe for Compiling Esterel into NDAM Assembly

transformation is straightforward and is described elsewhere. it is not presented here explicitly.¹

As mentioned in Section 6.3.2 the discrete event semantics of full VHDL is $\overline{RM}\overline{C}$ so the Synchronous VHDL subset has a modularity checking obligation in addition to a causality checking obligation. Also, as mentioned there, certain process network structures that have an existing topological order to them (*e.g.* combinational logic networks) are

1. In any case, were it presented, it would substantially follow Figure 7-23 with the only difference being the use of VHDL syntax in the left-hand column.

Esterel Statement	NDAM Instruction
<pre> var VAR1 := exp1: type in statement end var </pre>	<pre> try call P(child) I(0) add a new child process process P(child) is interface I(0) L(0) register R(VAR1) T(type) L(0): compute exp1 into temporary R(t0) R(VAR1) := R(t0) generate code for statement exit end process P(child) </pre>
<pre> signal SIG: type in statement end signal </pre>	<pre> try call P(child) I(0) add a new child process process P(child) is interface I(0) L(0) signal S(SIG) T(type) L(0): undef S(SIG) generate code for statement exit end process P(child) </pre>

Figure 7-23c. A Recipe for Compiling Esterel into NDAM Assembly

naturally modular and causal. However, failing the sufficiency conditions defined in Section 6.3.2.4 of 1) static sensitivity lists for all processes and 2) acyclic process network structure, some semantics-based check must be formulated to ensure that the admitted VHDL programs are M as well as C . The causality check defined in Section 7.2 is suitable in this regard because it necessarily checks that modularity is not violated: that 1) signals are singly assigned across δ -time and 2) no signal is ever read before it is written across δ -time.

7.4.3 (Synchronous) SpecCharts

In addition to Synchronous VHDL one could also posit the existence of a “Synchronous SpecCharts.” In fact, Gajski *et al.* give an explicit transformation from the SpecCharts language to standard VHDL.¹ There is no theoretical or practical reason why this translated

Esterel Statement	NDAM Instruction
<pre> trap TRAP: type in statement handle TRAP do statement2 end trap </pre>	<pre> try call P(handle) I(0) add a new child process for the exception handler process P(handle) is interface I(0) L(0) exception E(TRAP) T(type) L(0): try call P(body) I(0); catching handle E(TRAP) goto L(1) goto L(2) L(1): generate code for statement2 L(2): exit end process p(handle) add a new child process for the body process P(body) interface I(0) L(0) L(0): generate code for statement exit end process P(body) </pre>

Figure 7-23d. A Recipe for Compiling Esterel into NDAM Assembly

VHDL or even the original SpecCharts form itself could not be subjected to the same modularity checking and causality checking that Synchronous VHDL undergoes.

While no attempt has been made in this work to formulate a “Synchronous SpecCharts” variant, the possibility is so obvious that it is worth stating explicitly. Indeed a Synchronous SpecCharts, with its dual graphical/textual representation would provide the features hoped for in the potential marriage of the textual description style of Esterel and the (synchronous) StateChart-style of Argos.¹

1. See Gajski *et al.* [277], Chapter 5.

1. As mentioned in Halbwachs [320].

7.5 A Runtime Implementation¹

From the operational semantics an implementation based on dynamic potential set computation follows rather directly. All that needs only to be constructed is a runtime system, data structures and a scheduler, which dynamically computes the relevant potential sets and ensures that no NDAM instruction is executed before its necessary signals are defined. Such a runtime system does not check causality but rather depends on the fact that the network and process tree upon which it operates is *RMC*. This must have been previously verified.

In the design of the runtime system there are two issues:

1. how to organize the runnable processes and,
2. how to represent and compute the potential sets.

The technique used here is to take advantage of the fact that both pieces of information can be *mostly* determined from a static analysis of the network. The idea is to encode as much information as possible in tables or within the processes' code bodies and to perform only minor corrections at runtime.

In the first case, the runnability of processes is reported back to its caller by a status code returned by the function which implements a process' code body. That status code indicates one of four conditions: whether the process has *halted*, has exited and is now *terminated*, is *synchronizing* awaiting more signals to become defined or has *raised an exception*.

In the second case, a signal's potential for emission is estimated based on information which is encoded in tables generated at compile-time. These tables are indexed by a pro-

1. This implementation scheme was originally introduced in Edwards' prototype Esterel compiler [246]. This scheme can also be seen to have substantial similarities with Druisinsky and Harel's "hardware semantics" of StateCharts [241] [242].

cess' program counter. The only runtime action needed therefore is the aggregation of this information across all the actively running processes in the process tree. Code generation is thus intrinsically tied to the runtime scheduling system and the interactions required between processes and the runtime system. The following sections describe the code generation templates and the NDAM Runtime System (NRS).¹

7.5.1 Code Generation

The emulation of NDAM at the machine level is straightforward save for the control-type instructions such as **halt**, **exit**, **wait** and **TCWC**. With careful code generation, the implementation template for many NDAM instructions can correspond directly to one or more machine instructions (e.g. **goto**, **if**, **:=**, unary and binary operations). For the control-type instructions however, like **halt**, **exit**, **wait** or **TCWC**, the virtual program counter (pc) of the process has different properties that do not directly relate to those of the machine-level pc. For example, in **halt** and **exit**, the virtual pc is left unmodified, thereby indicating that the fixed point has been reached relative to that process' execution. The code templates for the NDAM instructions are shown in Figure 7-24.

Of the control-type instructions, the **TCWC** is the most complicated because it must manage the execution of its child processes. Its code template is broken out and shown in Figure 7-25. In practice the implementation of **TCWC** is complicated enough that it is kept in a separate library; it is not inlined. As such the **TCWC** forms a sort of crude dynamic scheduler for the NDAM subprocess tree. Fortunately, the scheduling and recovery algorithm shown in Figure 7-25 is generic enough that it can be table driven thereby saving on implementation code size while remaining highly general.

1. The full implementation of NRS can be found in Appendix D.

NDAM Instructions	NDAM Runtime Action	Notes
halt	return HALTED	pc remains here
exit	return HALTED	pc remains here
raise E(i) R(j)	return RAISED	pc remains here
require { S(1), ... }	if any S(1) is unknown return SYNCING else pc := pc + 1	
wait { S(1), ... }	phase1: pc := phase2 return HALTED	There are two phases to the wait instruction. The first phase is entering the wait and the process halts. The second phase occurs only in subsequent instants. It synchronizes until all signals are known and then is either HALTED or continuing on.
	phase2: if any S(1) is unknown return SYNCING else if no S(1) is present return HALTED else pc := pc + 1	
try call P(1) I(0); call P(2) I(0); ... call P(N) I(0); watching when C(1) S(1) goto L(s1); when C(2) S(2) goto L(s2); ... when C(K) S(K) goto L(sK); catching handle E(1) goto L(e1); handle E(2) goto L(e2); ... handle E(K) goto L(eK)	See Figure 7-25	There are three phases to TCWC.
<i>Any other NDAM instruction</i>	pc assigned as per the normal opcode flow	Either the instruction assigns into pc directly or pc := pc + 1

Figure 7-24. Templates for C Code Generation from NDAM Assembly

7.5.2 The Runtime System

The key to the runtime system is that each process' code body is idempotent within the macrostep. It can be repeatedly executed until all of its children have reached a stable point (this is the fixed point of δ -time). This can be seen in the rules for **exit**, **halt** and

wait in Figure 7-24. The **halt** and **exit** reach a fixed point by virtue of not assigning a new pc value. The **TCWC** indirectly reaches a fixed point when its child processes reach a fixed point. The **TCWC** is thus the driver for all of its children in that it delegates its reevaluation to its children, until either they become stable, or an exception is raised in one of them. At the top-level there is a network-level **EVAL** algorithm which simply evaluates the top-level process until it indicates that its no longer runnable. This is the execution of a network across δ -time to its fixed point. The following sections describe **EVAL** and demonstrate its potential set evaluations.

7.5.2.1 The Network-Level **EVAL**

The network-level **EVAL** is the outer loop the scheduler. All it does is repeatedly evaluate the top-level process until no more progress can be made anywhere in the network. After each reevaluation the outstanding signal potentials are examined to determine which signals can never occur in any future δ -step of the current macrostep. For signals that are **EMITTED**, having been emitted but not yet marked as **PRESENT**, the state is changed to **PRESENT**.¹ For signals that are **UNKNOWN**, the state is changed to **ABSENT**. The evaluation process is repeated. Ultimately there will be no further information about signal presence or absence that can be generated: the fixed point will have been reached in a finite number of iterations. At that point, the evaluation loop of *step 4* terminates and the macrostep is finished. The network-level **EVAL** algorithm is shown in Figure 7-26

7.5.2.2 Static and Dynamic Potential Sets

The potential set of a network is the union of the potential sets of the runnable process. In turn, the potential set of a process is the potential set, **POT**, associated with its program counter. Fortunately the potential set associated with each program counter is a *mostly*

1. This trick, making explicit the distinction between **EMITTED** (having executed the emit operation) and **PRESENT** (formally recognizing and marking the emission) was first proposed by Gonthier [295].

```

if (this process is stable)
    return STABLE;
switch (phase) {
case TAIL:
    assign each subprocesses' pc at the label of its interface I
    initialize any signal counters in when clauses
    mark the exceptions of the TCWC as LOWER
case SYNC:
    // This is the intermediate SYNCING phase
case HEAD:
    if (any of the guarding signals is UNKNOWN)
        return SYNCING
    foreach guarding signal that is present.
        decrement its counter (if any)
    foreach when clause {
        if (signal when.S is present and when.C has expired) {
            pc = when.pc,
            return CONTINUING
        }
    }
}
foreach subprocess {
    (re)call it.
    noting if any returned RAISED, STABLE or SYNCING
}
if (any returned SYNCING) {
    pc = pc'SYNC
    return SYNCING
}
if (any returned raised) {
    if (raised-exception depth > the process's depth in the process tree) {
        // process.pc stays the same
        return RAISED;
    } else {
        foreach handle clause {
            if (exception handle.E was raised) {
                pc = handle.pc;
                return CONTINUING;
            }
        }
    }
}
if (any returned stable) {
    mark this process as stable too
    pc = pc'TAIL;
    return STABLE;
}
pc = pc+1
return CONTINUING;

```

Figure 7-25. The Code Template for the **TCWC**


```

EVAL(network)
{
  1. mark all processes as not STABLE
  2. read input signal values and mark their presence or absence
  3. mark all other signals as 'UNKNOWN'
  4. do {
      s = EXEC(P[0].code);
      clear all signals as having no potential
      MARK(P[0]);
      for all signals S[i] {
          if (! S[i].potential) {
              if (S[i].presence == EMITTED)
                  S[i].presence = PRESENT;
              if (S[i].presence == UNKNOWN)
                  S[i].presence = ABSENT;
          }
      }
      while (s != TERMINATED);
  5. write all output signals to the environment
}

```

Figure 7-26. The Network EVAL Algorithm

constant function of the program counter. In particular, it is entirely unrelated to the potential sets of sibling or parent processes. This means that for most pc locations, the potential set computation can be implemented with a simple table lookup. Such tables can be computed at compile-time with simple data-flow estimation computation. As such the tables hold *may* information.¹ The example shown in Figure 7-27 illustrates the static property of most potential sets.

There is one special case where the set of signals potentially emitted by a process is not uniquely associated with its program counter. That case, predictably, is the ever-troublesome *TCWC*. An example of this sort of situation is illustrated in Figure 7-28. There, the

1. Hence the name *potential* set: the set is the set of signals which may potentially be emitted in a future δ -step of the instant.

```

type t(bool) 2
signal S(1) t(bool)
signal S(2) t(bool)
signal S(3) t(bool)
L(1):      R(1) := selection S(1)
L(2):      if R(1) goto L(6)
L(3):      R(2) := R(1) + R(2)
L(4):      emit S(2) R(2)
L(5):      goto L(9)
L(6):      emit S(3) R(1)
L(7):      wait S(1)
L(8):      goto L(1)
L(9):      halt

```

POT	Label	Instruction
S(2).S(3)	L(1)	R(1) := selection S(1)
S(2).S(3)	L(2)	if R(1) goto L(6)
S(2)	L(3)	R(2) := R(1) + R(2)
S(2)	L(4)	emit S(2) R(2)
∅	L(5)	goto L(9)
S(3)	L(6)	emit S(3) R(1)
∅	L(7)	wait S(1)
S(2).S(3)		
S(2).S(3)	L(8)	goto L(1)
∅	L(9)	halt

Figure 7-27. An Example of Static Signal Potentials

POT set for the **TCWC** at location **L(1)** depends on the state of its child processes **P(1)** and **P(2)**. The variance is described by the table which accompanies the figure.

In the case of a process whose pc is currently at a **TCWC** has several possibilities for future behavior within the instant which are enumerated in Figure 7-29. Most depend on what happens in the children, though some do not.

1. The children can execute more and emit more signals and then *all of them* can halt without exiting.
2. The children can execute more and emit more signals and *one of them* could raise an exception.
3. The children can execute more and emit more signals and *all of them* can exit. The **TCWC** continues on to its successor instruction.
4. A signal that the **TCWC** is guarding against could occur. The **TCWC** continues on to the signal's handler.

Figure 7-29. The Reasons for Variance in POT at a **TCWC**

7.5.2.3 The MARK Algorithm

The major aspect of the dynamically-iterated fixed point technique is the estimation and marking of signals which will never occur in any future δ -step of the instant. In the **EVAL** algorithm this determination is made by the **MARK** algorithm which is shown in Figure 7-

A TCWC with a dynamic POT set.

```

process P(0) is
type t(bool) 2
constant R(1) t(bool) := 0
signal S(i1) t(bool)
signal S(i2) t(bool)
signal S(i3) t(bool)

signal S(o1) t(bool)
signal S(o2) t(bool)
signal S(o3) t(bool)
signal S(o4) t(bool)
signal S(o5) t(bool)
signal S(o6) t(bool)
signal S(o7) t(bool)
signal S(o8) t(bool)

exception E(1) T(1)
L(0): null
L(1): try
    call P(1) I(1);
    call P(2) I(1);
    watching
    when S(i1) goto L(4)
    catching
    handle E(i1) goto L(5)

L(2): emit S(o1) R(1)
L(3): halt
L(4): emit S(o2) R(1)
L(5): goto L(1)
L(4): emit S(o3) R(1)
L(5): halt
end process P(0)

process P(1) is
interface I(1) L(0)
L(0): emit S(o4) r(1)
L(1): wait s(i2)
L(2): emit S(o5) R(1)
L(3): wait S(i2)
L(4): goto L(2)

process P(2) is
interface I(1) L(0)
L(0): emit S(o7) r(1)
L(1): wait s(i3)
L(2): emit S(o8) R(1)
L(3): wait S(i3)
L(4): raise E(1) R(1)
end process P(2)

```

Network POT	Process Locations and POTs					
	P(0)		P(1)		P(2)	
S(o4), S(o7)	L(0)	S(o4), S(o7)	n/a	n/a	n/a	n/a
S(o1), S(o3), S(o5), S(o8)	L(1)	S(o1), S(o3)	L(1)	S(o5)	L(1)	S(o8)
S(o1), S(o3), S(o4)	L(1)	S(o1), S(o3)	L(3)	S(o4)	L(3)	∅

Figure 7-28. An Example of Dynamic Signal Potentials at a TCWC
 30. It uses two compile-time generated tables which are associated with each program counter location:

- U, the potential set strictly within the process.
- C, the children below the process at pc (is vacuous if not a TCWC).

The C table is empty except for TCWC locations in which case it holds pointers to the

tables for the children mentioned in the **TCWC**. In that case, the **MARK** algorithm recurs.

```
MARK(process)
{
  1. if the process is STABLE
     return;
  2. foreach signal s in process.U[process.pc] {
     s.potential = TRUE
  }
  3. for child process p in process.C[process.pc] {
     MARK(p)
  }
}
```

Figure 7-30. The **MARK** Algorithm

7.5.3 Other Schemes

The direct interpretation of NDAM instructions by the method just presented takes advantage of the existence of an observable fixed point in the network's execution. Within each instant the network is repeatedly executed until no further progress can be made. Thus the operational semantics has a direct analogy to the computation of the fixed point which defines a macrostep. This scheme, which can loosely be called the *literal fixed point method*, is neither the first nor the only implementation scheme for imperative synchronous semantics. There are two other methods which have been developed for the Esterel language.¹

Direct Interpretation of the Process Algebra

The earliest implementation scheme for Esterel, due to Cosserrat [207], was the explicit rewrite of Esterel programs, interpreted as a process algebra. In the process algebra view the δ -steps are individual rewrites of the program according to a sort of "event derivative" operator. The macrostep is, as per the semantics, the fixed point of these δ -step derivative

1. The historical development of Esterel is summarized in the dissertations of F. Mignard [525] and F.-X. Fornari [264].

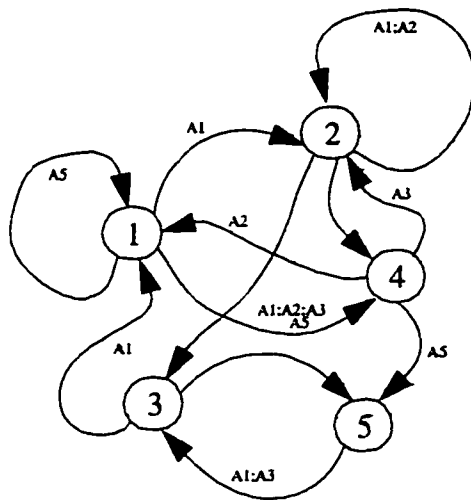
operators. This scheme was made concrete in the Esterel *v1* compiler. Interestingly, while this is not a deliverable implementation scheme, the process algebra approach has again attracted recent interest in conjunction with bisimilar minimization of systems described in Esterel [228].

Compilation to Automata

The next advance in implementation was arrived at when it was realized that the finite set of program derivatives could be encoded efficiently in the form of control automata which governed the execution of elements of action tables. The general style of such automata-based implementations are depicted in Figure 7-32. Implementations in this form were completed by Couronné [81] [213] and Gonthier [295] becoming the Esterel *v2* and *v3* compilers respectively. These compilers transform the program source text into intermediate forms which are then converted into an “host” language such as C, Ada, Lisp and the like. Supporting the automata style of implementation are the two internal representations **oc** and **ic** [577]. The **oc** code is a flat representation of the control automata and action table while the **ic** code provides a hierarchical and instruction-oriented representation of the program. The operational definition of the automata representations are well-defined in the mathematical sense, however the actual implementation is often problematic because the control automaton suffers the same state explosion problem; the control automaton is represented explicitly in tables in this implementation.

Compilation to Circuits

More recently a “hardware semantics” has been developed for Esterel [76] [78] [264] [525]. In this implementation style the control automaton is represented in the form of a generalized circuit that *computes* the successor state from the current state as depicted in Figure 7-32. This style is distinct from the automata style where the successor state is *declared* in a table. Whether this circuit is implemented in the form of transistors or



A1:	$v := x + y$ emit O2(x) call F(v)
A2:	emit O3(v)
A3:	emit O1
A4:	emit O4(x) call F(x) call $v := v + t$
A5:	emit O5

Control Automaton

Action Table

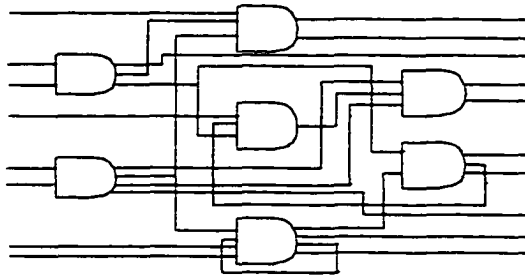
Figure 7-31. Implementation of Esterel in Automata

machine instructions is less important than the fact that the state explosion problem is avoided.

One interesting property of the control circuit is that the combinational logic may have internal cycles due to the direct translation from the source level. These cycles correspond to potential causality problems in the sense that certain kinds of the combinational cycles result in well-defined global behavior while others result in infinite oscillations. Causality checking thus becomes, in this formulation, the question of whether the *cyclic* combinational network has a well-defined behavior. Cyclic networks with well-defined global behavior are said to be *well-caused*. Algorithms and data structures to represent and identify well-caused networks have been proposed by Malik [494] and Burch *et al.* [142].

7.6 Review

An abstract computational model called the Nondeterministic Abstract Machine has



Control Circuit

A1: v := x + y emit O2(x) call F(v)
A2: emit O3(v)
A3: emit O1
A4: emit O4(x) call F(x) call v := v + x
A5: emit O5

Action Table

Figure 7-32. Implementation of Esterel in Circuits
 been presented. The NDAM assembly code was shown to be suitable for use in imperative synchronous language compilers and to this end compilation schemes were shown for the synchronous language Esterel and for the synchronous subset of VHDL (Synchronous VHDL). Additionally it was posited that a synchronous subset of the SpecCharts could be formulated along the same lines. It was pointed out that the nondeterminism of the NDAM is strictly selection nondeterminism. Nondeterminism is potentially important in the formal verification context as an abstraction mechanism so long as it preserves modularity. Selection nondeterminism preserves modularity and so is compatible with the $RM\bar{C}$ required for synchronous language semantics.

Because all non-abstract synchronous semantics are $RM\bar{C}$ and all necessarily have a causality checking obligation since time must run forward in any practical implementation scheme. NDAM networks are no exception to this rule. The causality checking problem has been posed in the context of an arbitrary NDAM network with the aid of an Estimate Graph. The Flattening Algorithm transforms the hierarchical NDAM network into a flat Estimate Graph where the concurrency and control flow ordering in the network are made

explicit. Using this representation, the causality checking algorithm was defined. That causality checking algorithm also had the benefit of verifying modularity and so could be used to identify the synchronous subset of \overline{RMC} semantics such as the discrete event semantics of VHDL or Verilog.

Working under the assumption of admitting only instances where RMC holds, the operational and relational semantics of the NDAM networks were defined. These semantics drew heavily from the computational microsemantics of δ -time defined in Chapter 3. In the relational realm, the denotations of each NDAM instruction are a transition relation and the definition of a macrostep is exactly the least fixed point, relative to a single δ -step, of the image approximator functional implied by the NDAM network. There is the obvious projection to full abstraction defined by ignoring the intra-instant instruction executions and focusing directly on the states of the network as a whole at the fixed points (just when it is stable).

Based on the operational definition of the NDAM, a runtime system was defined that dynamically schedules the processes in the NDAM network in a manner which is consistent with causality and modularity. The runtime system, encapsulated in the **EVAL** algorithm, repeatedly evaluates the NDAM network until there are no more runnable processes. The inter-process coordination aspect of signals in the network revolves around the **require** instructions. These instructions guarantee that no signal is referenced before its value becomes defined, if it is ever to become defined in the macrostep. On the other hand, it is the causality analysis which guarantees that no process blocks infinitely long while awaiting the definition of a signal (such programs are disallowed because they are \overline{C}). This leaves the problem of determining, at runtime, when a signal may no longer be potentially emitted in a further δ -step. To determine when a signal is absent in the instant, the runtime system uses a set of mostly static “potential sets” (POT sets) are used to estimate whether or not a signal emission may occur. The **MARK** algorithm computes

the dynamic sum of these POT sets from the network. Signals with no potential are marked as **absent** and the network evaluation continues for another δ -step. The operational scheduler of δ -time is defined so that when no process can make further progress, then the (least relative) fixed point has been reached and the macrostep is complete.

8 Conclusion

The preceding chapters have established the general theoretical basis of the semantics of synchronous system description languages. That theory arose in answering the two questions which were posed in the beginning:

1. How can the design representation language, inclusive of both its behavioral and structural aspects, be related to the mathematical model and to what extent should the structural artifacts of the design representation be made visible in the mathematical model?
2. How should the structural aspects of the design representation language and their visibility within the mathematical model be related to the performance and the implementation of a property-checking algorithm?

Both of these questions focused on *language semantics* and the kinds of internal structure which might exist within a semantic model. In turn, structure within a model is a condition of abstraction wherein a fully abstract model preserves no structure from the original source language. When full abstraction applies, two programs denote the same model element when they compute the same thing. In contrast, in a non-abstract semantics, model elements are not necessarily canonical. There may be many model elements which are equivalent but different.

Starting from the well-known premise that a fully abstract semantics of finite-state synchronous systems is far too unwieldy to work with in practice, this dissertation investigated the conditions when a non-abstract semantics can be substituted for the fully

abstract one. The result is a theory, referred to as computational semantics in this work, which established the precise conditions when substitutability could be said to hold. This theory gave meaning to the diagram of Figure 8-1 for (L, S, M) :

- L is the set of all programs in some finite-state synchronous language (e.g. Esterel),
- $M = (Q, T)$ is a semantic model consisting of a finite set of states Q and a transition relation $T \subseteq Q \times Q$ between them,
- $S: L \rightarrow M$ is a semantic map which associates programs in L with a model in M .

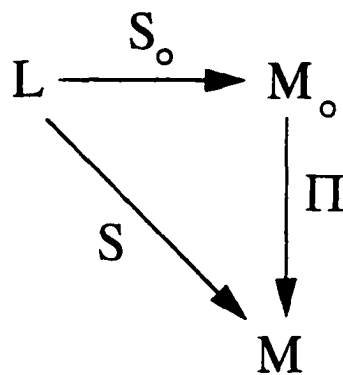


Figure 8-1. Substitutability of the Non-Abstract S_0 for the Fully Abstract S .

With the goal of substitutability in mind, two non-abstract semantics were defined and analyzed. These were the non-abstract semantics of δ -time and of σ -time which corresponded respectively to the fine structure induced onto fully abstract time by two well-understood computational schemes: the serialized execution of multiple processes on an abstract instruction-set processor and the concurrent execution of a generalized combinational gate network. The general analysis framework that was used to analyze each of these schemes was called microsemantic analysis. It consists of the eight steps which are listed in Figure 8-2. The strong result of the microsemantic analysis formulation is a definition of δ -time based on ideas from Scott's domain theory and algebraic topology. In particular, a forward macrostep is shown to be a least fixed point of the δ -time approximator

functional and a backward macrostep is its greatest fixed point. It is significant that this definition of δ -time is mathematically based and is independent of any simulator event loop.

1. Temporal Analysis
2. Domain Analysis
3. The Primitive Transition Relation T_0
4. The Primitive Image Functionals F_0 and B_0
5. The Approximator Functionals \mathcal{F}_0 and \mathcal{B}_0
6. The Approximated Image Functionals \tilde{F} and \tilde{B}
7. The Projection Π to Full Abstraction
8. Observations

Figure 8-2. The Eight Steps of Microsemantic Analysis

Unfortunately, it has been observed that there exists mathematical baggage generated by the non-abstract approach that simply cannot be removed in any projection operation. This directly impacts the utility of substitutable non-abstract semantics because the extra implementation details introduced in the non-abstract semantics are shown to affect and prevent other expressiveness properties from the fully abstract case. The observation is Huizing and Gerth's RMC Barrier Theorem which states that there can be no semantics which has all the properties of:

- Responsiveness (R): Mealy-machine behavior can be expressed.
- Modularity (M): outputs are singly assigned and unordered in microtime.
- Causality (C) microsteps are well-ordered in forward-moving time.

Thus a semantics can be crudely summarized by whether it has responsiveness (is R or \bar{R}), modularity (is M or \bar{M}) and causality (is C or \bar{C}). This led to the modification of the original substitutability diagram to the one shown in Figure 8-3 which illustrates the origin of these three elements. In particular, the new element is causality which is the

explicit ordering within a step that is introduced by the non-abstract microsemantics (e.g. by δ -time or σ -time).

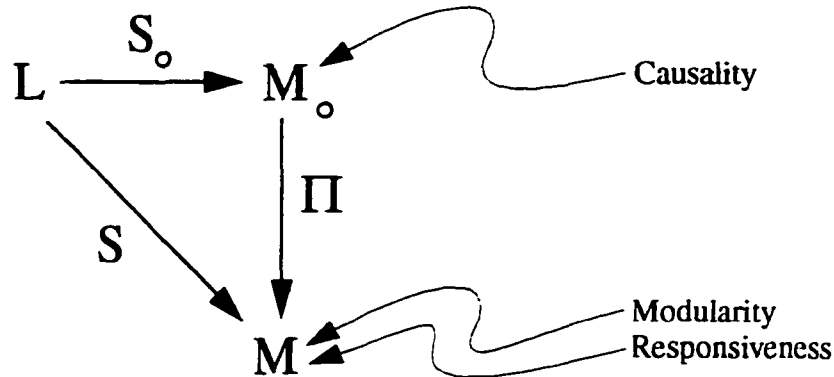


Figure 8-3. The Mathematical Baggage of the Non-Abstract S_0 .

The RMC Barrier Theorem is a powerful result for it states that no general semantic model can be *RMC*. A non-abstract semantics may be $\bar{R}MC$, $R\bar{M}C$, $RM\bar{C}$ or even $R\bar{M}\bar{C}$ and examples have been identified for all these combinations. The RMC Barrier implies that there is no clever microsemantics now extant or yet to be constructed where substitutability always applies. Fortunately, the RMC Barrier Theorem is phrased in terms of semantics, as a general class of rules for constructing systems, but not in terms of specific system descriptions. Thus, while a (micro)semantics cannot be *RMC*, a particular system instance can be *RMC*. This leads to the third element of the theory of computational semantics introduced in this dissertation: the methods for surpassing the RMC Barrier. Five methods for surpassing the RMC Barrier have been identified. These are listed in Figure 8-4.

1. The Two-of-Three Choice
2. Separated Semantics
3. Structural Restriction to *RMC*
4. Semantic Restriction to *RMC*
5. Vacated Semantics

Figure 8-4. The Methods of Surpassing the RMC Barrier

The theory of computational semantics therefore consists of three interrelated parts which can be used to understand, classify and even design language semantics. These are:

1. A microsemantic analysis.
2. A classification of the fine structure of time according to *R*, *M* and *C*.
3. The specific means used to surpass the RMC Barrier.

Using this framework a variety of semantics were surveyed. Additionally it was observed that the next obvious set of semantic extensions which might be added to a semantics (*e.g.* FRP, RCSP or the Clarke Languages) render it non-finite and even undecidable. From this survey it is clear that the Synchronous Languages, as a class, provide the best trade-off at the microsemantic level. They are $RM\bar{C}$ while recovering *C* on a per-program basis by disallowing non-causal programs at compile-time.

Additionally, a major result of this survey is the classification of the popular executable specification languages. Of significant interest are the standardized HDLs, VHDL and Verilog, which are both based on discrete event semantics (DES). The development of a mathematical understanding of these languages' semantics has long been a goal. Indeed, some concrete explanation of why they are problematic, as is the common perception, must be attained before there is any hope of repairing them or dispensing with the discrete event paradigm entirely. Merely dismissing it as mathematically intractable is not enough

given its operational convenience. Such a concrete explanatory result has been obtained with this investigation in the following sense:

- The temporal analysis of DES shows that it has a *three-level* model of time: macrotime, δ -time and η -time. This is in contrast to the popular view that DES time has two levels.
- At the macrotime level \overline{RMC} holds.
- At the δ -time level $\overline{R}_\delta M_\delta C_\delta$ holds.
- Because of $\overline{R}_\delta M_\delta C_\delta$, there exists no *general* procedure for projecting away the non-abstract details in δ -time.
- DES-based languages are not substitutable because Π cannot be defined.

This exposition makes clear the strong limitations of the discrete event approach to system specification.

In the general case DES-based languages are not substitutable, however there do exist semantically-defined subsets of DES which have the substitutability property. One such subset is the Synchronous VHDL subset which is the *RMC* subset of VHDL's \overline{RMC} . Unfortunately this subset has been shown in previous work to be highly problematic to work with because of the difficulty in modularity checking and causality checking coupled with the primitive flat process model of standard VHDL. The addition of new language features such as the hierarchical behaviors of the SpecCharts offer the best hope for salvaging the investment in training and infrastructure which has been built up around DES languages.

Finally the Nondeterministic Abstract Machine (NDAM) was introduced as an example of an internal representation designed according to the principles of computational semantics. The representation is oriented at imperative-style semantics and to that end the material form of an assembly language was used. Translation recipes for Esterel, Synchronous VHDL and the hypothetical "Synchronous SpecCharts" were described. In addition a runtime system here called the *literal fixed point method* was developed. This method has

the distinct advantage of having very small implementation size while retaining the relative speed of the older table lookup method.

To summarize, this dissertation has produced an explanation for how specification languages for mixed hardware-software systems must be interfaced to the class of problems where a precise definition of behavior is required: formal verification and eventually automated synthesis. This explanation was accomplished by making a very detailed examination of the fine structure of time within a step. In addition to the results already related, the two very important results of this analysis which stand out above all the others are:

1. The demonstration *and explanation* of why the discrete event semantics of VHDL and Verilog are fundamentally flawed and cannot be easily repaired.¹ This is the long searched-for result that shows why executable specifications must be constructed from some other stuff than a pending-transaction queue and an event loop. Non-abstract synchronous semantics are the only possible next step.
2. The exhibition that any non-abstract semantics *necessarily* has a causality checking obligation. Further, that as synchronous languages and semantics move to widespread adoption, as they surely will, that this causality checking obligation is here to stay; it is not an accidental misfeature of the original suite of synchronous languages.

1. Synchronous VHDL notwithstanding. The synchronous subset *is* difficult to program to because the base semantics is M and the programmer must mentally estimate for M .

References

- [1] M. Abadi and L. Lamport. "Composing Specifications." In *Transactions on Programming Languages and Systems*, Vol. 15, No. 1, January 1993, pages 73-132.
- [2] H. Abelson and P. Andrae. "Information Transfer and Area-Time Tradeoffs for VLSI Multiplication," In *Communications of the ACM*, Vol. 23, No. 1, 1980, pages 20-23.
- [3] S. Abramsky, "Experiments, Powerdomains and Fully Abstract Models for Applicative Multiprogramming," In *Proceedings of the International Conference on Foundations of Computation Theory*, Lecture Notes in Computer Science. Vol. 158, Springer, 1983, pages 1-13.
- [4] S. Abramsky, "Observation Equivalence as a Testing Equivalence." In *Theoretical Computer Science*, Vol. 53, 1987, pages 225-241.
- [5] S. Aggarwal, D. Barbará and K. Meth. "SPANNER: a Tool for the Specification, Analysis and Evaluation of Protocols," In *IEEE Transactions on Software Engineering*, Vol. 13, No. 12, December 1987, pages 1218-1237.
- [6] S. Aggarwal and Z. Har'El, "Simulation Analysis of Protocols in an Integrated Software Environment," In *Computer Networks and ISDN Systems*, Vol. 16, No. 3, January 1989, pages 197-215.
- [7] S. Aggarwal and R.P. Kurshan. "Automated Implementation from Formal Specification." In *Protocol Specification, Testing and Verification IV*, Y. Yemini, R. Strom and S. Yemini, editors, North-Holland, June 1984, pages 127-136.
- [8] S. Aggarwal, R.P. Kurshan, and K.K. Sabnani, "A Calculus for Protocol

- Specification and Validation.” In *Protocol Specification, Testing and Verification III*, North-Holland, 1983, pages 19-34.
- [9] G. Agha, *ACTORS: A Model of Concurrent Computation in Distributed Systems*, The MIT Press, 1988.
- [10] A. Aho, R. Sethi and J. Ullman, *Compilers - Principles, Techniques and Tools*, Addison-Wesley, 1986.
- [11] S.B. Akers, “Binary Decision Diagrams,” In *IEEE Transactions on Computers*, Vol.27, June 1978, pages 506-516.
- [12] B. Alpern and F.B. Schneider, *Defining Safety and Liveness*, Technical Report TR-85-708, Department of Computer Science, Cornell University, 1985.
- [13] B. Alpern and F.B. Schneider, “Defining Liveness,” In *Information Processing Letters*, Vol.21, October 1985, pages 181-185.
- [14] B. Alpern and F.B. Schneider, “Recognizing Safety and Liveness,” In *Distributed Computing*, Vol. 2, 1987, pages 117-126.
- [15] B. Alpern, M.N. Wegman and F.K. Zadek, “Detecting Equality of Values in Programs,” In *Proceedings of the 15th Symposium on the Principles of Programming Languages*, January 1988, pages 1-11.
- [16] T. Amon, G. Borriello and C Séquin, “Operation/Event Graphs: A Design Representation for Timing Behavior,” In *Proceedings of the 10th Conference on Computer Hardware Description Languages and their Applications*, April 1991.
- [17] T. Amon and G. Borriello, “OEsim: A Simulator for Timing Behavior,” In *Proceedings of the 28th Design Automation Conference*, June 1991, pages 565-661.
- [18] H.R. Andersen and G. Winskel, “Compositional Checking of Satisfaction,” In *Formal Methods in System Design*, Vol.1, No.4, December 1992, pages 323-354.
- [19] G.R. Andrews, “Synchronizing Resources,” In *ACM Transactions on Programming Languages and Systems*, Vol.3, No.4, October 1981, pages 405-430.
- [20] G.R. Andrews, “The Distributed Programming Language SR - Mechanisms, Design and Implementation,” In *Software Practice and Experience*, Vol.12, No.8, August

1982, pages 719-754.

[21]

G.R. Andrews and R. A. Olsson, *Revised Report on the SR Distributed Programming Language*, Technical Report TR-87-27, Department of Computer Science, University of Arizona, November, 1987.

[22]

G.R. Andrews, R.A. Olsson, M. Coffin, I.J.P. Elshoff, K. Nilsen, T. Purdin and G. Townsend, "An Overview of the SR Language and Implementation," In *ACM Transactions on Programming Languages and Systems*, Vol. 10, No. 1, Jan 1988, pages 51-86.

[23]

J.P. Ansart, P.D. Amer, V. Chari, J.F. Lenotre, L. Lumbroso, E. Mariana, and E. Mattera, "Software Tools for Estelle," In *Protocol Specification, Testing and Verification VI*, B. Sarikaya and G. van Bochmann, editors, North-Holland, June 1986, pages 55-62.

[24]

The Programming Language Ada, Reference Manual, ANSI MIL-STD-1815A, Also Lecture Notes in Computer Science, Vol. 115, Springer-Verlag, 1983.

[25]

T. Aoyagi, M. Fujita and T. Moto-oka, "Temporal Logic Programming Language e Tokio," In *Proceedings of the Logic Programming Conference '85*, Springer-Verlag, 1985, pages 128-137.

[26]

K.R. Apt and E.-R. Olderog, *Verification of Sequential and Concurrent Programs*, Springer-Verlag, 1991.

[27]

P. Ashar and M. Cheong, "Efficient Breadth-First Manipulation of Binary Decision Diagrams," In *Proceedings of the International Conference on Computer-Aided Design*, 1994, pages 622-627.

[28]

P. Ashar and S. Malik, "Implicit Computation of Minimum-Cost Feedback-Vertex Sets for Partial Scan and Other Applications," In *Proceedings of the 31st Design Automation Conference*, June 1994, pages 77-86.

[29]

E.A. Ashcroft and W.W. Wadge, "LUCID - A Formal System for Writing and Proving Programs," In *Journal of Computing*, Vol. 5, No. 3, 1976, pages 336-354.

[30]

E.A. Ashcroft and W.W. Wadge, *LUCID, the Data-Flow Programming Language*, Academic Press, 1985.

[31]

L.M. Augustin, "An Algebra of Waveforms," In *Proceedings of the IFIP*

- International Workshop on Applied Formal Methods for Correct VLSI Design*, L. Claesen editor, November 1989, pages 159-168.
- [32] L.M. Augustin, "Timing Models in VAL/VHDL," In *Proceedings of the International Conference on Computer-Aided Design*, November 1989, pages 122-125.
- [33] L.M. Augustin, B.A. Gennart, Y. Huh, D.C. Luckham and A.G. Stanculescu, "VAL: an Annotation Language for VHDL," In *Proceedings of the International Conference on Computer-Aided Design*, November 1987, pages 418-421.
- [34] L.M. Augustin, B.A. Gennart, Y. Huh, D.C. Luckham and A.G. Stanculescu, "Verification of VHDL Designs using VAL," In *Proceedings of the 25th Design Automation Conference*, June 1988, pages 48-53.
- [35] L. Augustin, D. Luckham, B. Gennart, Y. Huh and A. Stanculescu, *Hardware Design and Simulation in VAL/VHDL*, Kluwer Academic Publishers, 1991.
- [36] D. Austry and G. Boudol, "Algèbre de Processus et Synchronization," In *Theoretical Computer Science*, Vol.30, 1984, pages 91-131.
- [37] A. Aziz, F. Balarin, S.-T. Cheng, R. Hojati, T. Kam, S.C. Krishnan, R.K. Brayton, T.R. Shiple, V. Shinghal, S. Tasiran, H.-Y. Wang, R.K. Brayton and A.L. Sangiovanni-Vincentelli, "HSIS: A BDD-Based Environment for Formal Verification," In *Proceedings of the 31st Design Automation Conference*, 1994, pages 454-459.
- [38] A. Aziz and R.K. Brayton, *Verifying Interacting Finite State Machines*, Technical Report UCB/ERL M93/52, Electronics Research Laboratory, University of California, Berkeley, July 1993.
- [39] A. Aziz, T.R. Shiple, V. Singhal, R.K. Brayton and A.L. Sangiovanni-Vincentelli, *Formula-Dependent Equivalence for Compositional CTL Model Checking*, Technical Report UCB/ERL M94/??, Electronics Research Laboratory, University of California, Berkeley, 1994.
- [40] A. Aziz, T.R. Shiple, V. Singhal and A.L. Sangiovanni-Vincentelli, "Formula-Dependent Equivalence for Compositional CTL Model Checking," In *Proceedings of the Conference on Computer-Aided Verification (CAV '94)*, D.L. Dill editor, Lecture Notes in Computer Science, Vol.818, June 1994, pages 324-337.

- [41] A. Aziz, V. Singhal, G.M. Swamy and R.K. Brayton, "Minimizing Interacting Finite State Machines: A Compositional Approach to Language Containment," In *Proceedings of the International Conference on Computer Design*, October 1994, pages 255-261.
- [42] A. Aziz, S. Tasiran and R. K. Brayton, *BDD Variable Ordering for Interacting Finite Machines*, Technical Report UCB/ERL M93/71, Electronics Research Laboratory, University of California, Berkeley, September 1993.
- [43] A. Aziz, S. Tasiran and R.K. Brayton, "BDD Variable Ordering for Interacting Finite State Machines," In *Proceedings of the 31st Design Automation Conference*, 1994, pages 283-288.
- [44] P. Azema and J.C. Lloret, "ESTELLE Validation and PROLOG Interpreted Petri Nets," In *The Formal Description Technique ESTELLE*, G. Papapangiotakis, V. Verndat, M. Diaz, J.-P. Ansart, J.-P. Courtiat, P. Azema and V. Chari, editors, Elsevier Science Publishers, 1989, pages 273-302.
- [45] J. Backus, "Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs," In *Communications of the ACM*, Vol.21, No.8, August 1978, pages 613-641.
- [46] R.I. Bahar, E.A. Frohm, C.M. Gaona, G.D. Hachtel, E. Macii, A. Pardo and F. Somenzi, "Algebraic Decision Diagrams and their Applications," In *Proceedings of the International Conference on Computer-Aided Design*, November 1993, pages 188-191.
- [47] W. Baker, *An Application of a Synchronous/Reactive Semantics to the VHDL Language*, M.S. Report, Department of Electrical Engineering and Computer Science, University of California, Berkeley, January 1993, Technical Memo, UCB/ERL M93/10.
- [48] F. Balarin, *Iterative Methods for Verification of Digital Systems*, Ph.D. Thesis, University of California, Berkeley, 1994. Technical Memo UCB/ERL M94/??.
- [49] F. Balarin and G. York, "Verilog HDL Modeling Styles for Formal Verification," In *Proceedings of the IFIP Conference on Hardware Description Languages and their Applications*, April 1993, pages 439-452.
- [50] M.R. Barbacci, "A Comparison of Register-Transfer Languages," In *IEEE*

Transactions on Computers, Vol.24. 1975, pages 137-150.

- [51] M.R. Barbacci, "Instruction Set Processor (ISPS): The Notation and Its Applications," In *IEEE Transactions on Computers*, January 1981, pages 24-40.
- [52] M.R. Barbacci, S. Grout, G. Lindstrom, M.P. Maloney, E.L. Organick and D. Rudisill, "Ada as a Hardware Description Language: An Initial Report," In *Proceedings of the 7th Conference on Computer Hardware Description Languages and their Applications*, C.J. Koomen and T. Moto-Oka, editors, August 1985, pages 272-302.
- [53] H.P. Barendregt, *The Lambda Calculus - Its Syntax and Semantics*, North Holland, 1985.
- [54] H.P. Barendregt, "Functional Programming and Lambda Calculus," In *Handbook of Theoretical Computer Science* [327], Vol. B, 1990, pages 321-364.
- [55] M.R. Barbacci, G.E. Barnes, R.G. Cattel and D.P. Siewiorek, *The ISPS Computer Description Language*, Technical Report Computer Science Department, Carnegie Mellon University, August 1977.
- [56] H.P. Barendregt, "Functional Programming and Lambda Calculus," In *Handbook of Theoretical Computer Science* [327], Vol. B, 1990, pages 321-364.
- [57] H. Barringer, M.D. Fisher and G.D. Gough, "Fair SMG and Linear Time Model Checking," In *Automatic Verification Methods for Finite State Systems*, J. Sifakis editor, Lecture Notes in Computer Science, Vol.407, June 1989, pages 133-150.
- [58] D.L. Beatty, *A Methodology for Formal Verification with Application to Microprocessors*, Ph.D. Thesis, School of Computer Science, Carnegie Mellon University, August 1993.
- [59] I. Beer, S. Ben-David, D. Geist, R. Gewirtzman and M. Yoeli, "Methodology and System for Practical Formal Verification of Reactive Hardware," In *Proceedings of the Conference on Computer-Aided Verification (CAV '94)*, D. L. Dill, editor, Lecture Notes in Computer Science, Vol.818, Springer-Verlag, 1994, pages 182-193.
- [60] J.F. Beetem, "Hierarchical Topological Sorting of Apparent Loops via Partitioning," In *IEEE Transactions on Computer-Aided Design of Circuits and Systems*, Vol. 11, No.5, May 1992, pages 607-619.

- [61] M. Ben-Ari, Z. Manna and A. Pnueli, "The Temporal Logic of Branching Time," In *Proceedings of the Symposium on the Principles of Programming Languages*, 1981, pages 164-176.
- [62] A. Benveniste and G. Berry, "The Synchronous Approach to Reactive and Real-Time Systems," In *Proceedings of the IEEE*, Vol. 79, No.9, September 1991, pages 1270-1282.
- [63] A. Benveniste, P. Caspi, P. Le Guernic and N. Halbwachs, *Dataflow Synchronous Languages*, Publication Interne No.768, Institut de Recherche en Informatique et Systèmes Aléatoires, October 1993.
- [64] A. Benveniste and P. Le Guernic, *A Denotational Theory of Synchronous Communicating Systems*, Technical Report No.685, Institut National de Recherche en Informatique et en Automatique, Domaine de Voluceau, Rocquencourt, B.P. 105, 78153, Le Chesnay Cedex, France, 1987.
- [65] A. Benveniste and P. Le Guernic, "Hybrid Dynamical Systems Theory and the SIGNAL Language," In *IEEE Transactions on Automatic Control*, Vol. 35, No. 5, May 1990, pages 535-546.
- [66] A. Benveniste, P. Le Guernic, Y. Sorel and M. Sorine, "A Denotational Theory of Synchronous Communicating Systems," In *Information and Computation*, Vol.99, No.2, August 1992, pages 192-230.
- [67] J.M. Bergé, A. Fonkoua, S. Maginot and J. Rouillard, *VHDL '92*, Kluwer Academic Publishers, 1992.
- [68] J.-L. Bergerand, P. Caspi, N. Halbwachs, D. Pilaud and E. Pilaud. "Outline of a Real-Time Dataflow Language," In *Proceedings of the Real-Time Symposium*, 1985.
- [69] J.A. Bergstra and J.W. Klop, "Algebra of Communicating Processes," In J.W. de Bakker et al., editor, In *Mathematics and Computer Science*, Proceedings of the CWI Symposium, 1983; Available from North Holland, 1986.
- [70] F. Belina, D. Hogrefe and A. Sarma, *SDL with Applications from Protocol Specifications*, Prentice Hall, 1991.
- [71] R. A. Bergamaschi, A. Kuehlmann, S.-M. Wu, V. Venkataraman, D. Reischauer and D. Neumann, "A Methodology for Production Use of High-Level Synthesis," In

Proceedings of the 6th International Workshop on High-Level Synthesis, D.D. Gajski editor, November 1992.

- [72] C.L. Berman, "Ordered Binary Decision Diagrams and Circuit Structure," In *Proceedings of the International Conference on Computer Design*, October 1989, pages 392-395.
- [73] J. Bern, C. Meinel and A. Slobovodá, "Efficient OBDD-Based Manipulation in CAD Beyond Current Limits," In *Proceedings of the 32nd Design Automation Conference*, June 1995, pages 408-413.
- [74] G. Berry, "Bottom-Up Computation of Recursive Programs," In *Revue Française d'Automatique, Informatique et Recherche Opérationnelle*, Vol.10, No.3, March 1976, pages 47-82.
- [75] G. Berry, "Real Time Programming: Special Purpose or General Purpose Languages," In *Information Processing 89*, 1989, pages 11-17.
- [76] G. Berry, "A Hardware Implementation of Pure Esterel," In *International Workshop on Formal Methods in VLSI Design*, January 9-11 1991.
- [77] G. Berry, "A Hardware Implementation of Pure Esterel," In *Sadhana, Academy Proceedings in Engineering Sciences*, Vol. 17, No. 1, 1992, pages 95-130.
- [78] G. Berry, "Esterel on Hardware," In *Philosophical Transactions of the Royal Society of London*, Series A, Vol. 339, 1992, pages 87-104.
- [79] G. Berry and L. Cosserat, "The ESTEREL Synchronous Programming Language and its Mathematical Semantics," In *Seminar on Concurrency*, S.D. Brookes, A.W. Roscoe, and G. Winskel, editors, Lecture Notes in Computer Science, Vol.197, Springer-Verlag, 1984, pages 389-448.
- [80] G. Berry and P.L. Curien, "Sequential Algorithms on Concrete Data Structures," In *Theoretical Computer Science*, Vol.20, 1982, pages 265-321.
- [81] G. Berry, P. Couronné and G. Gonthier, "Synchronous Programming of Reactive Systems," In *France-Japan Artificial Intelligence and Computer Science Symposium 86*, 1986, pages 139-158.
- [82] G. Berry, S. Moison and J.P. Rigault, "Esterel: Toward a Synchronous and Semantically Sound High Level Language for Real Time Applications," In

Proceedings of the Real-Time Systems Symposium, 1983.

- [83] G. Berry, S. Ramesh and R.K. Shyamasundar, "Communicating Reactive Processes," In *Proceedings of the 20th Conference on the Principles of Programming Languages*, 1993.
- [84] G. Berry and R. Sethi, "From Regular Expressions to Deterministic Automata," In *Theoretical Computer Science*, Vol.48, Elsevier Science Publishers, 1986, pages 117-126.
- [85] C. Berthet, O. Coudert and J.-C. Madre, "New Ideas on Symbolic Manipulations of Finite State Machines," In *Proceedings of the International Conference on Computer Design*, September 1990, pages 224-227.
- [86] P. Bertin, D. Roncin and J. Vuillemin, *Introduction to Programmable Active Memories*, Technical Report PRL-3, Digital Equipment Corporation, 1989.
- [87] W.R. Bevier, W.A. Hunt, J.S. Moore and W.D. Young, "An Approach to Systems Verification," In *Journal of Automated Reasoning*, Vol.5, 1989, pages 411-428.
- [88] J.P. Billon, *Perfect Normal Forms for Discrete Functions*, BULL Research Report No.87019, 1987.
- [89] W. Billowitch, *The LOGIC_SYSTEM Package -- a Multivalued Logic System for VHDL*, IEEE Standard Logic Modeling Package, V2.300, Released October 1990.
- [90] M. Blum, A.K. Chandra and M.N. Wegman, "Equivalence of Free Boolean Graphs Can Be Decided Probabilistically Polynomial Time," In *Information Processing Letters*, Vol. 10, No.2, March 1980, pages 80-82.
- [91] G. Borriello, *A New Interface Specification Methodology and its Application to Transducer Synthesis*, Ph.D. Thesis, Computer Science Division, University of California, Berkeley, May 1988, Technical Report UCB/CSD M88/430.
- [92] D. Borrione and C. Le Faou, "Overview of the CASCADE Multilevel Hardware Description Language and its Mixed Mode Simulation Mechanisms," In *Computer Hardware Description Languages and their Applications (CHDL '85)*, 1985.
- [93] D. Borrione, L. Pierre and A. Salem, "Formal Verification of VHDL Descriptions in the Prevail Environment," In *IEEE Design and Test of Computers*, June 1992, pages 42-56.

- [94] E. Borger, U. Glasser and M. Wolfgang, "The Semantics of Behavioral VHDL '93 Descriptions," In *Proceedings of EURO-DAC '94/EURO-VHDL '94*, September 1994, pages 500-505.
- [95] S. Bose and A. Fisher, "Automatic Verification of Synchronous Circuits using Symbolic Logic Simulation and Temporal Logic," In *IMEC-IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, L.J.M. Claesen, editor, North Holland, 1989.
- [96] A. Bouajjani, J.-C. Fernandez, N. Halbwachs, P. Raymond and C. Ratel, "Minimal State Graph Generation," In *Science of Computer Programming*, Vol.18, No.3, 1992, pages 247-271.
- [97] G. Boudol, "Notes on the Algebraic Calculi of Processes," In *Logic and Models of Concurrent Systems*, K. Apt editor, NATO ASI Series F13, 1985.
- [98] G. Boudol and I. Castellani, "On the Semantics of Concurrency: Partial Orders and Transition Systems," In *Proceedings of TAPSOFT '87*, Lecture Notes in Computer Science, Vol.247, Springer-Verlag, pages 123-137.
- [99] G. Boudol, V. Roy, R. de Simone and D. Vergamini, "Process Calculi, from Theory to Practice: Verification Tools," In *Automatic Verification Methods for Finite State Systems*, J. Sifakis editor, Lecture Notes in Computer Science, Vol.407, June 1989, pages 1-10.
- [100] F. Bourdoncle, J. Vuillemin and G. Berry, *The 2Z Reference Manual*, Technical Report PRL 50, Digital Equipment Corporation, Paris Research Laboratory, 1994.
- [101] F. Boussinot and R. De Simone, "The Esterel Language," In *Proceedings of the IEEE*, Vol.79, No.9., September 1991.
- [102] R.S. Boyer and J.S. Moore, "Proof-Checking, Theorem-Proving, and Program Verification," In *Contemporary Mathematics*, Vol.29, 1984, pages 119-133.
- [103] R.S. Boyer and J.S. Moore, *A Computational Logic Handbook*, Academic Press, 1988.
- [104] R.D. Brandt, V. Garg, R. Kumar, F. Lin, S.I. Marcus and W.M. Wonham, "Formulas for Calculating Supremal Controllable and Normal Language," In *Systems & Control Letters*, No. 15, 1990, pages 111-117.

- [105] K. Brace, R. Rudell and R. Bryant, "Efficient Implementation of a BDD Package," In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, 1990, pages 40-45.
- [106] R.K. Brayton, R. Camposano, G. De Micheli, R. Otten and J. van Eijndhoven, "The Yorktown Silicon Compiler," In *Silicon Compilation*, D.D. Gajski editor, Addison Wesley, 1988, pages 204-310.
- [107] R. Brayton, M. Chiodo, R. Hojati, T. Kam, K. Kodandapani, R. Kurshan, S. Malik, A. Sangiovanni-Vincentelli, E. Sentovich, T. Shiple, K.J. Singh and H.-Y. Wang, *BLIF-MV: An Interchange Format for Design Verification and Synthesis*, Technical Report UCB/ERL M91/97, Electronics Research Laboratory, University of California, Berkeley, November 1991.
- [108] R.K. Brayton, C. McMullen, G.D. Hachtel and A.L. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publishers, 1984.
- [109] R.K. Brayton, R. Rudell, A.L. Sangiovanni-Vincentelli and A. Wang, "MIS: A Multiple-Level Logic Optimization System," In *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol.6, No.6, June 1987, pages 1062-1081.
- [110] R.K. Brayton and F. Somenzi, "Boolean Relations and the Incomplete Specification of Logic Networks." In *Proceedings of the International Conference on VLSI*, August 1989, pages 231-240.
Notes: implicit inputs and outputs in boolean relation form.
- [111] R.P. Brent and H.-T. Kung "The Chip Complexity of Binary Arithmetic." In *Proceedings of the Twelfth Annual ACM Symposium on the Theory of Computing*, 1980, pages 190-200.
- [112] R.P. Brent and H.-T. Kung, "On the Area of Binary Tree Layouts," In *Information Processing Letters*, Vol. 11, No. 1, 1980, pages 44-46.
- [113] R.P. Brent and H.-T. Kung, "The Area-Time Complexity of Binary Multiplication," In *Journal of the ACM*, Vol. 28, No. 3, 1981, pages 521-534.
- [114] E. Brinksma, "A Tutorial on LOTOS," In *Protocol Specification, Testing, and Verification V*, M. Diaz, editor, North-Holland, June 1985, pages 171-194.

- [115] A. Bronstein and C.L. Talcott, *String-Functional Semantics for Formal Verification of Synchronous Circuits*, Technical Report STAN-CS-88-1210, Computer Science Department, Stanford University, 1988.
- [116] A. Bronstein and C.L. Talcott, "Formal Verification of Synchronous Circuits based on String-Functional Semantics: The 7 Paillet Circuits in Boyer-Moore," In *Automatic Verification Methods for Finite State Systems*, J. Sifakis editor, Lecture Notes in Computer Science, Vol.407, 1989, pages317-333.
- [117] A. Bronstein and C.L. Talcott, "Formal Verification of Pipelines Based on String-Functional Semantics," In *Formal VLSI Correctness Verification, VLSI Design Methods II*, 1990, pages 349-366.
- [118] S.D. Brookes, *A Model for Communicating Sequential Processes*, Technical Report CMU-CS-83-149, Department of Computer Science, Carnegie Mellon University, 1983.
- [119] S.D. Brookes, C.A.R. Hoare and A.W. Roscoe, "A Theory of Communicating Sequential Processes," In *Journal of the ACM*, Vol.31, 1984, pages 560-599.
- [120] F.M. Brown, *Boolean Reasoning, The Logic of Boolean Equations*, Kluwer Academic Publishers, 1990.
- [121] M.C. Browne, *Automatic Verification of Finite State Machines Using Temporal Logic*, Ph.D. Thesis, School of Computer Science, Carnegie Mellon University, 1989.
- [122] M.C. Browne and E.M. Clarke, Jr., "SML - A High Level Language for the Design and Verification of Finite State Machines," Technical Report CMU-CS-85-179, Carnegie-Mellon University, 1985.
- [123] M.C. Browne and E.M. Clarke, Jr., "SML - A High Level Language for the Design and Verification of Finite State Machines," In *From HDL Descriptions to Guaranteed Correct Circuit Designs*, D. Borrione, editor, Elsevier Science Publishers, 1987, pages 269-292.
- [124] M.C. Browne, E.M. Clarke, Jr. and O. Grumberg, *Characterizing Kripke Structures in Temporal Logic*, Technical Report CS 87-104, Department of Computer Science, Carnegie Mellon University 1987.

- [125] R.E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," In *IEEE Transactions on Computers*, Vol.35, No. 8, August 1986, pages 677-691.
- [126] R.E. Bryant, "On the Complexity of VLSI Implementations and Graph Representations of Boolean Functions with Application to Integer Multiplication," In *IEEE Transactions on Computers*, Vol.40, No.2, February 1991, pages 205-213.
- [127] R.E. Bryant, "Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams," In *ACM Computing Surveys*, Vol.24, No.3, September 1992.
- [128] R.E. Bryant and Y.-A. Chen, "Verification of Arithmetic Circuits with Binary Moment Diagrams," In *Proceedings of the 32nd Design Automation Conference*, June 1995, pages 535-541.
- [129] J. Brzozowski, "A Survey of Regular Expressions and Their Applications," In *IRE Transactions on Electronic Computers*, Vol 11, 1962, pages 324-335.
- [130] J.R. Büchi, "On a Decision Method in Restricted Second Order Arithmetic," In *Z. Math Logik, Grundlag. Math.*, Vol. 6, 1960, pages 66-92.
- [131] J.R. Büchi, "On a Decision Method In Restricted Second-Order Arithmetic," In *Proceedings of the International Congress on Logic, Methodology, and Philosophy of Science*, E. Nagel editor, 1960, Stanford University Press (1962), pages 1-11.
- [132] J.R. Büchi, "The Monadic Theory of ω_1 ," In *Decidable Theories II*, Lecture Notes in Mathematics. Vol.328, Springer-Verlag, 1973, pages 1-127.
- [133] S. Budkowski and P. Dembinski, "An Introduction to ESTELLE: A Specification Language for Distributed Systems," In *Computer Networks and ISDN Systems*. Vol. 14, No. 1, 1987, pages 3-24.
- [134] J.R. Burch, "Using BDDs to Verify Multipliers," In *Proceedings of the 28th Design Automation Conference*, 1991, pages 408-412.
- [135] J.R. Burch, *Automatic Symbolic Verification of Real-Time Concurrent Systems*, Ph.D. Thesis. Carnegie-Mellon University, 1992.
- [136] J.R. Burch, E.M. Clarke, Jr., K.L. McMillan and D.L. Dill, "Sequential Circuit Verification Using Symbolic Model Checking," In *Proceedings of the 27th Design*

Automation Conference, 1990, pages 46-51.

[137]

J. Burch, E.M. Clarke, Jr., K.L. McMillan, D.L. Dill and L.J. Hwang, "Symbolic Model Checking: 10^{20} States and Beyond," In *Logic in Computer Science*, 1990.

[138]

J. Burch, E.M. Clarke, Jr. and D. Long, "Symbolic Model Checking with Partitioned Transition Relations," In *Proceedings of the IFIP TC 10/WG 10.5 Conference on Very Large Scale Integration (VLSI '91)*, 1991.

[139]

J. Burch, E.M. Clarke, Jr. and D. Long, "Representing Circuits More Efficiently in Symbolic Model Checking," In *Proceedings of the 28th Design Automation Conference*, 1991, pages 403-407.

[140]

J.R. Burch, E.M. Clarke, Jr., K.L. McMillan, D.L. Dill and J.L. Hwang, "Symbolic Model Checking: 10^{20} States and Beyond." In *Information and Computation*, Vol.28. No.2, June 1992, pages 142-170.

[141]

J.R. Burch, E.M. Clarke, D.E. Long, K.L. McMillan and D.L. Dill, "Symbolic Model Checking for Sequential Circuit Verification," In *Transactions on Computer-Aided Design of Circuits and Systems*, Vol.13, No.4, April 1994, pages 401-424.

[142]

J.R. Burch, D. Dill, E. Wolf and G. De Micheli, "Modeling Hierarchical Combinational Circuits," In *Proceedings of the International Conference on Computer-Aided Design*, November 1993, pages 612-617.

[143]

W.R. Bush, *The High-Level Synthesis of Microprocessors Using Instruction Frequency Statistics*, Ph.D. Thesis. Department of Electrical Engineering and Computer Science, University of California, Berkeley, 1992, Technical Memo UCB/ERL M92/109.

[144]

K. Butler, D. Ross, R. Kapur and M. Mercer, "Heuristics to Compute Variable Orderings for Efficient Manipulation of Ordered Binary Decision Diagrams," In *Proceedings of the 28th Design Automation Conference*, 1991, pages 458-463.

[145]

T. Bye, M.R. Lightner and D.L. Ravenscroft, "A Functional Modeling and Simulation Environment Based on ESIM and C," In *Proceedings of the International Conference on Computer-Aided Design*, November 1984, pages 51-53.

[146]

G. Cabodi and P. Camurati, "Exploiting Cofactoring for Efficient FSM Symbolic Traversal Based on the Transition Relation," In *Proceedings of the International*

Conference on Computer Design, 1993, pages 299-303.

[147]

G. Cabodi, P. Camurati and S. Quer, "Boolean Function Decomposition in Symbolic FSM Traversal." In *International Conference on VLSI and CAD (ICVC '93)*, 1993, pages 265-269.

[148]

G. Cabodi, P. Camurati and S. Quer, "Symbolic Traversals of Data Paths with Auxiliary Variables," In *GLS-VLSI '94*, 1994.

[149]

G. Cabodi, P. Camurati and S. Quer, "Auxiliary Variables for Extending Symbolic Traversal Techniques to Data Paths," In *Proceedings of the 31st Design Automation Conference*, 1994, pages 289-293.

[150]

G. Cabodi, P. Camurati, F. Corno, S. Gai, P. Prinetto and M. Sonza Reorda, "A New Model for Improving Symbolic Product Machine Traversal." In *Proceedings of the 29th Design Automation Conference*, June 1992, pages 614-619.

[151]

R.H. Campbell, A.M. Koelman and M.R. McLaughlan, "STRICT: A Design Language for Strongly Typed Recursive Integrated Circuits," In *Proceedings of the IEE*, Vol. 132, No.2, March 1985, pages 108-115.

[152]

R. Camposano, "Path-Based Scheduling for Synthesis," In *Transactions on Computer-Aided Design of Circuits and Systems*, Vol. 10, No. 1, January 1991, pages 85-93.

[153]

P. Caspi, N. Halbwachs, D. Pilaud and J.A. Plaice, "LUSTRE. A Declarative Language for Programming Synchronous Systems," In *Proceedings of the 14th Symposium on the Principles of Programming Languages*, January 1987.

[154]

L. Cardelli and R. Pike, *SQUEAK, A Language for Communicating with Mice*, Technical Report, AT&T Bell Laboratories, Murray Hill, New Jersey 07974, 1985.

[155]

J.L. Carter, B.K. Rosen, G.L. Smith and V. Pichumani, "Restricted Symbolic Evaluation is Fast and Useful," In *Proceedings of the International Conference on Computer-Aided Design*, 1989, pages 38-41.

[156]

P. Caspi, "Clocks in Data-Flow Languages," In *Theoretical Computer Science*, 1990.

[157]

CHILL Language Definition, Recommendation Z.200, Study Group XI, Consultative Committee for International Telegraphy and Telephony, 1988.

- [158] *Specification and Description Language SDL*, Recommendation Z.100, Consultative Committee for International Telegraphy and Telephony, 1988.
- [159] K.M. Chandy and J. Misra, *Parallel Program Design*, Addison-Wesley, 1988.
- [160] V. Chaiyakul and D.D. Gajski, *Assignment Decision Diagram and its Uses in High-Level Synthesis*, Technical Report #92-103, Department of Information and Computer Science, University of California, Irvine, October 1992.
- [161] V. Chaiyakul, D.D. Gajski and L. Ramachandran, "High-Level Transformations for Minimizing Syntactic Variances," In *Proceedings of the 30th Design Automation Conference*, June 1993, pages 413-418.
- [162] E. Charniak and D. McDermott. *Introduction to Artificial Intelligence*, Addison-Wesley, 1987.
- [163] C.-T. Chen, *VHDL2DDS: A VHDL to DDS Data Structure Translator*, Technical Report 91-21, Computer Engineering Division, Department of Electrical and Systems Engineering, University of Southern California, 1991.
- [164] S.T. Cheng, *Compiling Verilog into Automata*, Technical Report UCB/ERL M94/??, Electronics Research Laboratory, University of California, Berkeley, 1994.
- [165] M. Chiodo, T.R. Shiple, A. Sangiovanni-Vincentelli and R.K. Brayton, *Automatic Reduction in CTL Compositional Model Checking* Technical Memo UCB/ERL M92/55, Electronics Research Laboratory, University of California, January 1992.
- [166] M. Chiodo, T.R. Shiple, A. Sangiovanni-Vincentelli and R.K. Brayton, "Automatic Compositional Minimization in CTL Model Checking," In *Proceedings of the International Conference on Computer-Aided Design*, 1992, pages 172-178.
- [167] M. Chiodo, P. Guisto, H. Hseih, A. Jurecska, L. Lavagno and A.L. Sangiovanni-Vincentelli, "A Formal Methodology for Hardware/Software Codesign for Embedded Systems," In *IEEE Micro*, August 1994.
- [168] M. Chiodo, P. Guisto, H. Hseih, A. Jurecska, L. Lavagno and A.L. Sangiovanni-Vincentelli, *Synthesis of Software Programs from CFSM Specifications*, Technical Memo UCB/ERL M94/??, 1994.
- [169] M. Chiodo, L. Lavagno, H. Hseih, K. Suzuki, A. Sangiovanni-Vincentelli and

- E. Sentovich. "Synthesis of Software Programs for Embedded Control Applications." In *Proceedings of the 32nd Design Automation Conference*, June 1995, pages 587-592.
- [170] H. Cho, G. Hachtel, S.-W. Jeong, B. Plessier, E. Schwarz and F. Somenzi, "ATPG Aspects of FSM Verification," In *Proceedings of the International Conference on Computer-Aided Design*, 1990, pages 134-137.
- [171] Y. Choueka, "Theories of Automata on ω -Tapes: A Simplified Approach," In *Journal of Computing Systems Science*, Vol. 8, 1974, pages 117-141.
- [172] P. Chow, editor, *The MIPS-X RISC Microprocessor*, Kluwer Academic Publishers, 1989.
- [173] A. Church and J.B. Rosser. "Some Properties of Conversion," In *Transactions of the American Mathematical Society*, Vol. 39, 1936, pages 472-482.
- [174] A. Church, *The Calculi of Lambda Conversion*, Princeton University Press, 1941.
- [175] R. Cieslak and P. Varaiya, "Undecidability Results for Deterministic Communicating Sequential Processes," In *IEEE Transactions on Automatic Control*, Vol. 35, No. 9, September 1990, pages 1032-1039.
- [176] *Esterel V3 Language Reference Manual*, CIS INGENIERIE, Agence Provence Est, Les Cardoulines, B1 06560 Valbonne, France, 1990.
- [177] E.M. Clarke, Jr., "Programming Language Constructs for Which it is Impossible to Obtain Good Hoare Axiom Systems," In *Proceedings of the 5th Symposium on the Principles of Programming Languages*, 1977, pages 131-140.
- [178] E.M. Clarke, Jr., "Programming Language Constructs for Which it is Impossible to Obtain Good Hoare Axiom Systems," In *Journal of the ACM*, Vol. 26, No. 1, 1979, pages 129-147.
- [179] E.M. Clarke, Jr., "The Characterization Problem for Hoare Logic," In *Philosophical Transactions of the Royal Society of London*, Series A, Vol. 312, 1984, pages 423-440.
- [180] E.M. Clarke, Jr., "The Characterization Problem for Hoare Logic," In *Mathematical Logic and Programming Languages*, C.A.R. Hoare and J.C. Shepherdson, editors.

1985, pages 89-106.

- [181] E.M. Clarke, Jr., J.R. Burch, O. Grumberg, D.E. Long and K.L. McMillan. "Automatic Verification of Sequential Circuit Designs," In *Philosophical Transactions of the Royal Society of London, Series A*, Vol.339, No.1652, April 1992, pages 105-120.
- [182] E.M. Clarke, Jr. and I.A. Draghicescu, "Expressibility Results for Linear Time and Branching Time Logics." In *Linear Time, Branching Time and Partial Order Logics and Models for Concurrency*, Lecture Notes in Computer Science, Vol.354, Springer-Verlag, pages 428-437.
- [183] E.M. Clarke, Jr., I.A. Draghicescu and R.P. Kurshan, "A Unified Approach for Showing Language Containment and Equivalence Between Various Types of - Automata." In *Proceedings of the 15th Colloquium on Trees in Algebra and Programming*, Lecture Notes in Computer Science, Vol. 431, Springer-Verlag, May 1990.
- [184] E.M. Clarke, Jr. and E. A. Emerson, "Synthesis of Synchronization Skeletons from Branching Time Temporal Logic." In *Proceedings of Logic Programs Workshop*, Lecture Notes in Computer Science, Vol. 131, Springer-Verlag, 1981, pages 52-71.
- [185] E.M. Clarke, Jr., E. A. Emerson and A.P. Sistla, "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications," In *ACM Transactions on Programming Languages and Systems*, Vol.8, No.2, 1988, pages 244-263.
- [186] E.M. Clarke, Jr., S.M. German and J. Y. Halpern. "On Effective Axiomatizations of Hoare Logics." In *Proceedings of the 9th Symposium on the Principles of Programming Languages*, 1982, pages 309-321.
- [187] E.M. Clarke, Jr. and O. Grumberg, "Research on Automatic Verification of Finite-State Concurrent Systems," In *Annual Review of Computer Science*, Vol.2, 1987, pages 269-290.
- [188] E.M. Clarke, Jr., O. Grumberg and R.P. Kurshan, "A Synthesis of Two Approaches for Verifying Finite State Concurrent Systems," In *Logic at Totik '89, Symposium on Logical Foundations of Computer Science*, Lecture Notes in Computer Science, Vol.363, Springer-Verlag, July 1989.
- [189] E.M. Clarke, Jr., O. Grumberg and K.Hamaguchi. "Another Look at LTL Model

Checking,” In *Proceedings of the Conference on Computer-Aided Verification (CAV '94)*, D.L. Dill editor, Lecture Notes in Computer Science, Vol.818, June 1994, pages 415-427.

[190]

E.M. Clarke, Jr., O. Grumberg, H. Hirashi, S. Jha, D.E. Long, K.L. McMillan and L.A. Ness, “Verification of the Futurebus+ Cache Coherence Protocol,” In *Proceedings of the 11th International Symposium on Computer Hardware Description Languages and their Applications*, 1993.

[191]

E.M. Clarke, Jr., O. Grumberg and D.E. Long, “Model Checking and Abstraction,” In *Proceedings of the ACM Symposium on the Principles of Programming Languages*, 1992, pages 343-354.

[192]

E.M. Clarke, Jr., O. Grumberg, K.L. McMillan and X. Zhao, “Efficient Generation of Counterexamples and Witnesses in Symbolic Model Checking,” In *Proceedings of the 32nd Design Automation Conference*, June 1995, pages 427-432.

[193]

E.M. Clarke, Jr., D.E. Long and K.L. McMillan, “A Language for Compositional Specification and Verification of Finite State Hardware Controllers,” In *Proceedings of the 9th International Symposium on Hardware Description Languages and their Applications*, 1989.

[194]

E.M. Clarke, Jr., D.E. Long and K.L. McMillan, “Compositional Model Checking,” In *Proceedings of the 4th Annual Symposium on Logic in Computer Science*, 1989, pages 353-362.

[195]

E.M. Clarke, Jr., D.E. Long and K.L. McMillan, “A Language for Compositional Specification and Verification of Finite State Hardware Controllers,” In *Proceedings of the IEEE*, Vol.79, No.9, September 1991, pages 1283-1292.

[196]

E.M. Clarke, Jr., K.L. McMillan, X. Zhao, M. Fujita, and J.C.-Y. Yang, “Spectral Transforms for Large Boolean Functions with Applications to Technology Mapping,” In *Proceedings of the 30th Design Automation Conference*, June 1993, pages 54-60.

[197]

R. Cleaveland, “On Automatically Explaining Bisimulation Inequivalence,” In *Proceedings of the 2nd International Conference on Computer-Aided Verification (CAV '90)*, E. Clarke and R. Kurshan editors, Lecture Notes in Computer Science, Vol.531, Springer-Verlag, June 1990, pages 364-372.

[198]

R. Cleaveland, J. Parrow and B. Steffen, “The Concurrency Workbench,” In

Automatic Verification Methods for Finite State Systems, J. Sifakis editor, Lecture Notes in Computer Science, Vol.407, June 1989, pages 24-37.

[199]

R. Cleaveland, J. Parrow and B. Steffen, "The Concurrency Workbench." In *Transactions on Programming Languages and Systems*, Vol. 15, No. 1, January 1993, pages 36-72.

[200]

A. Coen, A. Lombardo and S. Lalazzo, "The EVA Tool: An Approach to Verify Structuring in Estelle Specifications." In *The Formal Description Technique ESTELLE*, M. Diaz, J.-P. Ansart, J.P. Courtiat, P. Azema, and V. Chari, editors, Elsevier Science Publishers, 1989 pages 303-321.

[201]

D.R. Coelho, *The VHDL Handbook*, Kluwer Academic Publishers, 1989.

[202]

D.R. Coelho and W.M. van Cleemput, "Helix: A Tool for Multilevel Simulation of VLSI Systems." In *Proceedings of the 3rd International Conference on Semi-Custom Integrated Circuits*, November 1983.

[203]

A. Cohn, "The Notion of Proof in Hardware Verification." In *Journal of Automated Reasoning*, Vol.5, 1989.

[204]

S.A. Cook, "Soundness and Completeness of an Axiom System for Program Verification." In *SIAM Journal of Computing*, Vol.7, 1978, pages 70-90.

[205]

J. Cooley, *Verilog Won & VHDL Lost? -- You Be The Judge!*, E-mail Synopsys Users Group (ESNUG), Holliston Poor Farm, P.O. Box 6222, Holliston, MA 01746-6222.

[206]

T.H. Cormen, C.E. Leiserson and R.L. Rivest, *Introduction to Algorithms*, McGraw-Hill, 1992.

[207]

L. Cosserrat, *Sémantique Opérationnelle du Langage Synchrone ESTEREL*, Thèse de Docteur Ingénieur, Université de Nice, 1985.

[208]

O. Coudert and J.-C. Madre, "A Unified Framework for the Formal Verification of Sequential Circuits." In *Proceedings of the International Conference on Computer-Aided Design*, November 1990, pages 126-129.

[209]

O. Coudert, C. Berthet and J.C. Madre, "Verification of Synchronous Sequential Machines Based on Symbolic Execution." In *Proceedings of the Workshop on Automatic Verification Methods for Finite State Systems*, J. Sifakis editor, Lecture Notes in Computer Science, Vol.407, Springer-Verlag, June 1989, pages 365-373.

- [210] O. Coudert, C. Berthet and J.C. Madre, "Verification of Sequential Machines Using Boolean Function Vectors," In *Proceedings of the IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, November 1989, pages 111-128.
- [211] O. Coudert, J.-C. Madre and C. Berthet, "Verifying Temporal Properties of Sequential Machines Without Building their State Diagrams." In *Proceedings of the 2nd International Conference on Computer-Aided Verification (CAV '90)*, E. Clarke and R. Kurshan editors, Lecture Notes in Computer Science, Vol.531, Springer-Verlag, June 1990, pages 23-32.
- [212] B. Courcelle, "Fundamental Properties on Infinite Trees," In *Theoretical Computer Science*, Vol. 25, 1983, pages 95-169.
- [213] B. Courcelle, "Recursive Applicative Program Schemes." In *Handbook of Theoretical Computer Science* [327], Vol. B, 1990, pages 459-492.
- [214] C. Courcoubetis, W. Damm and B. Josko, "Verification of Timing Properties of VHDL," In *Proceedings of the 5th International Conference on Computer-Aided Verification (CAV '93)*, C. Courcoubetis editor, Lecture Notes in Computer Science, Vol.697, Springer-Verlag, June 1993, pages 225-236.
- [215] C. Courcoubetis and M. Yannakakis, "Verifying Properties of Finite State Probabilistic Programs," In *Proceedings of the 29th Symposium on the Foundations of Computer Science*, 1988, pages 228-345.
- [216] P. Couronné. *Le Système Esterel V2*. Thèse. Université Paris VII, December 1990.
- [217] G. Cousineau, P.-L. Curien and M. Mauny, "The Categorical Abstract Machine," In *Science of Computer Programming*, Vol.8. No.2, 1987, pages 193-202.
- [218] P. Cousot and R. Cousot, "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Constructing or Approximation of Fixed Points." In *Proceedings of the 4th ACM Symposium on the Principles of Programming Languages*, 1977, pages 238-252.
- [219] P. Cousot, "Methods and Logics for Proving Programs," In *Handbook of Theoretical Computer Science* [327], Vol. B, 1990, pages 843-993.
- [220] R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman and F.K. Zadek, "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph." In

Transactions on Programming Languages and Systems, Vol.13, No.4, October 1991, pages 451-490.

[221]

W. Damm and B. Josko, "A Sound and Relatively Complete Axiomatization of Clarke's Language L_4 ," In *Proceedings of the Conference on Logics of Programs*, E.M. Clarke and D. Kozen, editors, Lecture Notes in Computer Science, Vol. 164, 1983, pages 161-175.

[222]

J.A. Darringer, "The Application of Program Verification Techniques to Hardware Verification," In *Proceedings of the 16th Design Automation Conference*, 1979, pages 375-380.

[223]

J.W. de Bakker, *Mathematical Theory of Program Correctness*, Prentice-Hall, 1980.

[224]

A. De Bruin and W. Boehn, "The Denotational Semantics of Dynamic Networks of Processes," In *Transactions on Programming Languages and Systems*, Vol.7, No.4, 1985, pages 656-679.

[225]

G. De Micheli and D.C-L. Ku, "HERCULES - A System for High-Level Synthesis," In *Proceedings of the 25th Design Automation Conference*, June 1988, pages 483-488.

[226]

R. De Nicola and M. Hennessy, "Testing Equivalences for Processes," In *Theoretical Computer Science*, Vol.34, No.83, 1984.

[227]

R. de Simone, "Higher-Level Synchronizing Devices in MEIJE-SCCS," In *Theoretical Computer Science*, Vol.37, 1985, pages 245-267.

[228]

R. de Simone and A. Ressouche, "Compositional Semantics of Esterel and Verification by Compositional Reductions," In *Proceedings of the Conference on Computer-Aided Verification (CAV '94)*, D.L. Dill editor, Lecture Notes in Computer Science, Vol.818, June 1994, pages 441-454.

[229]

VAX DECSIM Reference Manual, Digital Equipment Corporation, 1986.

[230]

P. Dembinski and S. Budkowski, "Specification Language ESTELLE," In *The Formal Description Technique ESTELLE*, M. Diaz, J.-P. Ansart, J.-P. Courtiat, P. Azema and V. Chari editors Elsevier Science Publishers, 1989, pages 35-75.

- [231] T. Despeyroux, "Executable Specification of Static Semantics," In *Proceedings of the Symposium on the Semantics of Data Types*, Lecture Notes in Computer Science, Vol. 173, Springer-Verlag, 1984.
- [232] S. Devadas, *Techniques for Optimization-Based Synthesis of Digital Systems*, Ph.D. Thesis, Department of Electrical Engineering and Computer Science, University of California, Berkeley, 1988, Technical Memo UCB/ERL M88/54.
- [233] S. Devadas, "Comparing Two-Level and Ordered Binary Decision Diagram Representations of Logic Functions," In *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 12, No.5, May 1993, pages 722-723.
- [234] S. Devadas, K. Keutzer and A.R. Newton, Personal Communication, From a tutorial presentation given at the International Conference on Computer-Aided Design, November 1989.
- [235] S. Devadas and K. Keutzer, "An Automata-Theoretic Approach to Behavioral Equivalence," In *Proceedings of the International Conference on Computer-Aided Design*, November 1990, pages 30-33.
- [236] S. Devadas, H.-K.T. Ma and A.R. Newton, "On the Verification of Sequential Machines at Differing Levels of Abstraction," In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, June 1988, pages 713-722.
- [237] E. W. Dijkstra, *A Discipline of Programming*, Prentice Hall, 1976.
- [238] D.L. Dill, *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*, Ph.D. Thesis, Department of Computer Science, Carnegie Mellon University, 1988, Technical Report CMU-CS-88-199. Also available from The MIT Press, 1989.
- [239] D. Dill, A. Drexler, A. Hu and C. Yang, "Protocol Verification as a Hardware Design Aid," In *Proceedings of the International Conference on Computer Design*, October 1992, pages 126-129.
- [240] L.K. Dillon, "Using Symbolic Execution for Verification of Ada Tasking Programs," In *Transactions on Programming Languages and Systems*, Vol. 12, No. 4, October 1990, pages 643-669.
- [241] D. Drusinsky and D. Harel, "Using StateCharts for Hardware Description," In

Proceedings of the International Conference on Computer-Aided Design, November 1987, pages 162-165.

[242]

D. Drusinsky and D. Harel, "Using Statecharts for Hardware Description and Synthesis," In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 8, No. 7, July 1989, pages 798-806.

[243]

A. Dsouza and B. Bloom, "Generating BDD Models for Process Algebra Terms," In *Proceedings of the 7th International Conference on Computer-Aided Design (CAV '95)*, P. Wolper editor, Lecture Notes in Computer Science, Vol.939, Springer-Verlag, July 1995, pages 16-30.

[244]

N. Dutt, T. Hadley and D.D. Gajski, *BIF: A Behavioral Intermediate Format for High-Level Synthesis*, Technical Report #89-03, Department of Information and Computer Science, University of California, Irvine, September 1989.

[245]

N. Dutt, T. Hadley and D.D. Gajski, "An Intermediate Representation for Behavioral Synthesis," In *Proceedings of the 27th Design Automation Conference*, 1990, pages 14-19.

[246]

S. Edwards, *An Esterel Compiler for a Synchronous/Reactive Development System*, M.S. Report, Department of Electrical Engineering and Computer Science, University of California, Berkeley, June 1994, Technical Memo UCB/ERL M94/43.

[247]

J. Ellis, *Bulldog: A Compiler for VLIW Architectures*, Ph.D. Thesis, Department of Computer Science, Yale University, Also available from The MIT Press, 1986.

[248]

E. A. Emerson, "Temporal and Modal Logic," In *Handbook of Theoretical Computer Science*, [327] Volume B, pages 996-1072.

[249]

E.A. Emerson and E.M. Clarke, Jr., "Characterizing Correctness Properties of Parallel Programs as Fixpoints," In *Proceedings of the 7th International Colloquium on Automata, Languages and Programming (ICALP)*, Lecture Notes in Computer Science, Vol.85, Springer-Verlag, 1981.

[250]

E.A. Emerson and J.Y. Halpern, "'Sometimes' and 'Not Never' revisited: on Branching versus Linear Time Temporal Logic," In *Journal of ACM*, Vol. 33, No. 1, 1986, pages 151-178.

[251]

E. A. Emerson and C.S. Jutla, "The Complexity of Tree Automata and Logics of Programs," In *Proceedings of the 28th Symposium on the Foundations of Computer*

Science, 1988, pages 328-337.

[252]

E.A. Emerson and C.S. Jutla, "On Simultaneously Determinizing and Complementing ω -Automata," In *Proceedings of the 4th Symposium on Logic on Computer Science*, 1989, pages 333-342.

[253]

E.A. Emerson and C.-L. Lei, "Modalities for Model Checking: Branching Time Logic Strikes Back," In *Proceedings of the 12th Symposium on the Principles of Programming Languages*, 1985, pages 84-96.

[254]

E.A. Emerson and C.-L. Lei, "Efficient Model Checking in Fragments of the Propositional μ -Calculus," In *Proceedings of the 2nd Symposium on Logic in Computer Science*, June 1986, pages 267-278.

[255]

E.A. Emerson and C.-L. Lei, "Temporal Reasoning under Generalized Fairness Constraints," In *Proceedings of the 3rd Symposium on Theoretical Aspects of Computer Science*, B. Monien and G. Vidal-Naquet, editors, Lecture Notes in Computer Science, Springer-Verlag, January 1986, pages 20-36.

[256]

E.A. Emerson and C.-L. Lei, "Modalities for Model Checking: Branching Time Logic Strikes Back," In *Science of Computer Programming*, Vol. 8, Elsevier Science Publishers, 1987, pages 275-306.

[257]

E. Felt, G. York, R. K. Brayton and A. Sangiovanni-Vincentelli, "Dynamic Variable Reordering for BDD Minimization," In *Proceedings of the European Design Automation Conference*, September 1993, pages 130-135.

[258]

J.-C. Fernandez, *Aldébaran. Une Système de Vérification par Réduction de Processus Communicants*. Thèse, Université de Grenoble, 1988.

[259]

J.C. Fernandez, L. Mounier, C. Jard and T. Jéron, "On-The-Fly Verification of Finite Transition Systems," In *Formal Methods in System Design*, Vol. 1, No. 1, 1992, pages 251-273.

[260]

J.C. Fernandez, A. Kerbrat and L. Mounier, "Symbolic Equivalence Checking," In *Proceedings of the Conference on Computer-Aided Verification (CAV '93)*, C. Courcoubetis editor, Lecture Notes in Computer Science, Vol. 697, 1993, pages 85-96.

[261]

J. Ferrante, K.J. Ottenstein and J.D. Warren, "The Program Dependence Graph and its Use in Optimization," In *Transactions on Programming Languages and Systems*,

Vol.9, No.3, July 1987, pages 319-349.

[262]

T. Filkorn, "Functional Extension of Symbolic Model Checking," In *Proceedings of the Conference on Computer-Aided Verification (CAV '91)*, K.G. Larsen and A. Skou, editors, Lecture Notes in Computer Science, Vol.575, Springer Verlag, pages 225-232.

[263]

R. W. Floyd, "Assigning Meanings to Programs," In *Proceedings of the Symposia in Applied Mathematics*, T. Schwartz editor, Mathematical Aspects of Computer Science American Mathematical Society, 1967, pages 19-32.

[264]

F.X. Fornari. *Optimisation du Contrôle et Implantation en Circuits de Programmes Esterel*, Thèse de Doctorat de L'Ecole des Mines de Paris, March 1995.

[265]

S. Fortune, J. Hopcroft and E.M. Schmidt, "The Complexity of Equivalence and Containment for Free Single Variable Program Schemes," In *Some Compendium*, Goos, Hartmanis, Ausiello and Böhm, editors, Lecture Notes in Computer Science, Vol.62, 1978, pages 227-240.

[266]

N. Francez, *Fairness*, Springer, 1987.

[267]

S. Friedman and K.J. Supowit, "Finding the Optimal Variable Ordering for Binary Decision Diagrams," In *IEEE Transactions on Computers*, Vol.39, No.5, May 1990, pages 710-713.

Also in *Proceedings of the 24th Design Automation Conference*, June 1987, pages 348-355.

[268]

A. Fuggetta, C. Ghezzi *et al.*, "Formal Data Flow Diagrams," In *Transactions on Software Engineering*, 1986.

[269]

H. Fujii, G. Ootomo, and C. Hori, "Interleaving Variable Ordering Methods for Ordered Binary Decision Diagrams," In *Proceedings of the International Conference on Computer-Aided Design*, November 1993, pages 38-41.

[270]

M. Fujita, H. Fujisawa and N. Kawato, "Evaluation and Improvements of Boolean Comparison Method Based on Binary Decision Diagrams," In *Proceedings of the International Conference on Computer-Aided Design*, November, 1988, pages 2-5.

[271]

F. Fujita, H. Fujisawa and Y. Matsunaga, "Variable Ordering Algorithms for Ordered Binary Decision Diagrams and their Evaluations," In *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 12, No. 1, January

1993, pages 6-12.

[272]

M. Fujita, Y. Matsunaga and T. Kakuda, "Automatic and Semiautomatic Verification of Switch-Level Circuits with Temporal Logic and Binary Decision Diagrams," In *Proceedings of the International Conference on Computer-Aided Design*, 1990, pages 38-41.

[273]

M. Fujita, Y. Matsunaga, and T. Kakuda, "On the Variable Ordering for Binary Decision Diagrams for the Application of Multilevel Logic Synthesis," In *Proceedings of the European Design Automation Conference*, February 1991, pages 50-54.

[274]

D.D. Gajski, N. Dutt, A. Wu and S. Lin, *High-Level Synthesis, Introduction to Chip and System Design*, Kluwer Academic Publishers, 1992.

[275]

D.D. Gajski, F. Vahid and S. Narayan, *SpecCharts: A VHDL Front-End for Embedded Systems*, Technical Report #93-31, Department of Information and Computer Science, University of California, Irvine, 1993.

[276]

D.D. Gajski, F. Vahid and S. Narayan, "A System Design Methodology: Executable-Specification Refinement," In *Proceedings of the European Conference on Design Automation*, 1994.

[277]

D.D. Gajski, F. Vahid, S. Narayan and J. Gong, *Specification and Design of Embedded Systems*, Prentice Hall, 1994.

[278]

M.R. Garey and D.S. Johnson, *Computers and Intractability, A Guide to the Theory of NP-Completeness*, W.H. Freeman and Company, 1979.

[279]

S. Garfinkel, D. Weise and S. Strassman, *The Unix-Haters Handbook*, IDG Books, 1993.

[280]

T. Gauthier, P. Le Guernic and L. Bernard, "Signal, a Declarative Language for Synchronous Programming of Real-Time Systems," In *Proceedings of the 3rd Conference on Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science, Vol.274, 1987.

[281]

D. Geist and I. Beer, "Efficient Model Checking by Automated Ordering of Transition Relation Partitions," In *Proceedings of the Conference on Computer-Aided Verification (CAV '94)*, Lecture Notes in Computer Science, Vol.818, Springer-Verlag, June 1994, pages 299-310.

- [282] B.A. Gennart, *Automated Analysis of Discrete Event Simulations Using Event Pattern Mappings*, Ph.D. Thesis, Stanford University, April 1991, Technical Report CSL-TR-91-464.
- [283] G.A. Gennart and D.C. Luckham, "Validating Discrete Event Simulations Using Event Pattern Mappings," In *Proceedings of the 29th Design Automation Conference*, June 1992, pages 414-419.
- [284] G.W. Gerber, *Une Methode d'Implantation Automatique de Systèmes Specifies Formellement*, Masters Report, University of Montreal, 1983.
- [285] R. Gerth and W.P. de Roever, "A Proof System for Concurrent Ada Programs," In *Science of Computer Programming*, Vol.4, Elsevier Science, 1984, pages 159-204.
- [286] E.F. Girczyc, Automatic Generation of Microsequenced Data Paths to Realize Ada Circuit Descriptions, Ph.D. Thesis, Department of Electronics, Faculty of Engineering, Carleton University, July 1984.
- [287] E.F. Girczyc and J.P. Knight, "An Ada to Standard Cell Hardware Compiler Based on Graph Grammars and Scheduling," In *Proceedings of the International Conference on Computer Design*, October 1984, pages 726-731.
- [288] E.F. Girczyc, R.J.A. Buhr, J.P. Knight, "Applicability of a Subset of Ada as an Algorithmic Hardware Description Language for Graph-Based Hardware Compilation," In *Transactions on Computer-Aided Design of Circuits and Systems*, Vol.4, No.2, April 1985, pages 134-142.
- [289] R.S. Glanville, *A Machine-Independent Algorithm for Code Generation and Its Use In Retargetable Compilers*, Ph.D. Thesis, Computer Science Division, University of California, Berkeley, December 1977, Technical Report CS-78-01.
- [290] R.S. Glanville and S.L. Graham, "A New Method for Compiler Code Generation," In *Proceedings of the 5th Symposium on Principles of Programming Languages*, January 1978, pages 509-514.
- [291] A.-C. Glory, *Vérification de Propriétés de Programmes Flots de Données*, Thèse, Université Joseph Fourier, Grenoble, 1989.
- [292] P. Godefroid, "Using Partial Orders to Improve Automatic Verification Methods," In *Proceedings of the Conference on Computer-Aided Verification (CAV '90)*,

- E.M. Clarke and R.P. Kurshan, editors, *Lecture Notes in Computer Science*, Vol.531, Springer-Verlag, 1990, pages 176-185.
- [293] P. Godefroid, G.J. Holzmann and D. Porottin, "State Space Caching Revisited," In *Proceedings of the Conference on Computer-Aided Verification (CAV '92)*, G. van Bochmann, D.K. Probst, editors, *Lecture Notes in Computer Science*, Vol. 663, 1992, Springer-Verlag, pages 178-191.
- [294] B. Gonipath and R.P. Kurshan, *The Selection/Resolution Model of Coordinating and Concurrent Processes*, Technical Report, AT&T Bell Laboratories, Murray Hill NJ, 1980.
- [295] G. Gonthier, *Sémantiques et Modèles d'Exécution des Langages Réactifs Synchrones: Application à Esterel*, Thèse, Université de Paris-Sud Centre d'Orsay, 1988.
- [296] D.I. Good, R.M. Cohen and L.W. Hunter, *A Report on the Development of Gypsy*, Certifiable Minicomputer Project Report ICSCA-CMP-13, Institute for Computing Science and Computer Applications, University of Texas at Austin, October 1978.
- [297] D.I. Good, R.M. Cohen and J. Keeton-Williams, *Principles of Proving Concurrent Programs in Gypsy*, Certifiable Minicomputer Project Report ICSCA-CMP-15, The Institute for Computing Science and Computer Applications, University of Texas at Austin, January 1979.
- [298] M.J.C. Gordon, "The Denotational Semantics of Sequential Machines," In *Information Processing Letters*, Vol. 10, No. 1, February 1980, pages 1-3.
- [299] M.J.C. Gordon, "Why Higher-Order Logic is a Good Formalism for Specifying and Verifying Hardware," In *Formal Aspects of VLSI Design*, 1986.
- [300] M.J.C. Gordon, *Programming Language Theory and Its Implementation: Applicative and Imperative Programs*, Prentice Hall, 1988.
- [301] S. Graf and B. Steffen, "Compositional Minimization of Finite State Systems," In *Proceedings of the Conference on Computer-Aided Verification (CAV '90)*, E.M. Clarke and R.P. Kurshan, editors, *Lecture Notes in Computer Science*, Vol.531, Springer-Verlag, 1990, pages 186-196.
- [302] S.L. Graham and R.S. Glanville, "The Use of a Machine Description for Compiler Code Generation," In *Proceedings of the 3rd Jerusalem Conference on Information*

Technology, August 1978, pages 508-513.

- [303] S. Greibach, "Remarks on Blind and Partially Blind One-Way Multi-Counter Machines," In *Theoretical Computer Science*, Vol.7, 1978, pages 310-324.
- [304] C.A. Guimale and H.J. Kahn, "An Information Model of Time," In *Proceedings of the 30th Design Automation Conference*, June 1993, pages 668-672.
- [305] C.A. Guimale and H.J. Kahn, "Information Models of VHDL," In *Proceedings of the 32nd Design Automation Conference*, June 1995, pages 678-683.
- [306] O. Grumberg and D.E. Long, "Model Checking and Modular Verification," In *Proceedings of the 2nd International Conference on Concurrency (CONCUR '91)*, J.C.M. Baeten and J.F. Groote, editors, Lecture Notes in Computer Science, V.527, August 1991.
- [307] O. Grumberg and D.E. Long, "Model Checking and Modular Verification," In *Transactions on Programming Languages and Systems*, Vol. 16, No. 3, May 1994, pages 843-871.
- [308] J. Gundawardena, "Causal Automata," In *Theoretical Computer Science*, Vol.101, No.2, 1992, pages 265-299.
- [309] C.A. Gunter and D.S. Scott, "Semantic Domains," In *Handbook of Theoretical Computer Science* [327], Vol.B, 1990, pages 635-675.
- [310] A. Gupta, "Formal Hardware Verification Methods: A Survey," In *Formal Methods in System Design*, Vol. 1, No.2/3, Kluwer Academic Publishers, New York, October 1992, pages 151-238.
- [311] A. Gupta and A.L. Fisher, *Representation and Manipulation of Inductive Boolean Functions*, Technical Report CMU-CS-92-129, School of Computer Science, Carnegie Mellon University, April 1992.
- [312] A. Gupta and A.L. Fisher, "Parametric Circuit Representation Using Inductive Boolean Functions," In *Proceedings of the Conference on Computer-Aided Verification*, June 1993.
- [313] A. Gupta and A.L. Fisher, "Representation and Symbolic Manipulation of Linearly Inductive Boolean Functions," In *Proceedings of the International Conference on*

Computer-Aided Design, November 1993, pages 192-199.

- [314] A.P. Gupta and D.P. Siewiorek, "Automated Multi-Cycle Symbolic Timing Verification of Microprocessor-based Designs," In *Proceedings of the 31st Design Automation Conference*, June 1994, pages 113-119.
- [315] G.D. Hachtel and M.R. Lightner, "Don't Care Conditions in Top-Down Synthesis," In *Proceedings of the International Conference on Computer-Aided Design*, 1987, pages 316-319.
- [316] G.D. Hachtel, E. Macii, A. Pardo and F. Somenzi, "Symbolic Algorithms to Calculate Steady-State Probabilities of a Finite State Machine," In *Proceedings of the European Design Automation Conference (EDAC '94)*, February 1994, pages 214-218.
- [317] G.D. Hachtel, E. Macii, A. Pardo and F. Somenzi, "Probabilistic Analysis of Large Finite State Machines," In *Proceedings of the 31st Design Automation Conference*, 1994, pages 270-275.
- [318] L.J. Hafer, *Automated Data-Memory Synthesis: A Formal Model for the Specification, Analysis and Design of Register-Transfer Level Digital Logic*, Ph.D. Thesis, Department of Electrical Engineering, Carnegie Mellon University, May 1981.
- [319] L.J. Hafer and A.C. Parker, "A Formal Method for the Specification Analysis and Design of Register-Transfer Level Digital Logic," In *Transactions on Computer-Aided Design of Circuits and Systems*, January 1983, pages 4-17.
- [320] N. Halbwachs, *Synchronous Programming of Reactive Systems*, Kluwer Academic Publishers, 1993.
- [321] H. Halbwachs, P. Caspi, P. Raymond and D. Pilaud, "The Synchronous Dataflow Language LUSTRE," In *Proceedings of the IEEE*, Vol.79, No.9, September 1991, pages 1305-1320.
- [322] N. Halbwachs, D. Pilaud, F. Ouabdesselam and A.-C. Glory, "Specifying Programming and Verifying Real-Time Systems Using Synchronous Declarative Language," In *Automatic Verification Methods for Finite State Systems*, J. Sifakis editor, Lecture Notes in Computer Science, Vol.407, June 1989, pages 213-231.
- [323] N. Halbwachs, F. Lagnier and C. Ratel, "An Experience in Proving Regular

Networks of Processes By Modular Model Checking," In *Acta Informatica*, Vol.29, No.6/7, 1992.

- [324] P.R. Halmos, *Naive Set Theory*, Van Nostrand Co., 1960.
- [325] P.R. Halmos, *Lectures on Boolean Algebras*, Springer-Verlag, 1974.
- [326] J. Halpern, Z. Manna and B. Moszkowski, "A Hardware Semantics Based on Temporal Intervals," In *Proceedings of the 10th International Colloquium on Automata, Languages and Programming (ICALP '83)*, Springer-Verlag, 1983, pages 278-291.
- [327] *Handbook of Theoretical Computer Science*, J. van Leeuwen, managing editor, The MIT Press/Elsevier, 1990.
- [328] J. Hannan, "Operational Semantics Directed Compilers and Machine Architectures," In *Transactions on Programming Languages and Systems*, Vol. 16, No.4, July 1994, pages 1215-1247.
- [329] J. Hannan and D. Miller, "From Operational Semantics to Abstract Machines," In *Mathematical Structures of Computer Science*, Vol.2, No.4, 1992, pages 412-459.
- [330] D. Harel, "STATECHARTS: a Visual Formalism for Complex Systems," In *Science of Computer Programming*, Vol.8, North-Holland, 1987, pages 231-274.
- [331] D. Harel "On Visual Formalisms," In *Communications of the ACM*, Vol.31, No.5, May 1988, pages 514-531.
- [332] D. Harel and A. Pnueli, "On the Development of Reactive Systems," In *Logic and Models of Concurrent Systems*, K.R. Apt editor, NATO Advanced Study Institute on Logics and Models of Verification and Specification of Concurrent Systems, Springer-Verlag, 1985, pages 477-498.
- [333] D. Harel, A. Pnueli, J. Pruzan-Schmidt and R. Sherman, "On the Formal Semantics of Statecharts, In *Proceedings of the Symposium on Logic in Computer Science*, 1987, pages 54-64.
- [334] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring and M. Trakhtenbrot, "STATEMATE: A Working Environment for the Development of Complex Reactive Systems," In *IEEE Transactions on Software Engineering*, Vol. 16, No. 4, April 1990, pages 403-414.

- [335] D. Harel and C. Kahana, "On Statecharts with Overlapping ,," In *ACM Transactions on Software Engineering and Methodology*, Vol. 1, No. 4, October 1992, pages 399-421.
- [336] D. Harel, D. Kozen and R. Parikh, "Process Logic: Expressiveness, Decidability, Completeness," In *Proceedings of the 21st Symposium on the Foundations of Computer Science*, October 1980, pages 129-142.
- [337] Z. Har'El and R.P. Kurshan, "Software for Analytical Development of Communication Protocols," In *AT&T Technical Journal*, AT&T Bell Laboratories, Murray Hill NJ, January 1990.
- [338] S. Hayashi and H. Nakano, *PX: A Computational Logic*, The MIT Press, 1988.
- [339] J. Helbig and P. Kelb, "An OBDD Representation of StateCharts," In *Proceedings of the European Design Automation Conference (EDAC)*, 1994.
- [340] J. Helbig, R. Schör, W. Damm, G. Döhmen and P. Kelb, "VHDL/S - Integrating Statecharts, Timing Diagrams and VHDL," In *Microprocessing and Microprogramming*, Vol.38, 1993, pages 571-580.
- [341] M.C.B. Hennessy and R. Milner, "Algebraic Laws for Nondeterminism and Concurrency," In *Journal of the ACM*, Vol.32, No. 1, 1985, pages 137-161.
- [342] M.C.B. Hennessy and G.D. Plotkin, "Full Abstraction for a Simple Parallel Programming Language," In *Proceedings of the Symposium on the Mathematical Foundations of Computer Science*, Lecture Notes in Computer Science, Vol.74, Springer 1979, pages 108-121.
- [343] M. Hennessy, *Algebraic Theory of Processes*, The MIT Press, 1986.
- [344] J.L. Hennessy and D.A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufman, 1990.
- [345] R.R. Henry, *Graham-Glanville Code Generators*, Ph.D. Thesis, Computer Science Division, University of California, Berkeley, May 1984, Technical Report UCB/CSD 84/184.
- [346] K. Hensel, "Zalentheorie," In *Goshen..*, Berlin, 1913; as cited in [710].

- [347] M. Heymann, "Concurrency and Discrete Event Control," In *Control Systems Magazine*, Vol. 10, No. 4, June 1990, pages 103-112.
- [348] P.N. Hilfinger, *Abstraction Mechanisms and Language Design*, Ph.D. Thesis, Carnegie Mellon University, 1983, Also available from The MIT Press, 1986.
- [349] P.N. Hilfinger, "A High-Level Language and Silicon Compiler for Digital Signal Processing," In *Proceedings of the Custom Integrated Circuits Conference*, 1985.
- [350] P.N. Hilfinger and J.A. Rabaey, *Anatomy of a Silicon Compiler*, Kluwer Academic Publishers 1992.
- [351] D.D. Hill, "ADLIB: A Modular Strongly-Typed Computer Design Language," In *Proceedings of the 4th International Symposium on Computer Hardware Description Languages and their Applications*, 1979, pages 75-81.
- [352] C.A.R. Hoare. "An Axiomatic Basis for Computer Programming," In *Communications of the ACM*, Vol. 12, October 1969, pages 576-583.
- [353] C.A.R. Hoare. "Communicating Sequential Processes." In *Communications of the ACM*, Vol. 21, No. 8, 1978, pages 666-677.
- [354] C.A.R. Hoare. *A Model for Communicating Sequential Processes*, Technical Report PRG-22, Programming Research Group, Oxford University Computing Laboratory, 1981.
- [355] C.A.R. Hoare. *Communicating Sequential Processes*, Prentice Hall, 1985.
- [356] C.A.R. Hoare and P. Lauer, "Consistent and Complementary Formal Theories of the Semantics of Programming Languages," In *Acta Informatica*, Vol. 3, 1984, pages 135-155.
- [357] R. Hojati and R.K. Brayton, "Automatic Datapath Abstraction in Hardware Systems," In *Proceedings of the 7th International Conference on Computer-Aided Design (CAV '95)*, P. Wolper editor, Lecture Notes in Computer Science, Vol. 939, July 1995, pages 98-113.
- [358] R. Hojati and R.K. Brayton, "An Environment for Formal Verification Based on Symbolic Computations," In *Journal of Formal Methods in System Design*, Vol. 6, 1995, pages 191-216.

- [359] R. Hojati, R.K. Brayton and R.P. Kurshan, "BDD-Based Debugging of Designs Using Language Containment and Fair CTL," In *Proceedings of the Conference on Computer-Aided Verification (CAV '93)*, C. Courcoubetis editor, Lecture Notes in Computer Science, Vol. 697, Springer-Verlag, June 1993, pages 41-58.
- [360] R. Hojati, S. Krishnan and R.K. Brayton, *Heuristic Algorithms for Early Quantification and Partial Product Minimization*, Technical Report UCB/ERL M94/11, Electronics Research Laboratory, University of California, Berkeley, 1994.
- [361] R. Hojati, R. Mueller-Thuns and R.K. Brayton, "Improving Language Containment Using Fairness Graphs," In *Proceedings of the Conference on Computer-Aided Verification (CAV '94)*, D. L. Dill, editor, Lecture Notes in Computer Science, Vol. 818, Springer-Verlag, 1994, pages 391-403.
- [362] R. Hojati, T.P. Shiple, R.K. Brayton and R.P. Kurshan, "A Unified Approach to Language Containment and Fair CTL Model Checking," In *Proceedings of the 30th Design Automation Conference*, June 1993, pages 475-481.
- [363] R. Hojati, V. Singhal and R.K. Brayton, *Edge-Street/Edge-Rabin Automata Environment for Formal Verification Using Language Containment*, Technical Report M94/12, Electronics Research Laboratory, University of California, Berkeley, 1994.
- [364] R. Hojati, H. Touati, R. P. Kurshan and R. K. Brayton, "Efficient ω -Regular Language Containment," In *Proceedings of Conference on Computer-Aided Verification (CAV '92)*, 1992, pages 371-382.
- [365] G.J. Holzmann, "A Theory for Protocol Validation," In *IEEE Transactions on Computers*, Vol. 31, August 1982, pages 730-738.
- [366] G.J. Holzmann, *Automated Protocol Validation in Argos, Assertion Proving and Scatter Searching*, Computer Science Press, 1987, pages 163-188.
- [367] J.J.M. Hooman, S. Ramesh and W.P. de Roever, "A Compositional Axiomatization of StateCharts," In *Theoretical Computer Science*, Vol. 101, 1992, pages 289-335.
- [368] J. Hopcroft and J. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, 1979.

- [369] T. Hoshino, O. Karatsu and T. Nakashima, "HSL-FX: A Unified Language for VLSI Design," In *Proceedings of the 7th Symposium on Computer Hardware Description Languages and Their Applications*, August 1985.
- [370] A.J. Hu, D.L. Dill, A.J. Drexler and C.H. Yang, "Higher-Level Specification and Verification with BDDs," In *Proceedings of the Conference on Computer-Aided Verification (CAV '92)*, July 1992, pages 82-95.
- [371] A.J. Hu and D.L. Dill, "Reducing BDD Size by Exploiting Functional Dependencies," In *Proceedings of the 30th Design Automation Conference*, June 1993, pages 266-271.
- [372] A.J. Hu and D.L. Dill, "Efficient Verification with BDDs Using Implicitly Conjoined Invariants," In *Proceedings of the Conference on Computer-Aided Verification (CAV '93)*, C. Courcoubetis editor, Lecture Notes in Computer Science, Vol.697, Springer-Verlag, 1993.
- [373] A.J. Hu, G. York and D.L. Dill, "New Techniques for Efficient Verification with Implicitly-Conjoined BDDs," In *Proceedings of the 31st Design Automation Conference*, 1994, pages 276-282.
- [374] S.C.-Y. Huang and W.H. Wolf, "Scheduling for Minimum Dependence in FSMs." In *Proceedings of the International Conference on Computer-Aided Design*, November 1993, pages 446-449.
- [375] G.E. Hughes and M.J. Creswell, *An Introduction to Modal Logic*. Methuen (London), 1977.
- [376] C. Huizing, R. Gerth and W.P. de Roever, "Modelling StateCharts Behavior in a Fully Abstract Way," In *Proceedings of the 13th Colloquium on Trees in Algebra and Programming*, Lecture Notes in Computer Science, Vol.299, 1988, pages 271-294.
- [377] C. Huizing, *Semantics of Reactive Systems: Comparison and Full Abstraction*. Ph.D. Thesis, Technical University Eindhoven, 1991.
- [378] C. Huizing and R. Gerth, "Semantics of Reactive Systems in Abstract Time." In *Real-Time: Theory in Practice*. J.W. de Bakker, C. Huizing, W.P. de Roever, and G. Rozenberg, editors, Proceedings of the REX Workshop, June 1991, pages 291-314.

- [379] W. Hunt, "Micro-processor Design Verification." In *Journal of Automated Reasoning*, Vol.5, 1989.
- [380] S.H. Hwang and A.R. Newton, "An Efficient Design Correctness Checker of Finite State Machines," In *Proceedings of the International Conference on Computer-Aided Design*, November 1987, pages 410-413.
- [381] S.H. Hwang, *Multilevel Behavioral Verification for VLSI Design*, Ph.D. Thesis, Department of Electrical Engineering and Computer Science, University of California, Berkeley, 1989, Technical Memo UCB/ERL M89/85.
- [382] S.H. Hwang and A.R. Newton, "An Efficient Verifier for Finite State Machines," In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol.10, No.3, March 1991, pages 326-334.
- [383] "The Semantics of StateCharts," In *The StateMate System System Documentation*, i-Logix Inc., 1989.
- [384] *IEEE Standard VHDL Language Reference Manual*, The Institute of Electrical and Electronics Engineers Inc., 345 East 47th Street, New York NY 10017, 1987, IEEE Std 1076-1987.
- [385] *The Sense of the VASG on IEEE Std 1076-1987, IEEE Standard VHDL Language Reference Manua*, The Institute of Electrical and Electronics Engineers Inc., 345 East 47th Street, New York NY 10017, October 1990, IEEE 1076-CONC-1990.
- [386] *IEEE Standards Interpretations of IEEE Std 1076-1987, IEEE Standard VHDL Language Reference Manual*, The Institute of Electrical and Electronics Engineers Inc., 345 East 47th Street, New York NY 10017, 1991, IEEE 1076-INT-1991.
- [387] *IEEE Standard VHDL Language Reference Manual*, The Institute of Electrical and Electronics Engineers Inc., 345 East 47th Street, New York NY 10017, 1993, IEEE Std 1076-1993.
- [388] K. Inan and P. Varaiya, "Finitely Recursive Process Models for Discrete Event Systems," In *IEEE Transactions on Automatic Control*, Vol.33, No.7, July 1988, pages 626-639.
- [389] Inmos Ltd. *OCCAM Programming Manual*, Prentice Hall International, 1984.

- [390] *iHDL: Intel Hardware Description Language Reference Manual*, Intel Corporation, 1990.
- [391] C.N. Ip and D.L. Dill, "Better Verification Through Symmetry," In *Proceedings of the 11th Symposium on Computer Hardware Description Languages and their Applications*, April 1993, pages 87-100.
- [392] C.N. Ip and D.L. Dill, "Efficient Verification of Symmetric Concurrent Systems." In *Proceedings of the International Conference on Computer Design*, October 1993.
- [393] N. Ishiura, Y. Deguchi and S. Yajima, "Coded Time-Symbolic Simulation Using Shared Binary Decision Diagrams." In *Proceedings of the 27th Design Automation Conference*, 1990, pages 130-135.
- [394] N. Ishiura, H. Sawada and S. Yajima, "Minimization of Binary Decision Diagrams Based on Exchanges of Variables," In *Proceedings of the International Conference on Computer-Aided Design*, November 1991, pages 472-475.
- [395] N. Ishiura, H. Yasuura and S. Yajima, "NES: The Behavioral Model for the Formal Semantics of a Hardware Design Language UDL/I," In *Proceedings of the 27th Design Automation Conference*, June 1990, pages 8-13.
- [396] *Estelle - A Formal Description Technique Based on an Extended State Transition Model*, International Organization for Standardization, Draft International Standard 9074, International Organization for Standardization, Information Processing Systems, Open Systems Interconnection, Geneva, 1987.
- [397] *LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behavior*, Draft International Standard 8807, International Organization for Standardization, Information Processing Systems, Open Systems Interconnection, Geneva, July 1987.
- [398] *EXPRESS Language Reference Manual*, International Standards Organization, ISO TC184/SC4/WG5 ISO10303, Part 11, Version N14, April 1991.
- [399] L.B. Jackson, *Digital Filters and Signal Processing*, Kluwer Academic Publishers, 1989, second edition.
- [400] L.J. Jagadeesan, C. Puchol and J.E. Von Olnhausen, "Safety Property Verification of Esterel Programs and Applications to Telecommunications," In *Proceedings of the*

7th Conference on Computer-Aided Verification, (CAV '95), 1995.

- [401] L.J. Jagadeesan, C. Puchol and J.E. Von Olnhausen, "A Formal Approach to Reactive Systems Software: A Telecommunications Application in Esterel," In *Proceedings on Industrial-Strength Formal Specification Techniques*, April 1995.
- [402] J. Jain, M. Abadir, J. Bitner, D.S. Fussell and J.A. Abraham, "TBDDs: an Efficient Functional Representation for Digital Circuits," In *Proceedings of the European Design Automation Conference (EDAC '92)*, 1992, pages 440-446.
- [403] J.J. Jain, J.A. Abraham, J. Bitner and D.S. Fussell, "Probabilistic Verification of Boolean Functions," In *Formal Methods in System Design*, Vol. 1, No. 1, July 1992, pages 61-115.
- [404] *UDLI Language Reference*, Japan Electronic Industry Development Association, 1990, English version.
- [405] S.-W. Jeong, B. Plessier, G. Hachtel and F. Somenzi, "Variable Ordering for FSM Traversal," In *Proceedings of the International Conference on Computer-Aided Design*. November 1991, pages 476-479.
- [406] S.-W. Jeong, B. Plessier, G. Hachtel and F. Somenzi, "Extended BDDs: Trading of Canonicity for Structure in Verification Algorithms," In *Proceedings of the International Conference on Computer-Aided Design*, November 1991, pages 464-467.
- [407] S.D. Johnson, *Synthesis of Digital Designs from Recursion Equations*, Ph.D. Thesis, Indiana University, 1983, Also, The MIT Press, 1984.
- [408] S.D. Johnson, B. Bose and C. David, "A Tactical Framework for Hardware Design," In *VLSI Specification, Verification and Synthesis*, G. Birtwistle and P.A. Subrahmanyam, editors, Kluwer Academic Publishers, 1987, pages 349-383.
- [409] G. Jones and M. Goldsmith, *Programming in OCCAM2*, Prentice Hall, 19??.
- [410] N.D. Jones and S.S. Muchnick, "Even Simple Programs are Hard to Analyze," In *Journal of the ACM*, Vol.24, No.2, 1977, pages 338-350.
- [411] N.D. Jones and S.S. Muchnick, "Complexity of Finite Memory Programs with Recursion," In *Journal of the ACM*, Vol.5, No.4, 1978, pages 312-321.

- [412] H.-P. Juan, V. Chaiyakul and D.D. Gajski, *Condition Graphs for High-Quality Behavioral Synthesis*, Technical Report #94-32, Department of Information and Computer Science, University of California, Irvine, 1994..
- [413] H.-P. Juan, V. Chaiyakul and D.D. Gajski, "Condition Graphs for High-Quality Behavioral Synthesis," In *Proceedings of the International Conference on Computer-Aided Design*, November 1994, pages 170-174.
- [414] G. Kahn, "Semantics of a Simple Language for Parallel Programming," In *Proceedings of the IFIP Congress 74*, J.L. Rosenfeld editor, North Holland, 1974, pages 471-475.
- [415] G. Kahn and D.B. Mac Queen "Coroutines and Networks of Parallel Processes," In *Proceedings of the IFIP Congress 77*, B. Gilchrist editor, 1977, pages 993-998.
- [416] G. Kahn, "Natural Semantics," In *Proceedings of the Symposium on Theoretical Aspects of Computer Science (STACS '87)*, Springer-Verlag, 1987, pages 22-39.
- [417] T. Y. Kam and R. K. Brayton, *Multi-Valued Decision Diagrams*, Technical Report UCB/ERL M90/125, Electronics Research Laboratory, University of California, December 1990.
- [418] O. Karatsu "VLSI Design Language Standardization Effort in Japan," In *Proceedings of the 26th Design Automation Conference*, June 1989, pages 50-55.
- [419] R. Karp and R. Miller, "Parallel Program Schemata," In *Proceedings of the 1967 Conference on Switching and Automata Theory*, October 1967, pages 55-61. Technical report version [420], journal version [421].
- [420] R. Karp and R. Miller, *Parallel Program Schemata*, Technical Report RC-2053, IBM T.J. Watson Research Center, April 1968, 54 pages.
- [421] R. Karp and R. Miller, "Parallel Program Schemata," In *Journal of Computer and System Science*, Vol.3, No.4, May 1969, pages 167-195.
- [422] K. Karplus, *Representing Boolean Functions with If-Then-Else DAGs*, Technical Report UCSC-CRL-88-28, Computer Engineering, University of California, Santa Cruz, December 1988.

- [423] K. Karplus, "Using If-Then-Else DAGs for Multi-Level Logic Minimization." In *Advanced Research in VLSI: Proceedings of the Decennial Caltech Conference on VLSI*, March 1989, pages 101-118.
- [424] J. Katzenelson and R.P. Kurshan, "S/R: A Language for Specifying Protocols and Other Coordinating Processes," In *Proceedings of the 5th Annual International Phoenix Conference on Computer Communications*, IEEE, 1986, pages 286-292.
- [425] K.M. Kavi, B.P. Buckles, U.N. Bhat, "A Formal Definition of Data Flow Graph Models," In *IEEE Transactions on Computers*, Vol35, No.11, November 1986, pages 940-948.
- [426] U. Keschull, E. Schubert and W. Rosenstiel, "Multilevel Logic Based on Functional Decision Diagrams," In *Proceedings of the European Design Automation Conference*, 1992, pages 43-47.
- [427] J.L. Kelley, *General Topology*, Springer-Verlag, 1955.
- [428] R. Keller, *Vector Replacement Systems: A Formalism for Modeling Asynchronous Systems*, Technical Report 117, Computer Science Laboratory, Princeton University, December 1972, 38 pages; revised January 1974, 50 pages.
- [429] R. Keller, *Generalized Petri Nets as Models for System Verification*, Technical report 202, Department of Electrical Engineering, Princeton University, August 1975, 50 pages.
- [430] J.J. Kenney, *Executable Formal Models of Distributed Transaction Systems Based on Event Processing*, Ph.D. Thesis, Stanford University, June 1995 (expected).
- [431] B.W. Kernighan and D.M. Richie, *The C Programming Language*, Prentice Hall, 1978.
- [432] S. Kimura, "Residue BDD and its Application to the Verification of Arithmetic Circuits," In *Proceedings of the 32nd Design Automation Conference*, June 1995, pages 542-545.
- [433] S. Kimura and E.M. Clarke, Jr., "A Parallel Algorithm for Constructing Binary Decision Diagrams," In *IEEE International Conference on Computer Design*, September 1990.

- [434] G. Kildall, "A Unified Approach to Global Program Optimization," In *ACM Symposium on Principles of Programming Languages*, 1973, pages 194-206.
- [435] C.D. Kloos, *Semantics of Digital Circuits*, Ph.D. Thesis, Where? Available as Lecture Notes in Computer Science, Vol.285, Springer-Verlag, 1987.
- [436] D.W. Knapp, *A Planning Model of the Design Process*, Ph.D. Thesis, Department of Computer Science, University of Southern California, December 1986.
- [437] D. Knapp, T. Ly, D. MacMillen and R. Miller, "Behavior Synthesis Methodology for HDL-Based Specification and Validation," In *Proceedings of the 32nd Design Automation Conference*, June 1995, pages 280-291.
- [438] D.E. Knuth, "Seminumerical Algorithms," In *The Art of Computer Programming*, Vol.2, Addison Wesley, 1981.
- [439] Z. Kohavi, *Switching and Finite Automata Theory*, McGraw-Hill, 1978.
- [440] S. Kono, T. Aoyagi, M. Fujita and H. Tanaka, "Implementation of Temporal Logic Programming Language Tokio," In *Proceedings of the Logic Programming Conference '85*, 1985, pages 138-147.
- [441] D. Kozen, "Results on the Propositional μ -Calculus," In *Theoretical Computer Science*, Vol.8, December 1983, pages 333-354.
- [442] D. Kozen and J. Tiuryn, "Logics of Programs," In *Handbook of Theoretical Computer Science* [327], Vol. B, 1990, pages 789-840.
- [443] S.A. Kripke, "Semantical Analysis of Modal Logic," In *I.Z. Mathematical Logick Grundlagen Math.* Vol.9, 1963, pages 67-96.
- [444] F. Kröger, *Temporal Logic of Programs*, Springer-Verlag, 1987.
- [445] D.C.-L. Ku and G. De Micheli, *Hardware C - A Language for Hardware Design*, Technical Report CSL-TR-88-362, August 1988.
- [446] D.C.-L. Ku and G. De Micheli, *Hardware C - A Language for Hardware Design (version 2.0)*, Technical Report CSL-TR-90-419, April 1990.

- [447] D.C.-L. Ku, *Constrained Synthesis and Optimization of Digital Integrated Circuits from Behavioral Specifications*, Ph.D. Thesis, Stanford University, June 1991, Technical Report CSL-TR-91-476.
- [448] H.-T. Kung, R. Sproull and G. Steele (editors), *VLSI Systems and Computations*, Computer Science Press, 1981.
- [449] R.P. Kurshan, "Modelling Concurrent Processes." In *Proceedings of the Symposium on Applied Math*, Vol.3, 1985, pages 45-57.
- [450] R.P. Kurshan, *Testing Containment of ω -Regular Languages*, Technical Memo 1121-861010-33-TM., Bell Laboratories, NJ, 1986.
- [451] R.P. Kurshan, "Reducibility in Analysis of Coordination." In *Discrete Event Systems: Models and Applications*, Lecture Notes in Computer Information Systems, Vol. 103, Springer-Verlag, 1987, pages 19-39.
- [452] R. P. Kurshan. "Reducibility in Analysis of Coordination." In *Journal of Computing Systems Science*, Vol. 35, 1987, pages 59-71.
- [453] R.P. Kurshan, "Analysis of Discrete Event Coordination," In *Proceedings of the REX Workshop on Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*. Lecture Notes in Computer Science, Vol.430, M.W. de Bakker, W.P. de Roever and G. Rozenberg, editors, Springer-Verlag, New York, 1989.
- [454] R.P. Kurshan. *Computer-Aided Verification of Coordinating Processes; The Automata-Theoretic Approach*, Princeton University Press, 1994.
- [455] Y.T. Lai and S. Sastry, "Edge-Valued Binary Decision Diagrams for Multi-Level Hierarchical Verification," In *Proceedings of the 29th Design Automation Conference*, June 1992, pages 608-613.
- [456] L. Lamport, "Proving the Correctness of Multiprocess Programs," In *Transactions on Software Engineering*, Vol.3, No.2, March 1977, pages 125-133.
- [457] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System." In *Communications of the ACM*, Vol.21, No.7, 1978, pages 558-564.
- [458] L. Lamport, "'Sometimes' is Sometimes 'Not Never': on the Temporal Logic of

Programs,” In *Proceedings of the 7th Symposium on the Principles of Programming Languages*, 1980, pages 207-222.

- [459] L. Lamport and F.B. Schneider, “The ‘Hoare Logic’ of CSP and All That,” In *ACM Transactions on Programming Languages*, Vol. 6, No. 2, April 1984, pages 281-296.
- [460] P. Landin, “The Mechanical Evaluation of Expressions.” In *Computer Journal*, Vol.6, 1964, pages 308-320.
- [461] P. Landin, “A Correspondence Between Algol60 and Church’s Lambda Notation,” In *Communications of the ACM*, Vol.8, 1965, pages 89-101, 158-165.
- [462] P. Landin, “A Lambda Calculus Approach,” In *Advances in Programming and Non-Numerical Computation*, L. Fox editor, Pergamon Press, 1966.
- [463] P.E. Lauer and R.H. Campbell, “Formal Semantics of a Class of High-Level Primitives for Co-ordinating Concurrent Processes,” In *Acta Informatica*, Vol.5, 1975, pages 297-332.
- [464] P. Le Guernic, A. Benveniste, P. Bournai and T. Gautier, “SIGNAL: A Data-Flow Oriented Language for Signal Processing,” In *Transactions on Application-Specific Signal Processing*, Vol.34, No.2, 1986, pages 362-374.
- [465] P. Le Guernic, T. Gauthier, M. Le Borgne and C. Le Maire, “Programming Real-Time Applications with Signal,” In *Proceedings of the IEEE*, Vol.79, No.9, September 1991, pages 1321-136.
- [466] J. O’ Leary, M. Linderman, M. Leeser and M. Aagaard, *HML: A Hardware Description Language Based on SML*, Technical Report EE-CEG-92-7, Cornell School of Electrical Engineering.
- [467] J. O’ Leary, M. Linderman, M. Leeser and M. Aagaard, “HML: A Hardware Description Language Based on SML,” In *Proceedings of the Conference on Computer Hardware Description Languages and their Applications*, 1993.
- [468] C. Y. Lee, “Representation of Switching Circuits by Binary-Decision Programs,” In *Bell Systems Technical Journal*, Vol.38, 1959, pages 985-999.
- [469] E. A. Lee and D.G. Messerschmitt, “Synchronous Data Flow,” In *Proceedings of the IEEE*, Vol.75, No.9, September 1987, pages 1235-1245.

- [470] E. A. Lee and D. G. Messerschmitt, "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing," In *IEEE Transactions on Computers*, Vol. 36, No. 2, January 1987.
- [471] E. A. Lee, *Consistency in Data Flow Graphs*, Technical Memo UCB ERL M89/125, Electronics Research Laboratory, University of California, Berkeley, 1989.
- [472] E. A. Lee, "Consistency in Data Flow Graphs," In *Transactions on Parallel and Distributed Systems*, Vol. 2, April 1991, pages 223-235.
- [473] P. Lee, *Realistic Compiler Generation*, Ph.D. Thesis, Massachusetts Institute of Technology, 1988, Also available from The MIT Press, 1989.
- [474] C. Leiserson and J. Saxe, "Retiming Synchronous Circuitry," In *Algorithmica*, Vol. 6, No. 1, 1991, pages 5-35.
- [475] O. Levia, S. Maginot and J. Rouillard, "Lessons in Language Design: Cost/Benefit Analysis of VHDL Features," In *Proceedings of the 31st Design Automation Conference*, June 1994., pages 447-453.
- [476] O. Lichtenstein and A. Pnueli, "Checking that Finite State Concurrent Programs Satisfy their Linear Specifications," In *Proceedings of the 12th Symposium on the Principles of Programming Languages*, January 1985, pages 97-107.
- [477] B. Lin, *Synthesis of VLSI Design with Symbolic Techniques*, Ph.D. Thesis, Department of Electrical Engineering and Computer Science, University of California, 1991. Technical Memo UCB/ERL M91/105.
- [478] R. Lipsett, C. F. Schaffer and C. Ussery, *VHDL: Hardware Description and Design*, Kluwer Academic Publishers, 1989.
- [479] R. J. Lipton and R. Sedgewick, "Lower Bounds for VLSI," In *Proceedings of the 13th Annual ACM Symposium on the Theory of Computation*, 1980, pages 300-307.
- [480] J. S. Lis, *Behavioral Synthesis from VHDL using Structural Modeling*, Ph. D. Thesis, Department of Information and Computer Science, University of California, Irvine, 1991, Technical Report #91-05.
- [481] B. Lisper, *Synthesizing Synchronous Systems by Static Scheduling in Space-Time*, Ph.D. Thesis, Department of Numerical Analysis and Computer Science, Royal

Institute of Technology, Stockholm, Sweden, February 1987. Also available from Springer-Verlag, Lecture Notes in Computer Science, Vol.362, 1989..

- [482] A. Lombardo and S. Palazzo, *An Extended Algebra for the Validation of Communication Protocols*, ESPRIT SEDOS, Project Public Report No.75, November 1986.
- [483] A. Lombardo and S. Palazzo, "An Algebraic Method for Modeling Communication Among Distributed Transition Systems," In *Proceedings of the Conference on Information Processing Systems(?)*, November 1987.
- [484] D.E. Long, *Model Checking Abstraction and Compositional Verification*, Ph.D. Thesis, School of Computer Science, Carnegie Mellon University, 1993.
- [485] D.E. Long, A. Browne, E.M. Clarke, Jr., S. Jha and W.R. Marrero, "An Improved Algorithm for the Evaluation of Fixpoint Expressions," In *Proceedings of the Conference on Computer-Aided Verification (CAV '94)*, D.L. Dill editor, Lecture Notes in Computer Science, Vol.818, June 1994, pages 338-350.
- [486] D.C. Luckham, Y. Huh and A.G. Stanculescu, *The Semantics of Timing Constructs in Hardware Description Languages*, Technical Report CSL-TR-86-303, Computer Systems Laboratory, Stanford University, August 1986.
- [487] D.C. Luckham, Y. Huh and A.G. Stanculescu, "The Semantics of Timing Constructs in Hardware Description Languages," In *Proceedings of the International Conference on Computer Design*, October 1986, pages 10-14.
- [488] D.C. Luckham and B.A. Gennart, *Event Patterns: A Language Construct for Hierarchical Design of Concurrent Systems*, Technical Report CSL-TR-90-453, Computer Systems Laboratory, Stanford University, November 1990.
- [489] D.C. Luckham, J.J. Kenney, L.M. Augustin, J. Vera, D. Bryan and W. Mann, "Specification and Analysis of System Architecture Using Rapide," In *Transactions on Software Engineering*, Vol.21, No.4, April 1995, pages 336-354.
- [490] N.A. Lynch and M.R. Tuttle, *Hierarchical Correctness Proofs for Distributed Algorithms*, Technical Report MIT/LCS/TR-387, MIT Laboratory for Computer Science, April 1987.
- [491] N.A. Lynch and M.R. Tuttle, "Hierarchical Correctness Proofs for Distributed Algorithms," In *Proceedings of the 6th Symposium on the Principles of Distributed*

Computing, 1987, pages 137-151.

[492]

E. Madelaine and D. Vergamini, "Finiteness Conditions and Structural Construction of Automata for All Process Algebras," In *Proceedings of the 2nd International Conference on Computer-Aided Verification (CAV '90)*, E. Clarke and R. Kurshan editors, Lecture Notes in Computer Science, Vol.531, Springer-Verlag, June 1990, pages 353-363.

[493]

F. Mailhot and G. De Micheli, *Structural/Logic Intermediate Form Specification*, Technical Report, Stanford University, 1988.

[494]

S. Malik, "Analysis of Cyclic Combinational Circuits," In *Proceedings of the International Conference on Computer Aided Design*, November 1993, pages 618-625.

[495]

S. Malik, "Analysis of Cyclic Combinational Circuits." In *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol.13, No.7, July 1994, pages 950-956.

[496]

S. Malik, A.R. Wang, R.K. Brayton and A. Sangiovanni-Vincentelli, "Logic Verification Using Binary Decision Diagrams in a Logic Synthesis Environment," In *Proceedings of the International Conference on Computer-Aided Design*, November 1988, pages 6-9.

[497]

Z. Manna, *Mathematical Theory of Computation*, McGraw-Hill, New York, 1974.

[498]

Z. Manna and A. Pnueli, "Verification of Concurrent Programs: the Temporal Framework," In *The Correctness Problem in Computer Science*, R. S. Boyer and J. S. Moore, editors. 1981, pages 215-273.

[499]

Z. Manna and A. Pnueli, "Specification and Verification of Concurrent Programs by \forall -Automata," In *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages*, 1987, pages 1-12.

[500]

Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems: Specification*, Springer-Verlag, 1991.

[501]

F. Maraninchi, "Argonaute: Graphical Description, Semantics and Verification of Reactive Systems Using a Process Algebra," In *Automatic Verification Methods for Finite State Systems*, J. Sifakis editor, Lecture Notes in Computer Science, Springer-Verlag, June 1989, pages 38-53.

- [502] F. Maraninchi, *Argos: un Langage Graphique Pour la Conception, la Description et la Validation des Systèmes Réactifs*, Ph.D. Thesis, Université Joseph Fourier, Grenoble, 1990.
- [503] F. Maraninchi, "Operational and Compositional Semantics of Synchronous Automaton Generation," In *Proceedings of the 3rd International Conference on Concurrency Theory (CONCUR '92)*, Lecture Notes in Computer Science, Vol.630, 1992.
- [504] T. Maro, F. Maruyama, K. Hayoshi, T. Kakuda, N. Kawato and T. Uehara, "OCCAM to CMOS: Experimental Logic Design Support System," In *Computer Hardware Description Languages and their Applications (CHDL '85)*, 1985, pages 381-396.
- [505] Y. Matsunaga, P.C. McGeer and R.K. Brayton, "On Computing the Transitive Closure of a State Transition Relation," In *Proceedings of the 30th Design Automation Conference*, June 1993, pages 260-265.
- [506] D. May, "Occam," In *ACM SIGPLAN Notices*, Vol. 18, No.4, 1983, pages 69-79.
- [507] D. May and R. Taylor, "Occam - an Overview," In *Microprocessors and Microsystems*, Vol.8, No.2, 1984, pages 73-79.
- [508] D. May and R. Shepherd, "Occam and the Transputer," In *Proceedings of the IFIP Workshop on Hardware Supported Implementations of Concurrent Languages in Distributed Systems*, March 1986.
- [509] A. Mazurkiewicz, "Trace Theory," In *Petri Nets: Applications and Relationships to Other Models of Concurrency, Advances in Petri Nets 1986, Part II; Proceedings of an Advanced Course*, Lecture Notes in Computer Science, Vol.255, 1986, pages 279-324.
- [510] A. Mazurkiewicz, "Basic Notions of Trace Theory," In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, Lecture Notes in Computer Science, Vol.354, 1988, pages 285-363.
- [511] J. McCarthy, "Recursive Functions of Symbolic Expressions and their Computation by Machine," In *Communications of the ACM*, Vol.3, No.4, 1960, pages 184-195.
- [512] M.C. McFarland, S.J., *The Value Trace: A Data Base for Automated Digital Design*,

Master's Thesis, Department of Electrical Engineering, Carnegie Mellon University, December 1978, Technical Report DRC-01-04-80.

- [513] M.C. McFarland, S.J., *Mathematical Models for Verification in a Design Automation System*, Ph.D. Thesis, School of Computer Science, Carnegie Mellon University, July 1981.
- [514] M.C. McFarland, S.J., "On Proving the Correctness of Optimizing Transformations in a Digital Design Automation System." In *Proceedings of the 18th Design Automation Conference*, June 1981, pages 90-97.
- [515] M.C. McFarland, S.J. and A.C. Parker, "An Abstract Model of Behavior for Hardware Description," In *IEEE Transactions on Computers*, Vol.32, July, 1983, pages 621-637.
- [516] P.C. McGeer and R.K. Brayton. *Integrating Functional and Temporal Domains in Logic Design*, Kluwer Academic Publishers, 1991.
- [517] J.R. McGraw. "The VAL Language, Description and Analysis," In *Transactions on Programming Languages and Systems*, Vol.4, No. 1, January 1982.
- [518] K. McMillan. *Symbolic Model Checking*, Ph.D. Thesis, School of Computer Science, Carnegie-Mellon University, May 1992; Also, Kluwer Academic Publishers, 1993.
- [519] K.L. McMillan, "Hierarchical Representations of Discrete Functions with Application to Model Checking," In *Proceedings of the Conference on Computer-Aided Verification (CAV '94)*, D.L. Dill editor, Lecture Notes in Computer Science, Vol.818, Springer-Verlag, June 1994, pages 41-54.
- [520] K.L. McMillan and J. Schwalbe. "Formal Verification of the Encore Gigamax Cache Consistency Protocols," In *Proceedings of the International Symposium on Shared Memory Multiprocessors*, April 1991.
- [521] R. McNaughton. "Testing and Generating Infinite Sequences by a Finite Automaton," In *Information and Control* 9, 1966, pages 521-530.
- [522] M.R. Mercer. R. Kapur and D.E. Ross, "Functional Approaches to Generating Orderings for Efficient Symbolic Representations," In *Proceedings of the 29th Design Automation Conference*, June 1992, pages 624-627.
- [523] D. Méry and D. Molkenden, "CrocOS: An Integrated Environment for Interactive

Verification of SDL Specifications,” In *Proceedings of the Conference on Computer-Aided Verification (CAV '92)*, G. van Bochmann, D.K. Probst, editors. Lecture Notes in Computer Science, Vol.663, 1992, Springer-Verlag, pages 343-356.

[524]

F. Meshkinpour and M.D. Ercegovac, “A Functional Language for Description and Design of Digital Systems: Sequential Constructs,” In *Proceedings of the 22nd Design Automation Conference*, 1985, pages 238-244.

[525]

F. Mignard, *Compilation du Langage Esterel en Systèmes d'Equations Booléennes*, Thèse de Doctorat de L'Ecole des Mines de Paris, October 1995.

[526]

G.J. Milne, “CIRCAL and the Representation of Communication, Concurrency and Time,” In *Transactions on Programming Languages and Systems*, Vol. 7, No.2, April 1985. pages 270-298.

[527]

G.J. Milne, “Design for Verifiability,” In *Proceedings of the Workshop on Hardware Specification, Verification and Synthesis: Mathematical Aspects*. Lecture Notes in Computer Science, July 1989, pages 1-13.

[528]

G.J. Milne and C.Strachey, *A Theory of Programming Language Semantics*, Chapman and Hall, 1976.

[529]

R. Milner, “Processes: A Mathematical Model of Computing Agents.” In *Proceedings of the Logic Colloquium '73*, H.E. Rose and J.C. Shepherdson, editors, North Holland 1975, pages 157-174.

[530]

R. Milner, “Fully Abstract Models of Typed Lambda-Calculi.” In *Theoretical Computer Science*, Vol.4, No. 1, 1977, pages 1-23. A[[◦]]

[531]

R. Milner, *A Calculus of Communicating Systems*, Lecture Notes in Computer Science, Vol.92, Springer-Verlag, 1980.

[532]

R. Milner, “Calculi for Synchrony and Asynchrony,” In *Theoretical Computer Science*, Vol.25, 1983, pages 267-310.

[533]

R. Milner, *Communication and Concurrency*, Prentice Hall 1989.

[534]

R. Milner, “Operational and Algebraic Semantics of Concurrent Processes,” In *Handbook of Theoretical Computer Science* [327], Vol. B, 1990, pages 1201-1242.

- [535] R. Milner, *The Polyadic π -Calculus: A Tutorial*, Research Report, ECS-LFCS-91-180. University of Edinburgh, October 1991.
- [536] R. Milner, *Action Structures for the π -Calculus*, Research Report ECS-LFCS-91-264, University of Edinburgh, May 1993.
- [537] R. Milner, M. Tofte and R. Harper, *The Definition of Standard ML*, The MIT Press, 1990.
- [538] S. Minato, "Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems," In *Proceedings of the 30th Design Automation Conference*, 1993, pages 272-277.
- [539] S. Minato, N. Ishiura and S. Yajima, "Shared Binary Decision Diagram with Attributed Edges for Efficient Boolean Function Manipulation," In *Proceedings of the 27th Design Automation Conference*, June 1990, pages 52-57.
- [540] J. Morison, N.E. Peeling and T.L. Thorpe, "ELLA: A Hardware Description Language," In *Proceedings of the International Conference on Circuits and Computers*, September 1982, pages 604-607.
- [541] J. Morison, N.E. Peeling and T.L. Thorpe, "ELLA: Hardware Description or Specification?" In *Proceedings of the International Conference on Computer-Aided Design*, November 1984.
- [542] J. Morison, N.E. Peeling and T.L. Thorpe, "The Design Rationale of ELLA. A Hardware Design and Description Language," In *Proceedings of the 7th International Conference on Computer Hardware Description Languages and their Applications*, August 1985.
- [543] P.D. Mosses, "Abstract Semantic Algebras!" In *Formal Description of Programming Concepts III*, IFIP IC-2 Working Conference, D. Bjorner editor, 1982, pages 63-88.
- [544] P.D. Mosses, "A Basic Abstract Semantic Algebra," In *Semantics of Data Types*, G. Kahn, D.B. MacQueen and G. Plotkin, editors, Lecture Notes in Computer Science, Vol. 173, 1984, pages 87-107.
- [545] P.D. Mosses, "Denotational Semantics," In *Handbook of Theoretical Computer*

Science [327], Vol. B, 1990, pages 136-186.

- [546] B. Moszkowski, *Reasoning About Digital Circuits*, Ph.D. Thesis, Stanford University, 1983.
- [547] B. Moszkowski, *Executing Temporal Logic Programs*, Cambridge University Press, 1986
- [548] B. Moszkowski and Z. Manna, "Reasoning in Interval Temporal Logic," In *Workshop on Logics of Programs*, Lecture Notes in Computer Science, Vol.164, Springer-Verlag, 1984.
- [549] D.E. Muller, "Infinite Sequences and Finite Machines," In *Proceedings of the 4th Symposium on Switching Circuit Theory and Logical Design*, October 1969, pages 3-16.
- [550] K. Mulmuley, *Full Abstraction and Semantic Equivalence*, Ph.D. Thesis, Carnegie Mellon University, 1986; Also available from The MIT Press, 1986.
- [551] H. Nakamura, Y. Kukimoto, M. Fujita and H. Tanaka, "A Data Path Verifier for Register Transfer Level Using Temporal Logic Language Tokio," In *Proceedings of the 2nd International Conference on Computer-Aided Verification (CAV '90)*, E. Clarke and R. Kurshan editors, Lecture Notes in Computer Science, Vol.531, Springer-Verlag, June 1990, pages 76-85.
- [552] S. Narayan and F. Vahid, *Translation SpecCharts to VHDL*, Technical Report #90-21, Department of Information and Computer Science, University of California, Irvine, 1990.
- [553] S. Narayan, F. Vahid and D. Gajski. *Modeling with SpecCharts*, Technical Report #90-20, Department of Information and Computer Science, University of California, Irvine, July 25, 1990, Revised October 15, 1992.
- [554] S. Narayan, F. Vahid and D. Gajski, "SpecCharts: A Language for System Level Synthesis," In *Proceedings of the International Symposium on Computer Hardware Description Languages and their Applications*, 1991.
- [555] S. Narayan, F. Vahid and D. Gajski, "System Specification and Synthesis with the SpecCharts Language," In *Proceedings of the International Conference on Computer-Aided Design*, 1991, page 266-269.

- [556] P. Naur, *et al.*, "Report on the Algorithmic Language Algol 60," In *Communications of the ACM*, Vol.3, No.5, 1960, pages 299-314.
- [557] P. Naur, *et al.*, "Revised Report on the Algorithmic Language Algol 60," In *Communications of the ACM*, Vol.6, No. 1, 1960, pages 1-17.
- [558] H.R. Nielson and F. Nielson, "Semantics-Directed Compiling for Functional Languages," In *Proceedings of the ACM Conference on Lisp and Functional Programming*, 1986, pages 249-257.
- [559] H.R. Nielson and F. Nielson, *Semantics with Applications: A Formal Introduction*, John Wiley and Sons, 1992.
- [560] M. Nielsen, G. Plotkin and G. Winskel, "Petri Nets, Event Structures and Domains, Part I," In *Theoretical Computer Science*, Vol. 13, 1981, pages 85-108.
- [561] *HSL-FX User's Manual*, NTT LSI Laboratories, 1988.
- [562] J.T. O'Donnell, "Hardware Description with Recursion Equations," In *Computer Hardware Description languages and their Applications*, M.R. Barbacci and C.J. Koomen editors, Elsevier Science Publishers, 1987, pages 363-382.
- [563] J.D. Oakley, *Symbolic Execution of Formal Machine Descriptions*, Ph.D. Thesis Computer Science Department, Carnegie Mellon University, April 1979.
- [564] H. Ochi, K. Yasuoka and S. Yajima, "Breadth First Manipulation of Very Large Binary Decision Diagrams," In *Proceedings of the International Conference on Computer-Aided Design*, November 1993, pages 48-55.
- [565] K. Oguri, Y. Nakamura and N. Nomura, "Evaluation of Behavior Description Based CAD System used in Prolog Machine Design," In *Proceedings of the International Conference on Computer-Aided Design*, November 1986, pages 116-119.
- [566] E.-R. Olderog, "A Characterization of Hoare's Logic for Programs with Pascal-Like Procedures," In *Proceedings of the 15th ACM Symposium on the Theory of Computers*, 1983, pages 320-329.
- [567] S. Oleoz and J.M. Colom, "The Discrete Event Simulation Semantics of VHDL," In *Proceedings of the International Conference on Simulation and Hardware Description Languages*, January 1994, pages 128-134.

- [568] A.V. Oppenheim, A.S. Willsky with I.T. Young, *Signals and Systems*, Prentice-Hall, 1983.
- [569] F. Orava, "Formal Semantics of SDL Specifications," In *Protocol Specification, Testing and Verification VIII*, Elsevier Science Publishers, 1988.
- [570] *Verilog Hardware Description Language Reference Manual (LRM) Version 1.0*, November 1991, Open Verilog International, Suite 408, 1016 East El Camino Real, Sunnyvale CA 94087.
- [571] J.K. Ousterhout, *Tcl and the Tk Toolkit*, Addison Wesley, 1994.
- [572] S. Owicki and L. Lamport, "Proving Liveness Properties of Concurrent Programs, In *Transactions on Programming Languages and Systems*, Vol4, No.3, July 1982, pages 455-495.
- [573] I. Page and W. Luk, "Compiling OCCAM into FPGAs," In *Field Programmable Gate Arrays (FPGAs)*, W. Moore and W. Luk, editors, Abingdon EE & CS Books 1991, pages 271-283.
- [574] J.L. Paillet, "A Functional Model for Descriptions and Specifications of Digital Devices," In *From HDL Descriptions to Guaranteed Correct Circuit Designs*, 1987, pages 21-42.
- [575] C.H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization*, Prentice Hall, 1982.
- [576] J.-P. Paris, *Exécution de Tâches Asynchrones Depuis Esterel*, Thèse d'Informatique, Université de Nice, 1992.
- [577] J. Paris, G. Berry, F. Mignard, P. Couronné, P. Caspi, N. Halbwachs, Y. Sorel, A. Benveniste, T. Gautier, P. Le Guernic, F. Dupont and C. Le Maire, *Projet SYNCHRONE. Les Formats Communs des Langages Synchrones*, Rapport Technique No.157, Institut National de Recherche en Informatique et en Automatique (INRIA), June 1993; English version W.C. Baker, March 1994.
- [578] D. Park, *Finiteness is μ -ineffable*, Theory of Computation Report No.3, University of Warwick, 1974.

- [579] D. Park, *Finiteness is μ -ineffable*, In *Theoretical Computer Science*, Vol.3, 1976, pages 173-181.
- [580] D. Park, "Concurrency and Automata on Infinite Sequences," In *Theoretical Computer Science*, Lecture Notes in Computer Science, Vol.104, Springer-Verlag, 1981, pages 167-183.
- [581] D.L. Parnas, "A Language for Describing the Functions of Synchronous Systems," In *Communications of the ACM*, Vol.9, Feb 1966, pages 72-75.
- [582] J.P. Pécuchet, "On the Complementation of Büchi Automata," In *Theoretical Computer Science*, Vol.47, 1986, page 95-98.
- [583] D. Perrin, "Finite Automata," In *Handbook of Theoretical Computer Science* [327], Vol. B, 1990, pages 1-57.
- [584] J.L. Peterson, *Petri Net Theory and the Modeling of Systems*, Prentice Hall, 1981.
- [585] C.A. Petri, *Kommunikation mit Automaten*, Ph.D. Thesis, Schriften des Institutes für Instrumentelle Mathematik, Bonn, 1962, in German; English version [586].
- [586] C.A. Petri, *Communication with Automata*, Final Report, Vol. 1, Supp. 1, RADC TR-65-377-vol1-suppl, Applied Data Research, Princeton NJ, translated by C.F. Greene, Jr., January 1966.
- [587] D. Pilaud and N. Halbwachs, "From a Synchronous Declarative Language to a Temporal Logic Dealing with Multiform Time," In *Proceedings of the Symposium on Formal Techniques in Real Time and Fault Tolerant Systems*, September 1988.
- [588] R. Piloty, M. Barbacci, D. Borriane, D. Deitmeyer, F. Hill and P. Skelly, *The CONLAN Report*, Lecture Notes in Computer Science, Vol.151, Springer-Verlag, 1983
- [589] R. Piloty and D. Borriane, "The CONLAN Project: Concepts, Implementations and Applications," In *IEEE Computer*, Vol. 18, No.2, February 1985, pages 81-92.
- [590] C. Pixley, "A Computational Theory and Implementation of Sequential Hardware Equivalence," In *DIMACS Technical Report 90-31, Vol.2, Workshop on Computer-Aided Verification*, R. Kurshan and E.M. Clarke, editors, 1990.

- [591] C. Pixley, "Introduction to a Computational Theory and Implementation of Sequential Hardware Equivalence," In *Proceedings of Conference on Computer-Aided Verification (CAV '90)*, E.M. Clarke and R.P. Kurshan, editors, Lecture Notes in Computer Science, Vol.531, Springer-Verlag, 1990, pages 54-65.
- [592] C. Pixley, "A Theory and Implementation of Sequential Hardware Equivalence," In *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 11, No. 12, December 1992, pages 1469-1472.
- [593] J.A. Plaice, *Sémantique et Compilation de Lustre, un Langage Déclaratif Synchrones*, Thèse, Institut National Polytechnique de Grenoble, May 1988.
- [594] U.F. Pleban and P. Lee, "High-level Semantics - An Integrated Approach to Programming Language Semantics and the Specification of Implementations." In *Proceedings of the 3rd Workshop on the Mathematical Foundations of Programming Language Semantics*, Lecture Notes in Computer Science, Vol.298, Springer-Verlag, 1987.
- [595] G.D. Plotkin, "A Powerdomain Construction," In *SIAM Journal of Computing*, Vol.5, No.3, 1976, pages 452-487.
- [596] G.D. Plotkin. *A Structural Approach to Operational Semantics*, Ph.D. Thesis, Aarhus University, Denmark, 1981, Technical Report DAMI FN-19.
- [597] G.D. Plotkin, "A Powerdomain for Countable Nondeterminism," In *Proceedings of the 9th International Colloquium on Automata, Languages and Programming (ICALP)*, Lecture Notes in Computer Science, Vol. 140, Springer, 1982, pages 418-428.
- [598] H. Plünnecke and W. Reisig, "Bibliography of Petri Nets 1990," In *Advances in Petri Nets 1991*, 1991.
- [599] A. Pnueli, "The Temporal Semantics of Concurrent Programs," In *Proceedings of the 18th Symposium on the Foundations of Computer Science*, 1977, pages 46-57.
- [600] A. Pnueli, "Linear and Branching Structures in the Semantics and Logics of Reactive Systems," In *Proceedings of the 12th International Colloquium on Automata Languages and Programming (ICALP)*, W. Brover editor, Lecture Notes in Computer Science, Vol. 194, Springer-Verlag, 1985, pages 15-32.

- [601] A. Pnueli, "Applications of Temporal Logic to the Specification and Verification of Reactive Systems: a Survey of Current Trends," In *Lecture Notes in Computer Science*, Vol.224, Springer-Verlag, 1986. pages 510-584.
- [602] A. Pnueli and M. Shalev, "What is in a Step: On the Semantics of Statecharts," In *Proceedings of the International Conference on the Theoretical Aspects of Computer Software (TACS '91)*, T. Ito and A. R. Meyer, editors, Springer-Verlag, 1991.
- [603] V.R. Pratt, "Semantical Considerations on Floyd-Hoare Logic," In *Proceedings of the 17th Symposium on the Foundations of Computer Science*, October 1976, pages 109-121.
- [604] V. R. Pratt, "A Decidable μ -Calculus," In *Proceedings of the 22nd IEEE Symposium on the Foundations of Computer Science*. 1981, pages 421-427.
- [605] V.R. Pratt, "Modelling Concurrency with Partial Orders," In *International Journal of Parallel Programming*, Vol.15, No. 1, February 1986, pages 33-71.
- [606] F.P. Preparata and M.I. Shamos, *Computational Geometry, An Introduction*, Springer-Verlag, 1985.
- [607] D.K. Probst and H.F. Li, "Using Partial-Order Semantics to Avoid the State Explosion Problem," In *Proceedings of the Conference on Computer-Aided Verification (CAV '90)*, E.M. Clarke and R.P. Kurshan, editors, Lecture Notes in Computer Science, Vol.531, June 1990, pages 146-555.
- [608] J.-P. Queille and J. Sifakis, "Specification and Verification of Concurrent Systems in CESAR," In *Proceedings of the Fifth International Symposium in Programming*, 1981.
- [609] J.-P. Queille, *Le Système CESAR: Description, Spécification et Analyse des Applications Réparties*, Ph.D. Thesis, University de Grenoble, 1982.
- [610] M.O. Rabin, "Decidability of Second-Order Theories and Automata on Infinite Trees," In *Transactions of the American Mathematical Society*, Vol. 141, 1969, pages 1-35.
- [611] M.O. Rabin, "Automata on Infinite Objects and Church's Problem," In *American Mathematical Society*, 1972.

- [612] P. Ramadge and W.M. Wonham "On the Supremal Controllable Sublanguage of a Given Language," In *SIAM Journal of Control and Optimization*, Vol. 25, No. 3, May 1987, pages 637-659.
- [613] P. Ramadge and W.M. Wonham, "Supervisory Control of a Class of Discrete Event Processes," In *SIAM Journal of Control and Optimization*, Vol. 25, No. 1, January 1987, pages 206-230.
- [614] P. Ramadge and W.M. Wonham, "Modular Feedback Logic for Discrete Event Systems," In *SIAM Journal of Control and Optimization*, Vol.25, No.5, September 1987, pages 1202-1218.
- [615] G. Ramalingam, *Bounded Incremental Computation*, Ph.D. Thesis, Computer Science Department, University of Wisconsin, Madison, 1993.
- [616] D.L. Ravenscroft and M.R. Lightner, "Functional Language Extractor and Boolean Cover Generator," In *Proceedings of the International Conference on Computer-Aided Design*, November 1986, pages 120-123.
- [617] J. Rees, W. Clinger et al., "The Revised³ Report on the Algorithmic Language Scheme." In *ACM SIGPLAN Notices*, Vol.21, No. 12, 1986, pages 37-39.
- [618] W. Reisig, *Petri Nets, An Introduction*, EATCS Monographs in Theoretical Computer Science, Vol.4, Springer-Verlag, 1985.
- [619] M. Rem. J.L.A. van de Snepscheut and J.T. Udding, "Trace Theory and the Definition of Hierarchical Components." In *Proceedings of the 3rd CalTech Conference on Very Large Scale Integration*, Computer Science Press, 1983, pages 225-239.
- [620] J.C. Reynolds, "Definitional Interpreters for Higher-Order Programming Languages," In *Proceedings of the 25th ACM National Conference*, 1972, pages 717-740.
- [621] J.C. Reynolds, "On the Relation Between Direct and Continuation Semantics." In *Proceedings of the 2nd Colloquium on Automata, Languages and Programming*, J. Loeckx editor, Springer-Verlag, 1974.
- [622] J.-K. Rho and F. Somenzi, "Inductive Verification of Iterative Systems," In

Proceedings of the 29th Design Automation Conference, 1992, pages 628-633.

- [623] J. Richier, C. Rodriguez, J. Sifakis and J. Voiron, "Verification in XESAR of the Sliding Window Protocol," In *Proceedings of the 7th IFIP Symposium on Protocol Specification Testing, and Verification*, North-Holland, 1987.
- [624] F. Rocheteau and N. Halbwegs, "Implementing Reactive Programs on Circuits / A Hardware Implementation of LUSTRE," In *Real-Time: Theory in Practice, REX Workshop 1991*, J.W. de Bakker, C. Huizing, W.P. De Roever and G. Rozenberg, editors, Springer-Verlag, June 1991.
- [625] B.K. Rosen, M.N. Wegman and F.K. Zadek, "Global Value Numbers and Redundant Computations," In *Proceedings of the ACM Symposium on the Principles of Programming Languages*, January 1988, pages 12-27.
- [626] D.E. Ross, *Functional Calculations Using Ordered Partial Multi Decision Diagrams*, Ph.D. Thesis, University of Texas Austin, August 1990.
- [627] D.E. Ross, K.M. Butler, R. Kapur and M.R. Mercer, "Fast Functional Evaluation of Candidate OBDD Variable Orderings," In *Proceedings of the European Design Automation Conference*, February 1991, pages 4-10.
- [628] V. Roy, *Autograph: un Outil d'Analyse Graphique de Processus Parallèles Communicants*, Thèse, Université de Nice, 1990.
- [629] V. Roy and R. de Simone, "Auto and Autograph," In *Proceedings of Conference on Computer-Aided Verification (CAV '90)*, E.M. Clarke and R.P. Kurshan, editors, Lecture Notes in Computer Science, Vol. 531, Springer-Verlag, 1990, pages 65-75.
- [630] V. Roy and R. de Simone, "Auto and Autograph," In *Formal Methods in System Design*, Vol. 1, No. 2/3, October 1992, pages 239-249.
- [631] R.L. Rudell, *Logic Synthesis for VLSI Design*, Ph.D. Thesis, University of California, Berkeley, 1989. Technical Memo UCB/ERL M89/49.
- [632] R. Rudell, "Dynamic Variable Ordering for Binary Decision Diagrams," In *Proceedings of the International Conference on Computer-Aided Design*, Nov 1993, pages 42-47.
- [633] S. Safra, "On the Complexity of ω -Automata," In *Proceedings of the 29th*

Symposium on the Foundations of Computer Science, 1988, pages 319-327.

- [634] S. Safra, "Exponential Determinization for ω -Automata with Strong Fairness Acceptance Condition," In *Proceedings of the 24th Symposium on the Theory of Computing (STOC '92)*, 1992.
- [635] D.J. Salomon, "Four Dimensions of Programming Language Independence," In *ACM SIGPLAN Notices*, Vol.27, No.3, March 1992, pages 35-53.
- [636] R. Saracco, J.R.W. Smith and R. Reed, *Telecommunications Systems Engineering using SDL*, North Holland, 19??.
- [637] J. Scheichenzuber, W. Grass, U. Lauther and S. März, "Global Hardware Synthesis from Behavioral Dataflow Descriptions," In *Proceedings of the 27th Design Automation Conference*, 1990, pages 456-461.
- [638] D.A. Schmidt, "Detecting Global Variables in Denotational Specifications," In *Transactions on Programming Languages and Systems*, Vol.7, No.2, 1985, pages 299-310.
- [639] J.M. Schoen, *Performance and Fault Modeling with VHDL*, Prentice Hall, 1992.
- [640] D.S. Scott, "Outline of a Mathematical Theory of Computation," In *Proceedings of the 4th Princeton Conference on Information Sciences and Systems*, 1970, pages 169-176.
- [641] D.S. Scott, "The Lattice of Flow Diagrams," In *Proceedings of the Symposium on the Semantics of Algorithmic Languages*, Lecture Notes in Mathematics, Vol.188, Springer-Verlag, 1971, pages 311-366.
- [642] D.S. Scott, "Data Types as Lattices," In *SIAM Journal on Computing*, Vol.5, No.3, 1976, pages 522-587.
- [643] D.S. Scott, "Domains for Denotational Semantics," In *Proceedings of the ?th International Colloquium on Automata, Languages and Programming (ICALP)*, 1982, pages 577-611.
- [644] D.S. Scott and C. Strachey, "Toward a Mathematical Semantics for Computer Languages," In *Proceedings of the Symposium on Computers and Automata*, Microwave Research Institute Symposia Series, Vol.21, Polytechnic Institute of

Brooklyn, 1971, pages 19-46.

[645]

A. Seawright and F. Brewer, "Synthesis from Production-Based Specifications," In *Proceedings of the 29th Design Automation Conference*, June 1992, pages 194-199.

[646]

A. Seawright and F. Brewer, "High-Level Symbolic Construction Techniques for High Performance Sequential Synthesis," In *Proceedings of the 30th Design Automation Conference*, June 1993, pages 424-428.

[647]

R.B. Segal, *BDSYN: Logic Description Translator, BDSIM: Switch-Level Simulator*, M.S. Report, Department of Electrical Engineering and Computer Science, University of California, Berkeley, May 1987, Technical Memo, UCB/ERL M87/33.

[648]

E. Sentovich, K.J. Singh, C. Moon, H. Savoj, R. K. Brayton and A. Sangiovanni-Vincentelli, "Sequential Circuit Design Using Synthesis and Optimization," In *Proceedings of the International Conference on Computer Design*, October 1992, pages 328-333.

[649]

E. Sentovich, K.J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. Stephan, R. Brayton and A. Sangiovanni-Vincentelli, *SIS: A System for Sequential Circuit Synthesis*, Technical Report UCB/ERL M92/41, Electronics Research Laboratory, University of California, Berkeley, May 1992.

[650]

M. Shadad, R. Lipsett, E. Marschner, K. Shechan, H. Cohen, R. Waxman and D. Ackley, "VHSIC Hardware Description Language," In *IEEE Computer*, Vol. 18, No.2, February 1985.

[651]

M. Sheeran, "μFP, A Language for VLSI Design," In *Proceedings of the Symposium on Lisp and Functional Programming*, 1984, pages 104-112.

[652]

A. Shen, S. Devadas and A. Ghosh, "Probabilistic Construction and Manipulation of Free Boolean Diagrams," In *Proceedings of the International Conference on Computer-Aided Design*, November 1993, pages 544-549.

[653]

T.R. Shiple, Personal Communication, 1995.

[654]

T.R. Shiple, M. Chiodo, A.L. Sangiovanni-Vincentelli and R.K. Brayton, "Automatic Reduction in CTL Compositional Model Checking," In *Proceedings of the Conference on Computer-Aided Verification (CAV '92)*, G. van Bochmann and D.K. Probst, editors, Lecture Notes in Computer Science, Vol.663, 1992, Springer-

Verlag, pages 234-247.

[655]

T.R. Shiple, R. Hojati, A.L. Sangiovanni-Vincentelli and R.K. Brayton, "Heuristic Minimization of BDDs Using Don't Cares." In *Proceedings of the 31st Design Automation Conference*, 1994, pages 225-231.

[656]

G. Shurek and O. Grumberg, "The Modular Framework of Computer-Aided Verification: Motivation Solutions and Evaluation Criteria," In *Proceedings of the Workshop on Computer-Aided Verification*, DIMACS Series, June 1990.

[657]

G. Shurek and O. Grumberg, "The Modular Framework of Computer-Aided Verification," In *Proceedings of the Conference on Computer-Aided Verification (CAV '90)*, E.M. Clarke and R.P. Kurshan, editors, Lecture Notes in Computer Science, Vol.531, Springer-Verlag, 1990, pages 186-196.

[658]

D. Siefkes, *Decidable Theories I, Büchi's Monadic Second-Order Successor Arithmetics*, Lecture Notes in Mathematics, Vol.120. Springer-Verlag, 1970.

[659]

J. Sifakis, "Property-Preserving Homomorphisms of Transition-Systems," In *Logic of Programs*, Lecture Notes in Computer Science, Vol.164, pages 183, pages 458-473.

[660]

D.P. Siewiorek, C. G. Bell and A. Newell, *Computer Structures: Principles and Examples*, McGraw-Hill, 1982.

[661]

A.P. Sistla and E.M. Clarke, Jr., "Complexity on Propositional Temporal Logics." In *Journal of the ACM*, Vol.32, No.3, July 1986, pages 733-749.

[662]

A.P. Sistla, M.Y. Vardi and P. Wolper, "The Complementation Problem for Büchi Automata, with Applications to Temporal Logic," In *Proceedings of the 12th International Colloquium on Automata, Languages and Programming*, Lecture Notes in Computer Science, Springer Verlag, 1985.

[663]

A.P. Sistla, M.Y. Vardi, and P. Wolper, "The Complementation Problem for Büchi Automata, with Applications to Temporal Logic," In *Theoretical Computer Science*, Vol.49, 1987, pages 217-237.

[664]

E. A. Snow, *Automation of Module Set Independent Register-Transfer Level Design*, Ph.D. Thesis, Carnegie Mellon University, April 1978.

[665]

S. Sjøe and K. Karplus, "Logic Minimization using Two-Column Rectangle

Replacement.” In *Proceedings of the 28th Design Automation Conference*, June 1991, pages 470-474.

[666]

O.V. Sokolsky and S.A. Smolka, “Incremental Model Checking in the Modal μ -Calculus,” In *Proceedings of the Conference on Computer-Aided Verification (CAV'94)*, D.L. Dill editor, Lecture Notes in Computer Science, Vol.818, June 1994, pages 351-363.

[667]

A. Srinivasan, T. Kam. S. Malik and R. Brayton, “Algorithms for Discrete Function Manipulation.” In *Proceedings of the International Conference on Computer-aided Design*, November 1990, pages 92-95.

[668]

G.L. Steele, Jr. and G.J. Sussman, *Scheme: An Interpreter for the Extended Lambda Calculus*, Memo 349, MIT Artificial Intelligence Laboratory, 1975.

[669]

B. Steffen, “Data Flow Analysis as Model Checking,” In *Proceedings of the Symposium on Theoretical Aspects of Computer Science (TACS '91)*, Lecture Notes in Computer Science, Vol.526, Springer-Verlag, 1991.

[670]

E. Sternheim, R. Singh and Y. Trivedi, *Hardware Modeling with Verilog HDL*, Automata Publishing Company, 1990.

[671]

N.S. Stollon and J.D. Provence, “Measures of Syntactic Complexity for Modeling Behavioral VHDL.” In *Proceedings of the 32nd Design Automation Conference*, June 1995, pages 684-689.

[672]

B. Stoustrup, *The Design and Evolution of C++*, Addison-Wesley, 1994.

[673]

J.E. Stoy, *The Scott-Strachey Approach to Programming Language Theory*, The MIT Press, 1977.

[674]

R.S. Streett, “Propositional Dynamic Logic of Looping and Converse is Elementary Decidable.” In *Information and Control*, Vol. 54, 1982, pages 121-141.

[675]

A. Suárez, “Compiling ML into CAM,” In *Logical Foundations of Functional Programming*, G. Huet, editor, Addison-Wesley, 1990, pages 47-73.

[676]

The Java Language and Java Virtual Machine Papers, Sun Microsystems Inc., Mountain View, CA, 1994; Also available at <http://java.sun.com>.

- [677] G.M. Swamy and R.K. Brayton, *Incremental Formal Design Verification*, Technical Report UCB/ERL M94/??, Electronics Research Laboratory, University of California, 1994.
- [678] G.M. Swamy and R.K. Brayton, "Incremental Formal Design Verification," In *Proceedings of the International Conference on Computer-Aided Design*, 1994, pages 458-465.
- [679] A. Takach, W. Wolf and M. Leeser, *An Automaton Model for Scheduling Constraints*, Technical Report CE-W92-16, Princeton University, February 1992.
- [680] R.E. Tarjan, "Depth-First Search and Linear Graph Algorithms," In *SIAM Journal of Computing*, Vol. 1, 1972, pages 146-160.
- [681] A. Tarski, "A Lattice-Theoretical Fixpoint Theorem and its Applications." In *Pacific Journal of Mathematics*, Vol. 5, 1955, pages 285-309.
- [682] D. Taubner, *Finite Representations of CCS and TCSP Programs by Automata and Petri Nets*, Ph.D. Thesis, Institut für Informatik, Technische Universität München, 1989. Also available from Springer-Verlag, 1989.
- [683] J. Thistle, *Control of Infinite Behavior of Discrete-Event Systems*, Ph.D. Thesis, University of Toronto, 1991.
- [684] W. Thomas, "Automata on Infinite Objects." In *Handbook of Theoretical Computer Science* [327], Vol. B, 1990, pages 136-186.
- [685] D. Thomas and P. Moorby, *The Verilog Hardware Description Language*, Kluwer Academic Publishers, 1991.
- [686] C.D. Thompson, "Area-Time Complexity for VLSI" In *Proceedings of the Annual ACM Symposium on the Theory of Computing*, 1979, pages 81-88.
- [687] C.D. Thompson, *A Complexity Theory for VLSI*, Ph.D. Thesis, Carnegie-Mellon University, 1980.
- [688] H. Touati and G. Berry, "Optimized Controller Synthesis using Esterel," In *Proceedings of the International Workshop on Logic Synthesis*, 1993.

- [689] H. Touati, R.K. Brayton and R.P. Kurshan, "Checking Language Containment Using BDDs," In *Proceedings of the International Workshop on Formal Methods in VLSI Design*, January 1990.
- [690] H. Touati, H. Savoj, B. Lin, R. Brayton and A. Sangiovanni-Vincentelli, "Implicit State Enumeration of Finite State Machines using BDDs," In *Proceedings of the International Conference on Computer-Aided Design*, November 1990, pages 130-133.
- [691] H. Trickey, *Compiling Pascal Programs into Silicon*, Technical Report STAN-CS-85-1059, Department of Computer Science, Stanford University, July 1985.
- [692] H. Trickey, "Flamel: A High-Level Hardware Compiler," In *IEEE Transactions on Computer-Aided Design of Circuits and Systems*, Vol.6, March 1987, pages 259-269.
- [693] J.D. Ullman, *Computational Aspects of VLSI*, Computer Science Press, 1984.
- [694] F. Vahid and D.D. Gajski, *Obtaining Functionally Equivalent Simulations Using VHDL and A Time-Shift Transform*, Technical Report #90-21, Department of Information and Computer Science, University of California, Irvine, 1990.
- [695] F. Vahid and D.D. Gajski, "Obtaining Functionally Equivalent Simulations Using VHDL and A Time-Shift Transform," In *Proceedings of the International Conference on Computer-Aided Design*, November 1991, pages 362-365.
- [696] F. Vahid, S. Narayan and D.D. Gajski, *Synthesis from Specifications*, Technical Report #90-03, Department of Information and Computer Science, University of California, Irvine, January 1990.
- [697] F. Vahid, S. Narayan and D.D. Gajski, *SpecCharts, a Language for System-Level Specification*, Technical Report #90-19, Department of Information and Computer Science, University of California, Irvine, July 1990.
- [698] F. Vahid, S. Narayan and D.D. Gajski, "SpecCharts: a VHDL Front-End for Embedded Systems," In *Transactions on Computer-Aided Design of Circuits and Systems*, Vol. 14, No.6, June 1995, pages 694-706.
- [699] F. van Aelten, J. Allen and S. Devadas, "Verification of Relations Between Synchronous Machines," In *Proceedings of the International Conference on*

Computer-Aided Design, 1991, pages 380-383.

[700]

G. van Bochmann and C. Sunshine, "Formal Methods in Communication Protocol Design," In *Transactions on Communications*, Vol.28, No.2, April 1980, pages 624-631.

[701]

P.H.J. van Eijk, C.A. Vissers and M. Diaz, editors, *The Formal Description Technique LOTOS*, North Holland, 1989.

[702]

K.S. van Horn, *An Approach to Concurrent Semantics Using Complete Traces*, Masters Thesis, Computer Science Department, California Institute of Technology, 1986, Technical Report TR-86-5236.

[703]

J.P. van Tassel, *A Formalisation of the VHDL Simulation Cycle*, Technical Report No.249, Computer Laboratory, University of Cambridge, 1992.

[704]

A. van Wijngaarden, B.J. Mailloux, J.L. Peck, C.H.A. Koster, M. Sintzoff, C.H. Lindsey, L.G.L. Meertens and R.G. Fisker, "Revised Report on the Algorithmic Language Algol68," In *Acta Informatica*, Vol.5, 1975, pages 1-236.

[705]

M.Y. Vardi, "A Temporal Fixed Point Calculus," In *Proceedings of the 15th Symposium on the Principles of Programming Languages*, 1988, pages 250-259.

[706]

M.Y. Vardi and P.L. Wolper, "An Automata-Theoretic Approach to Program Verification," In *Proceedings of the Conference on Logic In Computer Science*, 1986, pages 332-334.

[707]

P.M.B. Veiga and M.J.A. Lança, "HARPA: A Hierarchical Multi-Level Hardware Description Language," In *Proceedings of the 21st Design Automation Conference*, June 1984, pages 59-65.

[708]

B. Victor and F. Moller, "The Mobility Workbench - A Tool for the π -Calculus," In *Proceedings of the Conference on Computer-Aided Verification (CAV'94)*, D.L. Dill editor, Lecture Notes in Computer Science, Vol.818, June 1994, pages 428-440.

[709]

J.E. Vuillemin, *Proof Techniques for Recursive Programs*, Ph.D. Thesis, Stanford University, 1973.

[710]

J.E. Vuillemin, "On Circuits and Numbers," In *IEEE Transactions on Computers*, Vol.43, No.8, August 1994, pages 868-879.

- [711] S. T. Vuong, A.C. Lau and R.I. Chan, "Semiautomatic Implementation of Protocols Using Estelle-C Compiler." In *IEEE Transactions on Software Engineering*, Vol. 14, No. 3, March 1988, pages 384-393.
- [712] R. A. Walker and R. Camposano, *A Survey of High-Level Synthesis Systems*, Kluwer Academic Publishers, 1991.
- [713] D.W. Wall, "Experience with a Software-Defined Machine Architecture." In *Transactions on Programming Languages and Systems*, Vol. 14, No.3, July, 1994, pages 299-338.
- [714] M. Wand, "Semantics-Directed Machine Architecture." In *Proceedings of the 9th Symposium on Principles of Programming Languages*, 1982, pages 234-241.
- [715] I. Wegner, "On the Complexity of Branching Programs and Decision Trees for Clique Functions." In *Journal of the ACM*, Vol. 35, No. 2, 1988, pages 461-471.
- [716] P. Wegner, "The Vienna Definition Language," In *Computing Surveys*, Vol. 4, No. 1, 1972.
- [717] G.S. Whitcomb, B. O'Krafka and A.R. Newton, "The Hardware Data-Flow Representation and Synthesis Methodology," In *Proceedings of the ACM/IEEE Workshop on Behavioral Synthesis*, October 1989.
- [718] G.S. Whitcomb and A.R. Newton, "Abstract Data Types and High-Level Synthesis," In *Proceedings of the 27th Design Automation Conference*, June 1990, pages 680-685.
- [719] G.S. Whitcomb and A.R. Newton, *Data-Flow/Event Graphs*, Technical Memo UCB/ERI. M92/24, Electronics Research Laboratory, University of California, Berkeley, March 1992.
- [720] R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S.-W. Liao, C.-W. Tseng, M. Hall, M. Lam and J. Hennessy, "SIUF: An Infrastructure for Research on Parallelizing and Optimizing Compilers," In *ACM SIGPLAN Notices*, Vol.29, December 1994, pages 34-37.
- [721] G. Winskel, *Events in Computations*, Ph.D. Thesis, University of Edinburgh, 1980.
- [722] G. Winskel, "Event Structures," In *Petri Nets: Applications and Relationships to*

Other Models of Concurrency, Advances in Petri Nets 1986, Part II; Proceedings of an Advanced Course, Lecture Notes in Computer Science, Vol.255, 1986, pages 325-392.

- [723] N. Wirth, "The Programming Language Pascal," In *Acta Informatica*, Vol. 1, 1971, pages 35-63.
- [724] W. Wolf, "The FSM Network Model for Behavioral Synthesis of Control-Dominated Machines," In *Proceedings of the 27th Design Automation Conference*, June 1990, pages 692-697.
- [725] W. Wolf, S. Takach, C.-Y. Huang, R. Manno and E. Wu, "The Princeton University Behavioral Synthesis System," In *Proceedings of the 29th Design Automation Conference*, 1992, pages 182-187.
- [726] P. Wolper, "Temporal Logic Can Be More Expressive," In *Proceedings of the 22nd Symposium on the Foundations of Computer Science*, 1981, pages 340-348.
- [727] P. Wolper, "Temporal Logic Can Be More Expressive," In *Information and Control*, Vol. 56, 1983, pages 72-99.
- [728] P. Wolper, "The Tableau Method for Temporal Logic," In *Logique et Analyse*, Vol.28, No. 110, June-Sept 1995, pages 119-136.
- [729] P. Wolper, "On the Relation of Programs and Computations to Models of Temporal Logic," In *Proceedings of the Conference on Temporal Logic In Specification*, B. Banieqbal, H. Barringer and A. Pnueli, editors. Lecture Notes in Computer Science, Vol.398, 1989, pages 75-123.
- [730] F.F. Yao, "Computational Geometry," In *Handbook of Theoretical Computer Science* [327], Vol. A, pages 343-367.
- [731] H. Yasuura and N. Ishiura, "Semantics of a Hardware Design Language for Japanese Standardization," In *Proceedings of the 26th Design Automation Conference*, June 1989, pages 836-839.
- [732] Y. Yasuura and N. Ishiura, "Formal Semantics of UDL/I and its Application to CAD/DA Tools," In *Proceedings of the International Conference on Computer Design*, September 1990, page 90-94.

[733]

S.Y. Yee, *An Esterel to SHIFT Compiler for a Hardware/Software Codesign Environment*. Master's Thesis, Department of Electrical Engineering and Computer Science, University of California, Berkeley, 1994, Technical Memo UCB/ERL M94/??.

[734]

P. Zafiropulo, C.H. West, H. Rudin, D.D. Cowan and D. Brand. "Towards Analyzing and Synthesizing Protocols," In *IEEE Transactions on Communications*, Vol.28, No.4, April 1980, pages 651-661.

[735]

R.E. Ziemer, W.H. Tranger and D.R. Fanin, *Signals and Systems: Continuous and Discrete*, Macmillan Publishing, 1989.

A The Assembly Language of the NDAM

The following sections provide a complete definition for the Non-Deterministic Abstract Machine (NDAM) assembly language. The presentation here is of a syntactic and structural nature with the main body, and especially Chapter 7, defining the computational semantics of the NDAM.

A.1 Structure Description

A system description is made up of a network declaration and its associated process tree. The network and its processes are depicted in Figure A-1. Since each process can contain other processes, a tree-like structure is obtained. There is exactly one network definition which defines the system. Within the network there are declarations for types and signals. Some subset of these signals are labeled as the system's inputs and another disjoint subset is labeled as the system's output.

A.1.1 The Network

The network is a description of all of the top-level types and signals used in the system. The network declaration is:

```
network N(NAME) is  
  declarations  
  processes-tree  
end network N(NAME)
```

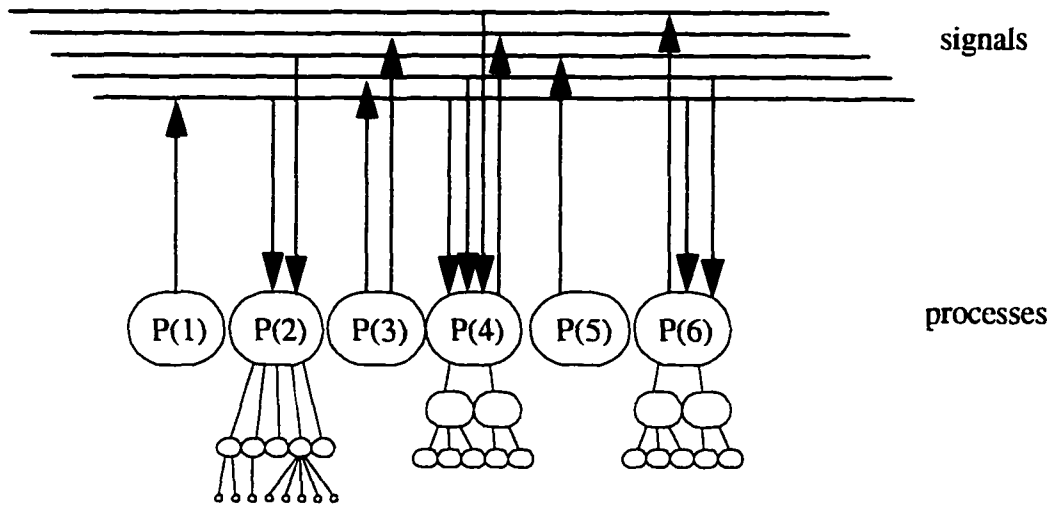


Figure A-1. Processes in a Network Communicating by Signals

The declarations consist of the type and signal declarations. These declarations are visible throughout the process tree. The declaration of the types at the top-level is needed to ensure that the types referenced in signal declarations can be mentioned before their use.

The process tree portion is a series of declarations mentioning process names which defines the structure of the process tree. The declarations run according to one of the two forms:

```

top:      { P(NAME) }
P(NAME):  { P(NAME) }

```

The first form declares the top of the process tree stating that the top of the tree contains the processes mentioned on the right-hand side. There can be only one such declaration.

The second form defines the internal branches of the process tree. The left-hand side defines the parent and the right-hand the children of that parent. A process can only be mentioned on the left-hand side once in the network. The processes on the right-hand side are children of that parent and must appear in some *TCWC* instruction in the parent. A

process can only be mentioned in one right-hand side in the network. These two restrictions ensure that the process tree declarations actually form a tree.

The leaf processes in the process tree are declared by being mentioned in a right-hand side and no left-hand side.

Example:

The example of Figure A-2 defines a network with three signals and four processes.

```

network N(1) is
type t(unit) 1
type t(bool) 2
type t(int5) 5
signal S(tick) t(unit)
signal S(go) t(bool)
signal S(val) t(int5)
top: P(1), P(2)
P(1): P(3), P(4)
P(4): P(5), P(6), P(7)
end network N(1)

```

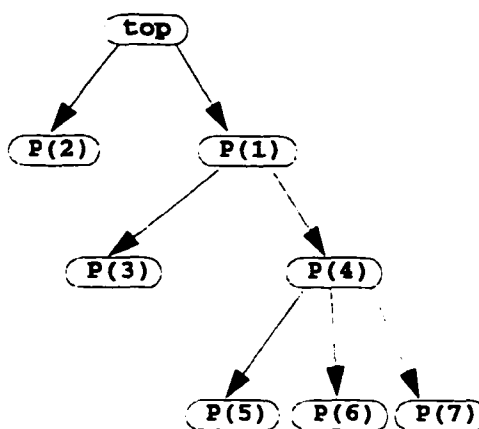


Figure A-2. An Network Declaration and Its Process Tree

A.2 The Process

The process is the fundamental unit of structure in the network. The process is the unit of execution and also defines the unit of concurrency. The process is also the fundamental structure onto which watchdog guards and exceptions are built. These aspects are treated in the section on the **TCWC** instruction in Section A.4.8. A process is declared as follows:

```

process P(NAME) is
declarations-and-instructions
end process P(NAME)

```

The **P(NAME)** must appear in the process tree declaration of the containing network. By virtue of that mention, the unique position of the process in the process tree is known. A

process contains some declarations which define the types, signals, registers, exceptions and sensors that are visible to it and its children. The set of types, signals, registers, exceptions and sensors visible in a process are those declared in any of its parents and including the network. A declaration of any of these entities within the process shadows a more global declaration.

The process may also define some counters to be used in counted signal guards of a **TCWC** instruction. These counters are visible only within the process.

For all but the top level processes there must be at least one **interface** declaration that indicates the points at which the process can be called by its parent. The top level processes are assumed to commence execution at the first executable instruction in their code body.

The process body is made up of one or more executable instructions. The set of instructions is presented in Section A.3.

Example

An example of two processes are shown in Figure A-3

```
process P(1) is
type T(unit) 1
signal S(this) T(unit)
constant R(0) t(unit) := 0
signal S(that) T(unit)
L(1): wait S(this)
L(2): call P(2) I(0)
      goto L(1)
end process P(1)

process P(2) is
interface I(0) L(1)
L(1): wait S(this)
L(2): emit s(that) R(0)
      goto L(1)
end process P(2)
```

Figure A-3. Two Simple Processes

A.3 Data Declarations

There are three classes of declarations: type declarations, counter declarations and storage declarations. Of the storage declarations these can be further broken down into the purpose of the storage: signal, exception, register and sensor.

A.3.1 Type Declarations

The type declaration introduces the name of a type and defines how many values that type may take on. The effect of type declarations is to declare a class of multi-valued variable. There are two variants:

```
type T(NAME) N  
type T(NAME) range V1 V2
```

The first defines a type domain that has N possible values. Equivalently, this can be thought of as declaring an unsigned integer domain that ranges from $0 . . N-1$. The second defines a type domain that ranges over values in the range $V1 . . V2$.

The type declaration does not offer a way of naming any of the values of the multi-valued variable. Such naming is considered to be an artifact of a high-level language and is best left at that level. A constant register declaration may associate names with the values of the type.

In the example of Figure A-4, one singleton type is declared along with multi-valued types. The type `t(unit)` contains only one possible value: 0. It would be used in an Esterel compiler as the type for a pure signal or exception. Of the other two types, the second has two possible values, 0 and 1; the third has eight possible values ranging from 0 to 7.

A type with values ranging from $V1$ on the low end to $V2$ on the high end can be declared. The value $V1$ must be no greater than $V2$ so that a non-vacuous range is defined.

```

type t(unit) 1
type t(bool) 2
type t(mv8) 8

```

Figure A-4. Some Commonly-Used Type Declarations

In the declarations of Figure A-5, two multi-valued types are declared: one type has thirty-two possible values, ranging from -16 to 15; the other has four billion possible values.

```

type t(int5) -16 15
type t(int32) -2147483648 2147483647

```

Figure A-5. Type Declarations Using a Bounded Range

A.3.2 Counter Declarations

Counter declarations are local to the process in which they are declared. This is unlike type declarations and storage declarations which are visible both in the process and all child processes. A counter declaration is only referenced in a **TCWC** instruction to manage the counting of a counted signal occurrence. The declaration follows the form:

```

counter C(NAME) INTEGER

```

This declares a counter to have **INTEGER** as its initial value whenever it is used in a **TCWC** instruction. See also Section A.3.x on the watching clause of the **TCWC** instruction.

A.3.3 Storage Declarations

Storage declarations define objects in a process such as registers, signals, exceptions and sensors.

A.3.3.1 Definition Classes

Definition classes identify commonly-used data patterns on storage declarations. These are singleton variables, records and arrays. The three definition classes are common to all

four storage classes so it is worth treating them orthogonally. All follow the general style of:

```
class ?(NAME)
```

where *class* is one of **register**, **signal**, **exception**, or **sensor** and ?(*NAME*) is the name of the object with ? being the relevant name-key character **R**, **S**, **E**, and **X** respectively.

Singletons

The fundamental unit of declaration is the singleton with each singleton unit of storage being associated with a type. The type defines how much information can be stored in that slot (how many bits are required).

```
class ?(NAME) T(NAME) [ := V ]
```

A singleton of the *class* is declared; it contains one field. The storage slot can optionally be initialized with a value at the point of declaration. If no initializer value is declared the default initialization value of zero is assumed.

Arrays

Arrays, though declared as a multidimensional entity are always indexed with a single integer quantity.

```
class ?(NAME) T(NAME) D, D, ... D [ := V, V, ... V ]
```

An array of the *class* is declared; it contains as many fields as dictated by the "volume" of the dimensions. These fields can be optionally initialized at the point of declaration. If initializer values are given then the number of initializers must be no greater than the number fields; uninitialized fields receive the default initialization value of zero.

Arrays provide a form of *computed name* that can be cogently handled in the symbolic formalism of the transition relation. Were unbound pointers allowed on the machine then

the transition relation corresponding to a pointer indirection operation would be have to take into account the possibility that the pointer value could be any possible data location.

Records

Records are treated in largely the same way as arrays on the machine. A record of the class is declared; it contains a number of fields. The fields can be optionally initialized at the point of declaration. If initializer values are given then the number of initializers must be no greater than the number fields; uninitialized fields receive the default initialization value of zero.

```
class ?(NAME) T(NAME), T(NAME), ... T(NAME) [ :=  
    V, V, ... V ]  
class ?(NAME) T(NAME) [ := V ], T(NAME)  
    [ := V ], ... T(NAME) [ := V ]
```

The major difference between an array and a record is that a record-classed storage is made up of slots that can differ in their type. As with the array case, there is a distinct assign-from-field and assign-to-field instruction. In contrast with the array case, the *field* value is a constant so it is not possible to construct a pointer-to-a-field.

A.3.3.2 Register Declarations

There are three kinds of registers, constant registers, temporary registers and permanent registers. It is most useful however to consider registers as belonging to one of two kinds: non-constant and constant registers of which the non-constant registers are further divided into temporary registers and permanent registers. The distinction between the temporary registers and the permanent registers is that the temporaries are understood to not persist across an instant boundary. Their use is thus restricted to intra-instant computations. Therefore they could conceivably be allocated from some temporary space such as a stack. In the verification context, they could be smoothed away to reduce the state space of the problem.

Non-constant registers are suitable for use in representing variables, temporary or otherwise, which are to be operated upon using the various assignment operators. They can appear on the left-hand side of an assignment.

Constant registers on the other hand are suitable for representing constants such as the members of a scalar type. Temporary registers are intended to be used to store (compiler-generated) values that do not need to be saved across instant boundaries; thus temporary registers may be added and destroyed with impunity by the compilation process. The permanent register declarations are:

```

register R(NAME) T(NAME) [ := V ]
register R(NAME) T(NAME), T(NAME), ... T(NAME) [ :=
    V, V, ... V ]
register R(NAME) T(NAME) [ := V ], T(NAME) [ := V ],
    ... T(NAME) [ := V ], T(NAME) [ := V ]
register R(NAME) T(NAME) D, D, ... D [ := V, V, ... V ]

```

The temporary registers cannot be initialized with a value because their values do not persist across instant boundaries. They must be assigned to within the instant before their values are referenced. Their declarations are:

```

temporary R(NAME) T(NAME)
temporary R(NAME) T(NAME), T(NAME), ... T(NAME)
temporary R(NAME) T(NAME) D, D, ... D

```

Constant registers can never be assigned. Thus they must be given an initial value with their declaration. The constant register declarations are:

```

constant R(NAME) T(NAME) := V
constant R(NAME) T(NAME), T(NAME), ... T(NAME) :=
    V, V, ... V
constant R(NAME) T(NAME) := V, T(NAME) := V,
    ... T(NAME) := V, T(NAME) := V
constant R(NAME) T(NAME) D, D, ... D := V, V, ... V

```

Registers, constant, temporary and permanent can be declared at any level of the process tree. Such registers are visible both within the declaring process and in its children unless their visibility is occluded by a more local declaration.

An example of the various sorts of register declarations is shown in Figure A-6.

```

type t(bool) 2
type t(int32) 4294967296
register r(a_bit) t(bool) := 1
constant r(r32) t(int32) := -9000233

register r(a_rec) t(bool), t(int32), t(bool) :=
    1, -9000233, 1
temporary r(three_r32) t(int32), t(int32), t(int32)

constant r(a_rec) t(bool) := 1,
    t(int32) := -9000233, t(bool) := 1
constant r(three_r32) t(int32) := 1,
    t(int32) := 2, t(int32) := 3

type t(frog) 7
temporary R(fig) T(frog) 1, 2, 3

constant r(an_array32) t(bool) 32 :=
    0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 1

type t(3) 3
register r(r32_3by3) t(3) 3, 3 :=
    1, 2, 3, 4, 5, 6, 7, 8, 9

```

Figure A-6. The Various Sorts of Register Declaratoins

A.3.3.3 Signal Declarations

Signal declarations are given according to the storage declaration form. They cannot have initial values. Their declarations are:

```

signal S(NAME) T(NAME)
signal S(NAME) T(NAME), T(NAME), ... T(NAME)
signal S(NAME) T(NAME) D, D, ... D

```

Signals can be declared at any level of the process tree. Such signals are visible both within the declaring process and in its children unless their visibility is occluded by a more local declaration. Signals declared at the network level are visible to all processes. Examples of signal declarations are shown in Figure A-7.

```

type t(pure) 1
type t(bit) 2
type t(reg32) 4294967296
signal S(tick) t(pure)
signal S(lovec) t(reg32), t(reg32), t(reg32)
signal S(dsp-dim3) t(bit) 3, 3, 3

```

Figure A-7. The Various Sorts of Signal Declaration

A.3.3.4 Exceptions Declarations

Exception declarations are given according to the storage declaration form. They cannot have initial values as their value is defined at the time the exception is raised and is lost at the end of the instant. Their declarations are:

```

exception E(NAME) T(NAME)
exception E(NAME) T(NAME), T(NAME), ... T(NAME)
exception E(NAME) T(NAME) D, D, ... D

```

Exceptions can be declared at any level of the process tree. Such exceptions are visible both within the declaring process and in its children unless their visibility is occluded by a more local declaration. Exceptions may only be referenced in a **TCWC** instruction or in a **raised** instruction in the process in which they were declared. A **raise** instruction may mention any visible exception.

A.3.3.5 Sensor Declarations

Sensor declarations are given according to the storage declaration form. They cannot have initial values as their value is defined only as it is written or read. Their declarations are:

```

sensor X(NAME) T(NAME)
sensor X(NAME) T(NAME), T(NAME), ... T(NAME)
sensor X(NAME) T(NAME) D, D, ... D

```

Sensors can be declared at any level of the process tree. Such sensors are visible both within the declaring process and in its children unless their visibility is occluded by a

more local declaration. Sensors may also be declared at the network level. Such sensors are visible to all processes.

A.4 Executable Instructions

The set of executable instructions can be broadly broken up into nine classes:

- Assignment Operations
- Unary Operations
- Binary Operations
- Control Transfers
- Signal Operations
- Exception Operations
- Sensor Operations
- Try Call Watching Catching
- Process Control

These are covered in the subsequent sections.

A.4.1 Assignment Operations

Assignment operations move information from one register to another. They are exclusively a register-to-register operation. The various flavors of assignment deal with moving information into and out of register arrays (member assignment using a dynamically-computed index) and moving information into and out of register records (field assignment using a statically-named index) and aggregate assignments between registers of the same type.

The notation used in this section as well as the following sections on unary and binary operations is that the two sides to an assignment operation are denoted by *LHS* for the left-hand side and *RHS* for the right-hand side.

Of course, only non-constant registers may appear on the left-hand side of an assignment while either constant or non-constant register references may appear on the right-hand side. This rule is driven by the common sense that one must not destroy copies of constants; this is a statically checkable condition.

R(LHS) := R(RHS)

An assignment is made from the register **R(RHS)** to the register **R(LHS)**. The assignment is independent of the size of the registers though the number of singletons in an array or record must match on both sides. The types of the registers do not need to match; the bits are moved from the left-hand side to the right-hand side with truncation occurring for transfers into a smaller domain.

The assignment must be between either two singleton registers, two record registers with the same number of fields or two array registers with the same volume. This is a statically-checkable condition.

R(LHS) := R(RHS).offset

An assignment is made from the field offset of the register **R(RHS)** to the register **R(LHS)**. The register **R(LHS)** must be a singleton as the assignment only references a single field of **R(RHS)**. The offset must be an unsigned integer which is within the bounds of the number of fields in the register **R(LHS)**. This is a statically-checkable condition.

R(LHS).offset := R(RHS)

An assignment is made to the field offset of the register **R(LHS)** from the register **R(RHS)**. The register **R(RHS)** must be a singleton as the assignment only references a single field of **R(LHS)**. The offset must be an unsigned integer which is within the bounds of the number of fields in the record register **R(LHS)**. This is a statically-checkable condition.

R(LHS) := R(RHS) [R(E)]

An assignment is made from the member indicated by the value of **R(E)** of the array register **R(RHS)** to the register **R(LHS)**. The register **R(LHS)** must be a singleton as the assignment only references a single member of **R(RHS)**. This is not a statically-checkable condition unless **R(E)** is a constant register.

R(LHS) [R(E)] := R(RHS)

An assignment is made to the member indicated by the value of **R(E)** of the array register **R(LHS)** from the register **R(RHS)**. The register **R(RHS)** must be a singleton as the assignment only references a single field of **R(LHS)**. The **R(E)** must be an unsigned integer which is within the bounds of the number of members

in the array register **R (LHS)**. This is not a statically-checkable condition unless **R (E)** is a constant register.

R (LHS) := R (RHS) [R (LOW) , R (HIGH)]

An assignment is made from the member indicated by the values of **R (LOW)** to **R (HIGH)** of the array register **R (RHS)** to the register **R (LHS)**. The register **R (LHS)** must be an array as the assignment references the multiple members of **R (RHS)**. This is not a statically-checkable condition unless **R (LOW)** and **R (HIGH)** are constant registers.

R (LHS) [R (LOW) , R (HIGH)] := R (RHS)

An assignment is made to the array indicated by the values in the range of **R (LOW)** to **R (HIGH)** of the array register **R (LHS)** from the register **R (RHS)**. The register **R (RHS)** must be an array as the assignment references the multiple fields of **R (LHS)**. The **R (LOW)** and **R (HIGH)** must be an unsigned integer which is within the bounds of the number of members in the array register **R (LHS)**. This is not a statically-checkable condition unless **R (LOW)** and **R (HIGH)** are constant registers.

A.4.2 Unary Operations

The unary operations are defined by a destination register called **R (LHS)** and a single source register called **R (RHS)**. There is also an operation *op* which is performed on the value of **R (RHS)** as it is transferred. The following sections describe the possible unary operation statements.

For the unary operations, the **R (RHS)** and **R (LHS)** registers may be any a singleton, record or array register, but both registers must match with respect to type, storage class and in size, for record registers, or volume for array registers. This condition is statically checkable.

In the case of aggregate registers, the unary operation is performed element-wise on all the members of the aggregate. Thus the sense of the unary operator acting upon a record or array register is that of a vector operation.

R(LHS) := abs R(RHS)

The absolute value of **R(RHS)**, interpreted as a 2s-complement number, is placed in **R(LHS)**.

R(LHS) := dec R(RHS)

The value of **R(RHS)**, interpreted as a 2s-complement number, decremented and placed in **R(LHS)**.

R(LHS) := inc R(RHS)

The value of **R(RHS)**, interpreted as a 2s-complement number, incremented and placed in **R(LHS)**.

R(LHS) := not R(RHS)

The value of **R(RHS)** is interpreted as an unsigned integer is complemented and placed in **R(LHS)**. The complementation performed is a logical one, so any non-zero value is transformed to 0 while the 0 value is transformed to 1. The type domain of **R(LHS)** should contain more than one member for this to be meaningful.

R(LHS) := ~ R(RHS)

The bits of **R(RHS)** are complemented and placed in **R(LHS)**.

R(LHS) := - R(RHS)

The value of **R(RHS)**, interpreted as a 2s-complement number, negated and placed in **R(LHS)**.

A.4.3 Binary Operations

The binary operations are defined by a destination register called **R(LHS)** and a two source registers called **R(RHS-1)** and **R(RHS-2)**. There is also an operation *op* which is performed on the values in **R(RHS-1)** and **R(RHS-2)** as it is transferred. The following sections describe binary operation instructions.

For the binary operations, the **R(RHS-1)**, **R(RHS-2)** and **R(LHS)** registers may be a singleton, record or array register, but both registers must match with respect to storage class and in size. For record registers this means the number of fields must match and for array registers this implies the volumes must match. This condition is statically checkable.

In the case of aggregate registers, the binary operation is performed element-wise on all the members of the aggregate. Thus the sense of the binary operator acting upon a record or array register is that of a vector operation.

$R(LHS) := R(RHS-1) \text{ and } R(RHS-2)$

The value of $R(RHS-1)$ and $R(RHS-2)$ are combined via logical-and and placed in $R(LHS)$. The operation performed is a logical one so nonzero values are transformed to 0 while the 0 value is transformed to 1.

$R(LHS) := R(RHS-1) \text{ or } R(RHS-2)$

The value of $R(RHS-1)$ and $R(RHS-2)$ are combined via logical-or and placed in $R(LHS)$. The operation performed is a logical one so nonzero values are transformed to 0 while the 0 value is transformed to 1.

$R(LHS) := R(RHS-1) \text{ nand } R(RHS-2)$

The value of $R(RHS-1)$ and $R(RHS-2)$ are combined via logical-nand and placed in $R(LHS)$. The operation performed is a logical one so nonzero values are transformed to 0 while the 0 value is transformed to 1.

$R(LHS) := R(RHS-1) \text{ nor } R(RHS-2)$

The value of $R(RHS-1)$ and $R(RHS-2)$ are combined via logical-nor and placed in $R(LHS)$. The operation performed is a logical one so nonzero values are transformed to 0 while the 0 value is transformed to 1.

$R(LHS) := R(RHS-1) \text{ xor } R(RHS-2)$

The value of $R(RHS-1)$ and $R(RHS-2)$ are combined via logical-xor and placed in $R(LHS)$. The operation performed is a logical one so nonzero values are transformed to 0 while the 0 value is transformed to 1.

$R(LHS) := R(RHS-1) = R(RHS-2)$

The value of $R(RHS-1)$ and $R(RHS-2)$ are combined via a comparison for equality, and placed in $R(LHS)$. A value of 1 results if the comparison is true; a value of 0 results if the comparison operation fails.

$R(LHS) := R(RHS-1) \neq R(RHS-2)$

The value of $R(RHS-1)$ and $R(RHS-2)$ are combined via a comparison for inequality, and placed in $R(LHS)$. A value of 1 results if the comparison is true; a value of 0 results if the comparison operation fails.

$R(LHS) := R(RHS-1) < R(RHS-2)$

The value of $R(RHS-1)$ and $R(RHS-2)$ interpreted as 2s-complement integers are combined via a less-than comparison, and placed in $R(LHS)$. A value of 1 results if the comparison is true; a value of 0 results if the comparison operation fails.

$R(LHS) := R(RHS-1) > R(RHS-2)$

The value of $R(RHS-1)$ and $R(RHS-2)$ interpreted as 2s-complement integers are combined via an integer greater-than comparison, and placed in $R(LHS)$. A value of 1 results if the comparison is true; a value of 0 results if the comparison operation fails.

$R(LHS) := R(RHS-1) <= R(RHS-2)$

The value of $R(RHS-1)$ and $R(RHS-2)$ interpreted as 2s-complement integers are combined via an integer less-than or equal comparison, and placed in $R(LHS)$. A value of 1 results if the comparison is true; a value of 0 results if the comparison operation fails.

$R(LHS) := R(RHS-1) >= R(RHS-2)$

The value of $R(RHS-1)$ and $R(RHS-2)$ interpreted as 2s-complement integers are combined via a greater-than or equal comparison, and placed in $R(LHS)$. A value of 1 results if the comparison is true; a value of 0 results if the comparison operation fails.

$R(LHS) := R(RHS-1) + R(RHS-2)$

The value of $R(RHS-1)$ and $R(RHS-2)$ interpreted as 2s-complement integers are combined via integer addition, and placed in $R(LHS)$. Overflow is ignored.

$R(LHS) := R(RHS-1) - R(RHS-2)$

The value of $R(RHS-1)$ and $R(RHS-2)$ interpreted as 2s-complement integers are combined via integer subtraction, and placed in $R(LHS)$. Overflow is ignored.

$R(LHS) := R(RHS-1) * R(RHS-2)$

The value of $R(RHS-1)$ and $R(RHS-2)$ interpreted as 2s-complement integers are combined via integer multiplication, and placed in $R(LHS)$. Overflow is ignored.

Depending on the bit width required for the operation, there may be practical problems associated with representing the transition relation of this instruction.

$R(LHS) := R(RHS-1) / R(RHS-2)$

The value of $R(RHS-1)$ and $R(RHS-2)$ interpreted as 2s-complement integers are combined via integer division, and placed in $R(LHS)$.

Depending on the bit width required for the operation, there may be practical problems associated with representing the transition relation of this instruction.

It is an error for the value of $R(RHS-2)$ to be the value 0 and so instances of operations where there is a possibility of such a situation should be protected by the appropriate branch instructions.

In a synthesis context a divide by zero has undefined consequences. In a verification context, a divide by zero implies that $R(LHS)$ has an undefined value (i.e. all possible values).

$R(LHS) := R(RHS-1) \bmod R(RHS-2)$

The value of $R(RHS-1)$ and $R(RHS-2)$, treated as 2s-complement integers are combined via integer the modulus operation, and placed in $R(LHS)$.

The rules for division apply to this instruction as well.

$R(LHS) := R(RHS-1) \bmod R(RHS-2)$

The value of $R(RHS-1)$ and $R(RHS-2)$, treated as 2s-complement integers are combined via integer remainder operation, and placed in $R(LHS)$.

The rules for division apply to this instruction as well.

A.4.4 Control Transfers

Control-transfer instructions modify the program counter, possibly based on some condition. They should be thought of as instructions which conditionally (or not) assign to the program counter, thereby dictating the next control-state of the abstract machine.

$\text{goto } L(TARGET-1), L(TARGET-2), \dots L(TARGET-N)$

An unconditional branch is performed the target. In the nondeterministic case, the successor pc of the machine is one of the targets. If there is only one target then the goto is a deterministic branch; if there is more than one branch then the goto is considered to be a nondeterministic branch.

The example of Figure A-8 shows the declaration of three paths which are non-deterministically executed. The first goto is nondeterministic and declares three possible targets $L(\text{left})$, $L(\text{middle})$ and $L(\text{right})$. Subsequently for each target an instruction

performs some work, either incrementing, decrementing or taking the absolute value of the previously-declared register $r(1)$. A deterministic goto is used to continue at common point labeled $L(\text{end})$.

```

goto L(left), L(middle), L(right)
L(left):   r(1) := inc r(1)
           goto L(end)
L(middle): r(1) := dec r(1)
           goto L(end)
L(right):  r(1) := abs r(1)
           goto L(end)
L(end):    null

```

Figure A-8. A Nondeterministic Goto Instruction

if [not] R(TEST) goto L(TARGET)

A conditional branch is performed depending on the binary value of $R(\text{TEST})$. If the value is positive then the goto is affected; if the value is zero then control passes to the next statement. The keyword **not** inverts the sense of this test. Of note, the type of $R(\text{TEST})$ must be binary in the sense that the type must have at least the range 0 to 1.

The example of Figure A-9 shows a type $t(\text{bool})$ which is declared to have an appropriate range, “from 0 to 1” for use in an if statement. This type is then used to declare a temporary register $r(\text{tmp})$ which is in turn used to store the result of a comparison between two previously-declared registers $r(1)$ and $r(2)$. The core of the example is the use of the if statement in both phases to test the value of the comparison. As there is no possibility of executing beyond the if statements an exit can be used to declare that the fall-through path is unexecutable.

```

type t(bool) 2
temporary r(tmp) t(bool)
r(tmp) := r(1) = r(2)
if r(tmp) goto L(good)
if not r(tmp) goto L(bad)
exit

```

Figure A-9. A Code Fragment Using the If Instruction

```

case R(KEY)
when V-1-1, V-1-2, ... V-1-N goto L(TARGET-1);
when V-2-1, V-2-2, ... V-2-N goto L(TARGET-2);
...
when V-M-1, V-M-2, ... V-M-N goto L(TARGET-N)

```

A multi-way branch is performed based on the value in **R (KEY)**. Conceptually the **R (KEY)** is matched against the **V-i-j** and whichever value is equal, then the goto of that when clause is taken; if none match then execution continues with the following statement.

All of the **V-i-j** are required to be in the range of the type of the **R (KEY)** and each **V-i-j** must be unique to ensure that the statement is deterministic. The **V-i-j** can be either integer literals or constant registers. They may not be non-constant registers.

The example of Figure A-10 shows a small fragment of code which acquires the current selection on the signal **s (value)** and uses the value as the key for a case statement. The case statement targets one of three possible outcomes **L(low)**, **L(middle)** and **L(redo)** which are not described. Finally an exit is used to indicate that falling through the case statement to the subsequent statement is impossible.

```

temporary r(key) t(type)
constant r(EXIT) t(type) := 3
constant r(FAIL) t(type) := 4
r(key) := selection s(value)
case r(key)
when 0, 1, 2 goto L(low);
when R(EXIT) goto L(high);
when R(FAIL) goto L(redo)
exit

```

Figure A-10. A Code Fragment Using the Case Instruction

```

present [ not ] S(NAME) goto L(TARGET)

```

A conditional branch is performed depending on whether the signal **S (NAME)** is present in the current instant. If so then the transfer is affected; if not then control passes to the next statement. The keyword *not* inverts the sense of this test. Of note, the type of **S (NAME)** can be arbitrary as its value is not referenced; only its presence or absence is detected.

All flow paths leading to the **present** instruction must have the property that the instruction is preceded by either a **require** or an **emit**. The presence or absence of the mentioned signal must be known at the instruction's execution.

The example of Figure A-11 shows a type **t (any)** which is declared to have an some appropriate range. This type is then used to declare a signal **s(some)** onto which is emitted a constant **r(1)** which was presumably declared previously. The core of the example is the use of the presence statement in both phases to test for the presence or absence of the signal. As there is no possibility of executing beyond the present statements an **exit** is used to declare that the fall-through path is unexecutable.

```

type t(any) is
signal s(some) t(any)
  emit s(some) := r(1)
  require s(some)
  present s(some) goto L(here)
  if not s(some) goto L(there)
  exit

```

Figure A-11. A Code Fragment Using the Present Instruction

```

select
when sexp-1-1, sexp-1-2 ... sexp-1-M goto L(NAME-1);
when sexp-2-1, sexp-2-2 ... sexp-2-M goto L(NAME-2);
...
when sexp-N-1, sexp-N-2 ... sexp-N-M goto L(NAME-N);

```

The **select** instruction is a multi-way branch that is related to the **present** instruction in the same way that the **if** instruction is generalized to **case**.

The **sexp** is an conjunctive expression denoting the presence or absence of a set of signals. Each **sexp-i-j** tests that a specific set of signal event occurred in the current instant.

In the **sexp-i-j** syntax, presence is denoted by **S (NAME)** - the signal name appearing by itself. Absence is denoted by **~S (NAME)** - the signal name negated. Conjunction is denoted with the ***** symbol with signal expressions formed as per the example of Figure A-12.

The **select** is deterministic. If more than one mentioned **sexp-i-j** matches for the current instant then control is transferred via the **when** clause mentioned first.

As with the **present** instruction, all flow paths leading to the **select** instruction must have the property that the instruction is preceded by either a **require** or an **emit**. The presence or absence status of all signals mentioned in the **select** must be known at the instruction's execution.

```
S(this) * S(that) * ~S(the-other)
~S(this) * ~S(that) * ~S(the-other)
```

Figure A-12. Some Example Signal Expressions

The example of Figure A-13 is a generalization of the previous example that used the **present** instruction.

```
type t(any) 18
signal s(some) t(any)
signal s(more) t(any)
  emit s(some) := r(1)
  require s(some)
  select
  when s(some)*~s(more) goto L(here);
  when ~s(some)*s(more) goto L(there)
  exit
```

Figure A-13. A Code Fragment Using the '**select**' Instruction

A.4.5 Signal Operations

The signal operations provide the means by which the values in the registers of the process are assigned to signals and by which the presence and value components of signals are recovered into the registers of a process.

```
emit S(LHS) R(RHS-1), R(RHS-2), ... R(RHS-N)
```

A set of possible values is emitted onto the signal **S(LHS)**. The value of **S(LHS)** is then, nondeterministically, one of the **R(RHS-I)**. The **selection** statement

can be used to reference this non-deterministic value. A deterministic emission is simply an emission where there is but one **R (RHS)**.

Examples:

The first example is a deterministic emission which places the value currently found in the register **r(1)** onto the signal **s(result)**. The second example is a nondeterministic emission which places the value of both **r(bad)** and **r(good)** onto **s(ok)**.

```
emit s(result) r(1)
emit s(ok) r(bad), r(good)
```

Figure A-14. A Code Fragment the Emit Instruction

undef S (LHS)

The signal **S (LHS)** is reset to its unknown state. This instruction is used for implementing local signals.

The use of **undef** must be carefully controlled for if not used to clear the internal state of local it will destroy the single assignment property of signals and thereby corrupt the modularity of the computational semantics.

R(LHS) := selection S (RHS)

The selection of **S (RHS)** is assigned to **R (LHS)**. In the deterministic case, this merely represents extracting the current value of the signal **S (RHS)** and placing that value in **R (LHS)**.

The signals value (and thus its presence or absence) must be known at the instruction's execution. All flow paths leading to the **selection** instruction must have the property that instruction they are preceded by either a **require** or an **emit**.

R(LHS) := presence S (RHS)

The presence of **S (RHS)** is assigned to **R (LHS)**.

The rules for **selection**, apply to this instruction as well. The presence or absence of **S (RHS)** must be known at the instruction's execution.

require S(RHS-1), S(RHS-2), ... S(RHS-N)

The execution of the containing process suspends until the value and presence/absence status of the **S(RHS-I)** are defined.

This is a synchronization instruction and either it or an **emit** must precede every explicit signal reference instruction (**selection, presence** or **present**).

A.4.6 Exception Operations

Exceptions are provided explicitly through the notion of subprocesses. The granularity of exceptions is at the level of the subprocess. Exceptions are handled across the subprocess call boundary through the **handle**-clause of a call instruction. Exceptions are raised in subprocesses by naming the exception name and parent process to which control is to be transferred.

The value recovered in the handling parent is transferred also. In this sense an exception name is a register through which the exception-raising subprocess and the exception-handling parent process may communicate a value. The exception value only persists from the **handle** clause of the **TCWC** to the end of the instant. In this sense, exception registers are more akin to temporaries. The exception value must be explicitly read from the exception register space into the true register space if the raised value is to be preserved.

raise E(NAME) R(NAME)

The exception **E(NAME)** that is visible in the process tree is raised and the value contained in **R(NAME)** is assigned to the exception register of the process where the exception declaration appears. The storage class (singleton, record, array) of **E(NAME)** and **R(NAME)** must match.

R(NAME) := raised E(NAME)

The value in the exception register **E(NAME)** is transferred to the register **R(NAME)** which may be either a temporary or permanent register (but of course may not be a constant register) and must have the same storage class (singleton, record, array) as **E(NAME)**.

This instruction is only valid within the process where **E(NAME)** is declared and it must appear on a flow path from the **handle** subclause of the **TCWC**.

A.4.7 Sensor Operations

Sensors are defined in their own name space, **X (NAME)** to make it clear that the sensor values have nothing to do with either the state space of the process or with the event- semantics of the process. The letter **X** was chosen to indicate that the value of a sensor is unknown and unpredictable. The value of a sensor can thus not be predicted by dataflow analysis and thus imposes constraints on the high-level optimizations which can be performed around it. Sensors are supported by two instructions:

write X(NAME) R(NAME)

The value in **R (NAME)** is written to the sensor **X (NAME)**; the storage class of **X (NAME)** and **R (NAME)** must match.

R(NAME) := read X(NAME)

The value of the sensor **X (NAME)** is read into the register **R (NAME)**. **R (NAME)** must be a temporary or permanent register (not a constant register of course) and must have the same storage class (singleton, record, array) as **X (NAME)**.

A.4.8 Try Call Watching Catching (TCWC)

The fundamental unit of process control is the “*Try Call Watching Catching,*” or **TCWC** instruction. While all the other instructions are conceptually simple, this one instruction bears the burden of three features: concurrent subprocess execution, guarding and pre-emption.

```

try
  call P(NAME-1) at I(NAME-1);
  call P(NAME-2) at I(NAME-2);
  ...
  call P(NAME-L) at I(NAME-L);
watching
  when C(NAME-1) S(NAME-1) goto L(NAME-w1);
  when C(NAME-2) S(NAME-2) goto L(NAME-w2);
  ...
  when C(NAME-M) S(NAME-M) goto L(NAME-cM);
catching
  handle E(NAME-1) goto L(NAME-c1);
  handle E(NAME-2) goto L(NAME-c2);
  ...
  handle E(NAME-M) goto L(NAME-cM)

```

The behavior of the **TCWC** is described in the following sections. Both the **catching** and **watching** clauses are optional.

A.4.8.1 Calling a Child

The call clauses provide for control transfers between a parent process and one or more child processes. The singleton form without all the decoration for guarding and preemption is simply:

```

  call P(NAME) I(NAME)

```

which entails a control flow from the current **TCWC** instruction to the interface **I(NAME)** of the child process **P(NAME)**. **P(NAME)** must be a child process of the containing process as declared in the network's process tree.

A.4.8.2 Parallelism

More generally, the variant of the **TCWC** that invokes multiple subprocesses at one time is stated as:

```

try
  call P(NAME-1) I(NAME-1);
  call P(NAME-2) I(NAME-2);
  ...
  call P(NAME-N) I(NAME-N)

```

The semantics of this instruction is that the named subprocesses commence executing in the current instant at the indicated interfaces. The **TCWC** blocks until all of the subprocesses have terminated. At that point, control is passed to the next instruction.

A.4.8.3 Guarding - Watching For Signals

The watching-clause of the **TCWC** instruction declares that the invoked subprocesses are interrupted in the current instant if an event on any of the indicated signals occurs and its associated counter is decremented zero. The counter mention is optional with a default count of 1 being assumed. The form of this variant of the call instruction is as follows:

```
try
  ...call-clauses...
watching
  when C(NAME-1) S(NAME-1) goto L(NAME-1);
  when C(NAME-2) S(NAME-2) goto L(NAME-2);
  ...
  when C(NAME-N) S(NAME-N) goto L(NAME-N)
```

The **TCWC** is deterministic. In case more than one **when** clause is triggered, the first-mentioned one takes priority. That is, if more than one counter becomes zero because two **S(NAME-I)** occurred in the same instant then control is transferred to the **L(NAME-I)** of that clause. The other possibility is ignored.

A.4.8.4 Preemption - Handling Exceptions

The **TCWC** instruction also allows for handlers for exceptions to be declared. After an exception is raised by any one of the subprocesses, control is recovered from all the subprocesses, preempting any which were executing. Control is then transferred according to the **handle** clauses of the instruction. The form of this variant of the instruction is as follows:

```

try
  ...call-clauses...
catch
  handle E(NAME-1) goto L(NAME-1);
  handle E(NAME-2) goto L(NAME-2);
  ...
  handle E(NAME-M) goto L(NAME-M)

```

The **TCWC** is deterministic. In case more than one **handle** clause could be taken because multiple exceptions were raised in the current instant, then the first-mentioned one takes priority. That is, if more than one exception with a handle **E**(*NAME-I*) clause in the current **TCWC** is raised, then control is transferred to the **L**(*NAME-I*) of the first-mentioned clause. The other possibility is ignored.¹

A.4.9 Process Control

The other process control instructions are extremely simple.

halt

The **halt** statement serves to break up the flow of computation in separating one reactive instant from the next. The current process is halted; control is not returned to the caller.

exit

The **exit** statement returns control to the calling **TCWC** instruction. If all parallel threads of the **TCWC** have exited then execution continues from the **TCWC**. Otherwise the **TCWC** continues to be blocked for the instant.

wait **S**(*NAME-1*), **S**(*NAME-2*), ... **S**(*NAME-N*)

The **wait** instructions are but shorthands for the equivalent **TCWC**, subprocess and **halt** structure. This equivalence is shown in Figure A-15.

wait all

The **wait all** variant is a shorthand that implicitly refers to all visible signals.

1. There needs to be a **TCWC-2** to model same-priority exceptions in Esterel. This is more difficult because in that case the handlers correspond to **call** clauses which invoke subprocesses. In the case of same-priority exceptions, the handlers are invoked in parallel.

wait S(1), S(2)	<pre> L(wait): try call P(wait) I(1); watching when S(1) goto L(next); when S(2) goto L(next) L(next): ... continue ... process P(wait) is interface I(1) L(0) L(0): halt end process P(wait) </pre>
------------------------	--

Figure A-15. The Expansion of a **wait** Instruction

B The Flatten Algorithm

This appendix gives in full the **flatten** algorithm described in Section 7.2.3.2.

```
struct BuildState {
    graph    -- the estimate graph data structure
    visited  -- a table mapping a basic block to its generated estimate
    stack    -- stack of TCWC that have been called
};

flatten()
{
    BuildState state;
    state.graph = a new estimate graph
    add to state.graph a start node
    // Pretend there is single TCWC that calls all of
    // the top-level processes at once and halts if they return
    add to state.graph a fork node (with no lowered exceptions)
    add an edge from the start to the fork
    add to state.graph a collect node (with an empty raised set)

    add to state.graph a halting node (with an empty raised set)
    add an edge from collect to the halting
    add to state.graph a halted node (with an empty raised set)

    initialize tcwc_frame with no when and no handle clauses
    stack.push(tcwc_frame);

    foreach process in the top-level of the process-tree {
        BasicBlock bb = the first executable instruction in the process
        bb.generate(state, fork);
    }
    stack.pop();
}
```

B.1 BasicBlock::generate

```
void
BasicBlock::generate(BuildState& state, Estimate *pred)
// pred may be 0 in which case the caller
// promises to attach the predecessor somehow
{
    if (state.visited contains this basic block) {
        if (0 != pred) {
            estimate = state.visited[this];
            state.graph.attach(pred, estimate);
        }
        return;
    }
    if (this basic block is a TCWC) {
        estimate_TCWC(state, pred);
    } else if (this basic block is a wait or halt) {
        estimate_wait_or_halt(state, pred);
    } else {
        estimate_misc(state, pred);
    }
}
```


B.2 BasicBlock::estimate_TCWC

```
void
BasicBlock::estimate_TCWC(BuildState& state, Estimate *pred)
{
    LE = the set of exceptions mentioned in handle clauses of this TCWC
    raised_set = raised clauses of all exception handlers on state.stack.
    add to state.graph a fork node (lowering LE)
    if (0 != pred)
        add an edge from pred to the fork
        add to state.graph a collect node with raised_set
        mark this basic block as visited in state.visited
        by setting state.visited[this bb] = the fork

    NS = the set of signals mentioned in when clauses of state.stack
    initialize tcwc_frame with no when and no handle clauses
    foreach [when C(i) S(i) goto L(i)] {
        BasicBlock bb = target block L(i)
        Signal sig = signal S(i)
        bb.generate(state, /*pred*/ 0);
        add to state.graph a when node on signal sig and not NS
        add an edge from the when to bb's estimate
        add the when to tcwc_frame
        NS.add(sig);
    }

    foreach [handle E(i) goto L(i)] {
        BasicBlock bb = target block L(i)
        Exception exc = exception E(i)
        bb.generate(state, /*pred*/ 0);
        add to state.graph a handle node on exception exc
        add an edge from the handle to bb's estimate
        add the handle to tcwc_frame
    }

    BasicBlock fallout = default successor of this basic block
    fallout.generate(state, collect);

    // Handle the subprocesses now that the surrounding context in this process is fully defined:
    // all when-clauses, handle-clauses and the fallout of the TCWC.
    state.stack.push(tcwc_frame);
    foreach [call P(i) I(i)] {
        BasicBlock bb = target block of I(i) in P(i)
        bb.generate(state, fork);
    }
    state.stack.pop();
}
```

B.3 BasicBlock::estimate_wait_halt

```
void
BasicBlock::estimate_wait_halt(BuildState& state, Estimate *pred)
{
    raised_set = raised clauses of all exception handlers on state.stack.
    add to state.graph a halting node using raised_set
    if (0 != pred)
        add an edge from pred to the halting
        add to state.graph a halted node using raised_set
        mark this basic block as visited in state.visited
        by setting state.visited[this bb] = the halting

    if (this basic block is a wait) {
        BasicBlock succ = successor basic block of the wait
        succ.generate(state, /*pred*/ 0);
        NS = the set of signals mentioned in when clauses of the state.stack
        foreach signal in [wait { S(i), ... }] {
            Signal sig = S(i);
            add to state.graph a when node on signal sig and not NS
            add an edge from the when to succ
            NS.add(sig);
        }
    }
}
```

B.4 BasicBlock::estimate_misc

```
void
BasicBlock::estimate_misc(BuildState& state, Estimate *pred, Estimate*& last)
{
    create estimate nodes, saving the first estimate node created

    foreach instruction in this basic block {
        if (instruction is an emit) {
            add to state.graph an emit estimate node
            if (0 != pred)
                add an edge from pred to the emit estimate
            pred = the emit
        } else if (instruction is an undef) {
            add to state.graph an undef estimate node
            if (0 != pred)
                add an edge from pred to the undef
            pred = the undef
        } else if (instruction is an require) {
            add to state.graph a require estimate node
            if (0 != pred)
                add an edge from pred to the require
            pred = the require
            create an emit estimate node
        } else {
            // ignore it for causality estimation
        }
    }

    if (the last instruction is an exit) {
        mark this basic block as visited in state.visited
        by setting state.visited[this bb] = the first estimate created
        collect = the collect on top of state.stack
        add an edge from pred to the collect
    } else if (the last instruction is a raise) {
        Exception exc = the exception E(i) of [raise E(i) R(i)]
        raised_set = raised clauses of all exception handlers on state.stack.
        add to state.graph a raise estimate node on exc and raised_set
        add an edge from pred to the raise
        mark this basic block as visited in state.visited
        by setting state.visited[this bb] = the first estimate created
    } else {
        if (the last instruction was a present or select) {
            add to state.graph a select estimate node
            add an edge from pred to the select
        } else if (no estimate was created) {
            add to state.graph a null estimate node
            add an edge from pred to the null
        }
        mark this basic block as visited in state.visited
        by setting state.visited[this bb] = the first estimate created
        foreach succ of this basic block
            succ.generate(state, last);
    }
}
```

C Compilation of Esterel

This appendix details an example taken from Esterel all the way to compilable C++ code. When compiled in conjunction with the NDAM Runtime System defined in Appendix D, it forms a complete working module. This example¹ is a simple button interpreter as might be used in a digital watch. The interpreter's job is to recognize the "lap counter" function and to emit certain signals when it is recognized. The following three sections illustrate a complete example.

C.1 An Esterel Example

The example is assumed to be in a file called `h03.str1`.

```
module BUTTON_INTERPRETER:

input BUTTON_2, STOPWATCH_RUNNING;
output RESET;
output FROZEN_TIME;

signal
  LAP_START,
  LAP_END
in
  every BUTTON_2 do
    present STOPWATCH_RUNNING then
      emit LAP_START
    else
      present FROZEN_TIME then
        emit LAP_END
```

1. Adapted from Halbwachs [320], page 28 and 31.

```

        else
            emit RESET
        end present
    end present
end every
||
loop
    await LAP_START;
    trap T in
        sustain FROZEN_TIME
    ||
        await LAP_END;
        exit T
    end trap
end loop
end signal
end module

```

C.2 NDAM Assembly Code

Command: `est h03.str1`

The assembly code produced by the compiler is as follows:

```

network N(1) is
top: P(0) P(0): P(1), P(4)
P(1): P(2), P(3)
P(4): P(5), P(6)
P(6): P(7), P(9)
P(7): P(8)
P(9): P(10)
type t(unit) 1
type t(bool) 2
type t(int) -2147483648 2147483647
signal s(0) t(unit) -- tick
signal s(1) t(unit) -- BUTTON_2
signal s(2) t(unit) -- STOPWATCH_RUNNING
signal s(3) t(unit) -- RESET
signal s(4) t(unit) -- FROZEN_TIME
signal s(5) t(unit) -- LAP_START
signal s(6) t(unit) -- LAP_END
input: s(0), s(1), s(2)
output: s(3)
end network N(1)

process P(0) is
constant R(unit) t(unit) := 0
constant R(false) t(bool) := 0
constant R(true) t(bool) := 1
temporary R(tmp0) t(int)
L(0): undef s(4)

```

```

L(1): undef s(5)
L(2): undef s(6)
L(3): try
    call P(1) I(0);
    call P(4) I(0)
L(4): exit
end process P(0)

process P(1) is
interface I(0) L(0)
L(0): try
call P(2) I(0);
watching
when s(1) goto L(1)
L(1): try
call P(3) I(0);
watching
when s(1) goto L(2)
L(2): goto L(1)
end process P(1)

process P(2) is
interface I(0) L(0)
L(0): halt
end process P(2)

process P(3) is
interface I(0) L(0)
L(0): require s(2)
L(1): present not s(2) goto L(4)
L(2): emit s(5) R(unit)
L(3): goto L(9)
L(4): require s(4)
L(5): present not s(4) goto L(8)
L(6): emit s(6) R(unit)
L(7): goto L(9)
L(8): emit s(3) R(unit)
L(9): halt
end process P(3)

process P(4) is
interface I(0) L(0)
exception E(0) t(unit)-- T
L(0): try
    call P(5) I(0);
    watching
    when s(5) goto L(1)
L(1): try
    call P(6) I(0);
    catching
    handle e(0) goto L(2)

```

```

L(2): goto L(0)
end process P(4)

process P(5) is
interface I(0) L(0)
L(0): halt
end process P(5)

process P(6) is
interface I(0) L(0)
L(0): try
    call P(7) I(0);
    call P(9) I(0)
L(1): exit
end process P(6)

process P(7) is
interface I(0) L(0)
L(0): try
    call P(8) I(0);
    watching
    when s(0) goto L(1)
L(1): goto L(0)
end process P(7)

process P(8) is
interface I(0) L(0)
L(0): emit s(4) R(unit)
L(1): halt
end process P(8)

process P(9) is
interface I(0) L(0)
L(0): try
    call P(10) I(0);
    watching
    when s(6) goto L(1)
L(1): raise e(0) R(unit)
end process P(9)

process P(10) is
interface I(0) L(0)
L(0): halt
end process P(10)

```

C.3 Generated C++ Code

Command: `ndam -codegen h03.ndam`

The C++ code generated for the example is as follows:


```

#include <assert.h>
#include "NRS/NRSNetworkInstance.h"
static NRSSignal __S[] = {
    { /*presence*/ NRSSignal::UNKNOWN, /*potential*/ FALSE },// N(1).s(0)
    { /*presence*/ NRSSignal::UNKNOWN, /*potential*/ FALSE },// N(1).s(1)
    { /*presence*/ NRSSignal::UNKNOWN, /*potential*/ FALSE },// N(1).s(2)
    { /*presence*/ NRSSignal::UNKNOWN, /*potential*/ FALSE },// N(1).s(3)
    { /*presence*/ NRSSignal::UNKNOWN, /*potential*/ FALSE },// N(1).s(4)
    { /*presence*/ NRSSignal::UNKNOWN, /*potential*/ FALSE },// N(1).s(5)
    { /*presence*/ NRSSignal::UNKNOWN, /*potential*/ FALSE },// N(1).s(6)
};
static NRSCounter __C[] = {
};
static NRSException __E[] = {
    { /*status*/ NRSException::LOWERED, /*depth*/ -1 },// P(4).E(0)
};
static NRSSensor __X[] = {
};
static NRSvalue __V0 = 0;
static NRSvalue __V1 = 1;
static NRSvalue __V2 = /*null*/ 0;
static NRSTcwc::Call __tcwc_calls_top[] = {
    { /*process*/ 1, /*pc*/ 0 },
};
static NRSTcwc __tcwc_top = {
    /*calls*/ 1, __tcwc_calls_top,
    /*require*/ { /*signals*/ 0, 0, /*(unused)u*/ -1 },
    /*whens*/ 0, 0,
    /*handles*/ 0, 0,
    /*pc1*/ 1,
    /*pc2*/ 2,
    /*pcN*/ 3,
    /*u*/ -1
};
static NRSindex __require_signals0[] = { 2 };
static NRSRequire __require0 = { /*signals*/ 1, /*signals*/ __require_signals0, /*u*/ 0 };
static NRSindex __require_signals1[] = { 4 };
static NRSRequire __require1 = { /*signals*/ 1, /*signals*/ __require_signals1, /*u*/ 1 };
static NRSTcwc::Call __tcwc_calls2[] = {
    { /*process*/ 11, /*pc*/ 0 },
};
static NRSindex __tcwc_require_signals2[] = { 1 };
static NRSTcwc::When __tcwc_whens2[] = {
    { /*counter*/ -1, /*signal*/ 1, /*pc*/ 3 },
};
static NRSTcwc __tcwc2 = {
    /*calls*/ 1, __tcwc_calls2,
    /*require*/ { /*signals*/ 1, /*signals*/ __tcwc_require_signals2, /*u*/ 2 },
    /*whens*/ 1, __tcwc_whens2,
    /*handles*/ 0, 0,
    /*pc1*/ 1,
    /*pc2*/ 2,
    /*pcN*/ 3,
    /*u*/ 2
};
static NRSTcwc::Call __tcwc_calls3[] = {
    { /*process*/ 10, /*pc*/ 0 },
};
static NRSindex __tcwc_require_signals3[] = { 1 };
static NRSTcwc::When __tcwc_whens3[] = {
    { /*counter*/ -1, /*signal*/ 1, /*pc*/ 6 },
};
static NRSTcwc __tcwc3 = {
    /*calls*/ 1, __tcwc_calls3,
    /*require*/ { /*signals*/ 1, /*signals*/ __tcwc_require_signals3, /*u*/ 3 },
    /*whens*/ 1, __tcwc_whens3,
    /*handles*/ 0, 0,
    /*pc1*/ 4,
    /*pc2*/ 5,
    /*pcN*/ 6,
    /*u*/ 3
};
static NRSTcwc::Call __tcwc_calls4[] = {
    { /*process*/ 7, /*pc*/ 0 },
};
static NRSindex __tcwc_require_signals4[] = { 0 };
static NRSTcwc::When __tcwc_whens4[] = {
    { /*counter*/ -1, /*signal*/ 0, /*pc*/ 3 },
};
static NRSTcwc __tcwc4 = {

```

```

/*calls*/ 1, __tcwc_calls4,
/*require*/ ( /*signals*/ 1, /*signals*/ __tcwc_require_signals4, /*u*/ 4 ),
/*whens*/ 1, __tcwc_whens4,
/*handles*/ 0, 0,
/*pc1*/ 1,
/*pc2*/ 2,
/*pcN*/ 3,
/*u*/ 4
};
static NRSTcwc::Call __tcwc_calls5[] = (
( /*process*/ 5, /*pc*/ 0 ),
);
static NRSTcwc::When __tcwc_whens5[] = (
( /*counter*/ -1, /*signal*/ 6, /*pc*/ 3 ),
);
static NRSTcwc __tcwc5 = (
/*calls*/ 1, __tcwc_calls5,
/*require*/ ( /*signals*/ 1, /*signals*/ __tcwc_require_signals5, /*u*/ 5 ),
/*whens*/ 1, __tcwc_whens5,
/*handles*/ 0, 0,
/*pc1*/ 1,
/*pc2*/ 2,
/*pcN*/ 3,
/*u*/ 5
);
static NRSTcwc::Call __tcwc_calls6[] = (
( /*process*/ 6, /*pc*/ 0 ),
( /*process*/ 4, /*pc*/ 0 ),
);
static NRSTcwc __tcwc6 = (
/*calls*/ 2, __tcwc_calls6,
/*require*/ ( /*signals*/ 0, /*signals*/ 0, /*u*/ 6 ),
/*whens*/ 0, 0,
/*handles*/ 0, 0,
/*pc1*/ 1,
/*pc2*/ 2,
/*pcN*/ 3,
/*u*/ 6
);
static NRSTcwc::Call __tcwc_calls7[] = (
( /*process*/ 8, /*pc*/ 0 ),
);
static NRSTcwc::When __tcwc_whens7[] = (
( /*counter*/ -1, /*signal*/ 5, /*pc*/ 3 ),
);
static NRSTcwc __tcwc7 = (
/*calls*/ 1, __tcwc_calls7,
/*require*/ ( /*signals*/ 1, /*signals*/ __tcwc_require_signals7, /*u*/ 7 ),
/*whens*/ 1, __tcwc_whens7,
/*handles*/ 0, 0,
/*pc1*/ 1,
/*pc2*/ 2,
/*pcN*/ 3,
/*u*/ 7
);
static NRSTcwc::Call __tcwc_calls8[] = (
( /*process*/ 3, /*pc*/ 0 ),
);
static NRSTcwc::Handle __tcwc_handles8[] = (
( /*exception*/ 0, /*pc*/ 6 ),
);
static NRSTcwc __tcwc8 = (
/*calls*/ 1, __tcwc_calls8,
/*require*/ ( /*signals*/ 0, /*signals*/ 0, /*u*/ 8 ),
/*whens*/ 0, 0,
/*handles*/ 1, __tcwc_handles8,
/*pc1*/ 4,
/*pc2*/ 5,
/*pcN*/ 6,
/*u*/ 8
);
static NRSTcwc::Call __tcwc_calls9[] = (
( /*process*/ 9, /*pc*/ 0 ),
( /*process*/ 2, /*pc*/ 0 ),
);
static NRSTcwc __tcwc9 = (
/*calls*/ 2, __tcwc_calls9,
/*require*/ ( /*signals*/ 0, /*signals*/ 0, /*u*/ 9 ),
);

```

```

/*whens*/ 0, 0,
/*handles*/ 0, 0,
/*pc1*/ 2,
/*pc2*/ 3,
/*pcN*/ 4,
/*u*/ 9
};
static NRSIndex __U_local0[] = { 5, 6, 3 };
static NRSIndex __U_local1[] = { 6, 3 };
static NRSIndex __U_local2[] = { 5, 6, 3 };
static NRSIndex __U_delegated2[] = { 11 };
static NRSIndex __U_local3[] = { 5, 6, 3 };
static NRSIndex __U_delegated3[] = { 10 };
static NRSIndex __U_local4[] = { 4 };
static NRSIndex __U_delegated4[] = { 7 };
static NRSIndex __U_delegated5[] = { 5 };
static NRSIndex __U_local6[] = { 4 };
static NRSIndex __U_delegated6[] = { 6, 4 };
static NRSIndex __U_local7[] = { 4 };
static NRSIndex __U_delegated7[] = { 8 };
static NRSIndex __U_local8[] = { 4 };
static NRSIndex __U_delegated8[] = { 3 };
static NRSIndex __U_local9[] = { 5, 6, 3, 4 };
static NRSIndex __U_delegated9[] = { 9, 2 };
static NRSPotential __U[] = {
    { /*nlocal*/ 3, /*local*/ __U_local0, /*ndelegated*/ 0, /*delegated*/ 0 }, // P(3).L(0)
    { /*nlocal*/ 2, /*local*/ __U_local1, /*ndelegated*/ 0, /*delegated*/ 0 }, // P(3).L(4)
    { /*nlocal*/ 3, /*local*/ __U_local2, /*ndelegated*/ 1, /*delegated*/ __U_delegated2 }, //
P(1).L(0)
    { /*nlocal*/ 3, /*local*/ __U_local3, /*ndelegated*/ 1, /*delegated*/ __U_delegated3 }, //
P(1).L(1)
    { /*nlocal*/ 1, /*local*/ __U_local4, /*ndelegated*/ 1, /*delegated*/ __U_delegated4 }, //
P(7).L(0)
    { /*nlocal*/ 0, /*local*/ 0, /*ndelegated*/ 1, /*delegated*/ __U_delegated5 }, // P(9).L(0)
    { /*nlocal*/ 1, /*local*/ __U_local6, /*ndelegated*/ 2, /*delegated*/ __U_delegated6 }, //
P(6).L(0)
    { /*nlocal*/ 1, /*local*/ __U_local7, /*ndelegated*/ 1, /*delegated*/ __U_delegated7 }, //
P(4).L(0)
    { /*nlocal*/ 1, /*local*/ __U_local8, /*ndelegated*/ 1, /*delegated*/ __U_delegated8 }, //
P(4).L(1)
    { /*nlocal*/ 4, /*local*/ __U_local9, /*ndelegated*/ 2, /*delegated*/ __U_delegated9 }, //
P(0).L(3)
};
static NRSInput __I[] = {
    { /*signal*/ 0, /*io*/ NRSNetwork::read, { /*size*/ 0, /*value*/ 0 }, // N(1).s(0)
    { /*signal*/ 1, /*io*/ NRSNetwork::read, { /*size*/ 0, /*value*/ 0 }, // N(1).s(1)
    { /*signal*/ 2, /*io*/ NRSNetwork::read, { /*size*/ 0, /*value*/ 0 }, // N(1).s(2)
};
static NRSOutput __O[] = {
    { /*signal*/ 3, /*io*/ NRSNetwork::write, { /*size*/ 0, /*value*/ 0 }, // N(1).s(3)
};
static NRSProcess::State
__CODE_top(NRSNetworkInstance& network, NRSProcess& self)
{
    NRSProcess::State s;
    switch (self.pc) {
    case 0:// top(TAIL)
    case 1:// top(SYNC)
    case 2:// top(HEAD)
        s = network.TCWC(self, __tcwc_top, (NRSTcwc::Phase) self.pc);
        if (NRSProcess::CONTINUING != s)
            return s;
        self.pc = 3;
        // fallthru
    case 3:// top(done)
        self.u = -1;
        return NRSProcess::STABLE;
    default:// this is an erroneous pc value (just return stable)
        self.u = -1;
        return NRSProcess::STABLE;
    }
}
static NRSProcess::State
__CODE1(NRSNetworkInstance& network, NRSProcess& self)
{
    do {
        switch (self.pc) {
        case 0:// L(0)
            network.S[4].presence = NRSSignal::UNKNOWN;
            network.S[5].presence = NRSSignal::UNKNOWN;

```

```

network.S[6].presence = NRSSignal::UNKNOWN;
self.pc = 1;
continue;
case 1:// L(3){TAIL}
case 2:// L(3){SYNC}
case 3:// L(3){HEAD}
{
    NRSProcess::State s = network.TCWC(self, __tcwc9, (NRSTcwc::Phase) (self.pc-1));
    if (NRSProcess::CONTINUING != s)
        return s;
}
continue;
case 4:// L(4)
self.u = -1;
return NRSProcess::TERMINATED;
default:// this is an erroneous pc value (just return stable)
self.u = -1;
return NRSProcess::STABLE;
}
} while (1);
}
static NRSProcess::State
__CODE2(NRSNetworkInstance& network, NRSProcess& self)
{
    do {
        switch (self.pc) {
            case 0:// L(0){TAIL}
            case 1:// L(0){SYNC}
            case 2:// L(0){HEAD}
            {
                NRSProcess::State s = network.TCWC(self, __tcwc7, (NRSTcwc::Phase) (self.pc-0));
                if (NRSProcess::CONTINUING != s)
                    return s;
            }
            continue;
            case 3:// L(1){TAIL}
            case 4:// L(1){SYNC}
            case 5:// L(1){HEAD}
            {
                NRSProcess::State s = network.TCWC(self, __tcwc8, (NRSTcwc::Phase) (self.pc-3));
                if (NRSProcess::CONTINUING != s)
                    return s;
            }
            continue;
            case 6:// L(2)
            self.pc = 0;
            continue;
            default:// this is an erroneous pc value (just return stable)
            self.u = -1;
            return NRSProcess::STABLE;
        }
    } while (1);
}
static NRSProcess::State
__CODE3(NRSNetworkInstance& network, NRSProcess& self)
{
    do {
        switch (self.pc) {
            case 0:// L(0){TAIL}
            case 1:// L(0){SYNC}
            case 2:// L(0){HEAD}
            {
                NRSProcess::State s = network.TCWC(self, __tcwc6, (NRSTcwc::Phase) (self.pc-0));
                if (NRSProcess::CONTINUING != s)
                    return s;
            }
            continue;
            case 3:// L(1)
            self.u = -1;
            return NRSProcess::TERMINATED;
            default:// this is an erroneous pc value (just return stable)
            self.u = -1;
            return NRSProcess::STABLE;
        }
    } while (1);
}
static NRSProcess::State
__CODE4(NRSNetworkInstance& network, NRSProcess& self)
{
    do {

```

```

switch (self.pc) {
case 0:// L(0){TAIL}
case 1:// L(0){SYNC}
case 2:// L(0){HEAD}
{
    NRSProcess::State s = network.TCWC(self, __tcwc5, (NRSTcwc::Phase) (self.pc-0));
    if (NRSProcess::CONTINUING != s)
        return s;
}
continue;
case 3:// L(1)
network.E[0].status = NRSException::RAISED;
if (network.edepth < network.E[0].depth)
    network.edepth = network.E[0].depth;
self.u = -1;
return NRSProcess::RAISED;
default:// this is an erroneous pc value (just return stable)
self.u = -1;
return NRSProcess::STABLE;
}
} while (1);
}
static NRSProcess::State
__CODE5(NRSNetworkInstance& network, NRSProcess& self)
{
do {
switch (self.pc) {
case 0:// L(0)
self.u = -1;
return NRSProcess::STABLE;
default:// this is an erroneous pc value (just return stable)
self.u = -1;
return NRSProcess::STABLE;
}
} while (1);
}
static NRSProcess::State
__CODE6(NRSNetworkInstance& network, NRSProcess& self)
{
do {
switch (self.pc) {
case 0:// L(0){TAIL}
case 1:// L(0){SYNC}
case 2:// L(0){HEAD}
{
    NRSProcess::State s = network.TCWC(self, __tcwc4, (NRSTcwc::Phase) (self.pc-0));
    if (NRSProcess::CONTINUING != s)
        return s;
}
continue;
case 3:// L(1)
self.pc = 0;
continue;
default:// this is an erroneous pc value (just return stable)
self.u = -1;
return NRSProcess::STABLE;
}
} while (1);
}
static NRSProcess::State
__CODE7(NRSNetworkInstance& network, NRSProcess& self)
{
do {
switch (self.pc) {
case 0:// L(0)
network.S[4].presence = NRSSignal::PRESENT;
self.pc = 1;
continue;
case 1:// L(1)
self.u = -1;
return NRSProcess::STABLE;
default:// this is an erroneous pc value (just return stable)
self.u = -1;
return NRSProcess::STABLE;
}
} while (1);
}
static NRSProcess::State
__CODE8(NRSNetworkInstance& network, NRSProcess& self)
{

```

```

do {
  switch (self.pc) {
    case 0:// L(0)
      self.u = -1;
      return NRSProcess::STABLE;
    default:// this is an erroneous pc value (just return stable)
      self.u = -1;
      return NRSProcess::STABLE;
  }
} while (1);
}
static NRSProcess::State
_CODE9(NRSNetworkInstance& network, NRSProcess& self)
{
  do {
    switch (self.pc) {
      case 0:// L(0)(TAIL)
      case 1:// L(0)(SYNC)
      case 2:// L(0)(HEAD)
        {
          NRSProcess::State s = network.TCWC(self, __tcwc2, (NRSTcwc::Phase) (self.pc-0));
          if (NRSProcess::CONTINUING != s)
            return s;
        }
        continue;
      case 3:// L(1)(TAIL)
      case 4:// L(1)(SYNC)
      case 5:// L(1)(HEAD)
        {
          NRSProcess::State s = network.TCWC(self, __tcwc3, (NRSTcwc::Phase) (self.pc-3));
          if (NRSProcess::CONTINUING != s)
            return s;
        }
        continue;
      case 6:// L(2)
        self.pc = 3;
        continue;
      default:// this is an erroneous pc value (just return stable)
        self.u = -1;
        return NRSProcess::STABLE;
    }
  } while (1);
}
static NRSProcess::State
_CODE10(NRSNetworkInstance& network, NRSProcess& self)
{
  do {
    switch (self.pc) {
      case 0:// L(0)
        {
          NRSProcess::State s = network.REQUIRE(self, __require0);
          if (NRSProcess::CONTINUING != s)
            return s;
        }
        self.pc = 1;
        continue;
      case 1:// L(1)
        if (NRSSignal::ABSENT == network.S[2].presence) {
          self.pc = 2;
          continue;
        }
        self.pc = 3;
        continue;
      case 2:// L(4)
        {
          NRSProcess::State s = network.REQUIRE(self, __require1);
          if (NRSProcess::CONTINUING != s)
            return s;
        }
        self.pc = 5;
        continue;
      case 3:// L(2)
        network.S[5].presence = NRSSignal::PRESENT;
        self.pc = 4;
        continue;
      case 4:// L(9)
        self.u = -1;
        return NRSProcess::STABLE;
      case 5:// L(5)
        if (NRSSignal::ABSENT == network.S[4].presence) {

```

```

        self.pc = 6;
        continue;
    }
    self.pc = 7;
    continue;
    case 6:// L(8)
    network.S[3].presence = NRSSignal::PRESENT;
    self.pc = 4;
    continue;
    case 7:// L(6)
    network.S[6].presence = NRSSignal::PRESENT;
    self.pc = 4;
    continue;
    default:// this is an errorneous pc value (just return stable)
    self.u = -1;
    return NRSProcess::STABLE;
    }
} while (1);
}
static NRSProcess::State
__CODE11(NRSNetworkInstance network, NRSProcess self)
{
    do {
        switch (self.pc) {
            case 0:// L(0)
            self.u = -1;
            return NRSProcess::STABLE;
            default:// this is an errorneous pc value (just return stable)
            self.u = -1;
            return NRSProcess::STABLE;
        }
    } while (1);
}
static NRSProcess __P[] = {
    ( /*pc*/ 0, /*u*/ -1, /*depth*/ 1, /*stable*/ FALSE, /*CODE*/ __CODE_top },// synthetic toplevel
    ( /*pc*/ 0, /*u*/ -1, /*depth*/ 0, /*stable*/ FALSE, /*CODE*/ __CODE1 },// P(0)
    ( /*pc*/ 0, /*u*/ -1, /*depth*/ -1, /*stable*/ FALSE, /*CODE*/ __CODE2 },// P(4)
    ( /*pc*/ 0, /*u*/ -1, /*depth*/ -2, /*stable*/ FALSE, /*CODE*/ __CODE3 },// P(6)
    ( /*pc*/ 0, /*u*/ -1, /*depth*/ -3, /*stable*/ FALSE, /*CODE*/ __CODE4 },// P(9)
    ( /*pc*/ 0, /*u*/ -1, /*depth*/ -4, /*stable*/ FALSE, /*CODE*/ __CODE5 },// P(10)
    ( /*pc*/ 0, /*u*/ -1, /*depth*/ -3, /*stable*/ FALSE, /*CODE*/ __CODE6 },// P(7)
    ( /*pc*/ 0, /*u*/ -1, /*depth*/ -4, /*stable*/ FALSE, /*CODE*/ __CODE7 },// P(8)
    ( /*pc*/ 0, /*u*/ -1, /*depth*/ -2, /*stable*/ FALSE, /*CODE*/ __CODE8 },// P(5)
    ( /*pc*/ 0, /*u*/ -1, /*depth*/ -1, /*stable*/ FALSE, /*CODE*/ __CODE9 },// P(1)
    ( /*pc*/ 0, /*u*/ -1, /*depth*/ -2, /*stable*/ FALSE, /*CODE*/ __CODE10 },// P(3)
    ( /*pc*/ 0, /*u*/ -1, /*depth*/ -2, /*stable*/ FALSE, /*CODE*/ __CODE11 },// P(2)
};
NRSNetworkInstance N(0, __C, 1, __E, 3, __I, 1, __O, 10, __U, 12, __P, 0, __X, 7, __S);

```

D The NDAM Runtime System

The NDAM Runtime System (NRS) consists of two separable parts. The first is the basic minimal runtime system. The second is a debugger interface that incorporates assembly-level debugger information. This section covers the minimal runtime system. A larger runtime system is available that supports a source-level debugger interface.

The minimal runtime system comprises roughly 346 lines of C++ code that support the functions of **EVAL** and **MARK** and the routines to support the operations of the instructions **TCWC**, **REQUIRE** and **WAIT**. While the code to support these operations could be inlined, a substantial code size savings was realized by placing these functions in a library.

Section D.1 defines the actual data structures which are used to implement the minimal runtime system. Section D.2 defines the support code of the NRS and Section D.3 defines the debugger support data structures and how they are used in a simple debugger.

D.1 Data Structures

The basic data structures can be considered in two classes:

Structural: **NRSNetwork, NRSNetworkInstance, NRSProcess.**

Behavioral: **NRSRequire, NRWait, NRSTcwc,
NRSPotential, NRSSignal, NRSException,
NRSCounter, NRSValue,
NRSInput, NRSOutput, NRSSensor**

These are explained in the following sections.

D.1.1 NRSNetwork

The network is actually an abstraction of a network instance. This is a sort of wrapper which hides the actual details of a particular network instance. This allows code which calls an NDAM-implemented module to avoid an explicit dependence on the particulars of a network instance. The avoidance of such dependencies is often extremely important in C++. This data structure is an ancestor of **NRSNetworkInstance**: it is never constructed by a user but is declared as the abstraction of the **NRSNetworkInstance** of the next section.

```
struct NRSNetwork {
    virtual void EVAL() = 0;
    virtual void RESET() = 0;
    // The default read and write for signals and sensors
    // p is the presence; the value is ignored if p is FALSE
    static void read(NRSNetwork& N, bool& p, NRSlength n, NRSvalue *v);
    static void write(NRSNetwork& N, bool p, NRSlength n, NRSvalue *v);
    void *userData;
    NRSNetwork();
    virtual ~NRSNetwork();
};
```

NRSNetwork::EVAL

Evaluate the underlying network instance.

NRSNetwork::RESET

Reset the underlying network instance

NRSNetwork::userData

A pointer to some piece of associated user data. Typically this is the **NDBDebugger**.

D.1.2 NRSNetworkInstance

This is the actual network instance which is created by the compiler. The static declarations are filled in with the pointers to the statically-generated data.

```
struct NRSNetworkInstance {
    void EVAL();
    void MARK(NRSIndex process);
    NRSlength NPROCESSES;
    NRSProcess *P;
    NRSlength NPOTENTIALS;
    NRSPotential *U;
    NRSlength NSIGNALS;
    NRSSignal *S;
    NRSlength NEXCEPTIONS;
    NRSException *E;
    NRSlength NCOUNTERS;
    NRSCounter *C;
    NRSlength NINPUTS;
    NRSInput *I;
    NRSlength NOUTPUTS;
    NRSOutput *O;
    NRSlength NSENSORS;
    NRSSensor *X;
    NRSdepth edepth; // the MAX of the exception(s) just raised
    NRSProcess::State REQUIRE(NRSProcess& process,
                             const NRSRequire& require);
    NRSProcess::State WAIT(NRSProcess& process, const NRSWait& wait);
    NRSProcess::State TCWC(NRSProcess& process,
                          const NRSTcwc& tcwc, NRSTcwc::Phase phase);
};
```

NRSNetworkInstance::EVAL

Evaluate the network instance.

NRSNetworkInstance::MARK

The mark algorithm for the network.

NRSNetworkInstance::NPROCESSES

NRSNetworkInstance::P

The processes of the network.

NRSNetworkInstance::NPOTENTIALS

NRSNetworkInstance::U

The potential sets of the processes in the network.

NRSNetworkInstance::NSIGNALS
NRSNetworkInstance::S

The signals of the network.

NRSNetworkInstance::NEXCEPTIONS
NRSNetworkInstance::E

The exceptions of the network.

NRSNetworkInstance::NCOUNTERS
NRSNetworkInstance::C

The counters of the network.

NRSNetworkInstance::NINPUTS
NRSNetworkInstance::I

The inputs of the network. These are indices into **S**.

NRSNetworkInstance::NOUTPUTS
NRSNetworkInstance::O

The outputs of the network. These are indices into **S**.

NRSNetworkInstance::NSENSORS
NRSNetworkInstance::X

The sensors of the network.

NRSNetworkInstance::edepth

The maximum depth of exceptions raised by a **TCWC**; used in NRS.

D.1.3 NRSProcess

The process structure manages all of the dynamic data relevant for a process. It also includes a pointer to the generated-code which executes the process.

```
struct NRSProcess {
    enum State { SYNCING, CONTINUING, TERMINATED, STABLE, RAISED };
    static const bool TCLOSE[5];           // used in EVAL; indexed by State
    static const char * const WORD[5];    // used in NDAM codegen
    NRSindex pc;                           // the program counter;
                                           // switch key in code
    NRSindex u;                             // the potential function;
                                           // index NRSindex U[] (may be -1)
    NRSdepth depth;
    bool stable;                            // not runnable
    State (*CODE)(NRSNetworkInstance& network, NRSProcess& self);
};
```

NRSProcess::pc

The program counter of the process.

NRSProcess::u

The potential set of the process.

NRSProcess::depth

The depth of the process for the purpose of exception resolution.

NRSProcess::stable

A boolean value indicating whether a process has gone stable in the instant.

NRSProcess::CODE

The code body corresponding to the process.

D.1.4 NDAM Instruction Template Structures

Certain NDAM instructions are implemented by NRS library routines. Those routines are controlled by tables. These instructions are therefore completely characterized by these statically-generated tables.

D.1.4.1 Require

```
struct NRSRequire {
    NRSlength nsignals;
    NRSindex *signals;
    NRSindex u;
};
```

NRSRequire::nsignals
NRSRequire::signals

The statically-allocated vector of signal indices that the **require** is to require.

NRSRequire::u

An index into the potential set vector.

D.1.4.2 Wait

```
struct NRSWait {
    enum Phase { TAIL, HEAD };
    NRSRequire require;
};
```

NLSWait::Phase

A **wait** has two phases: there is the **TAIL** the part which ends an instant; and there is the **HEAD** part which may commence an instant. These values are used within NRS.

NLSwait::require

A 'wait' implicitly requires the knowledge of the signals that it mentions. This field is that implicit 'require' instruction.

D.1.4.3 TCWC

```
struct NRSTcwc {
    enum Phase { TAIL, SYNC, HEAD };
    struct Call {
        NRSindex process;    // index into P[]
        NRSindex interface; // the initial pc value
    };
    NRSlength ncalls;
    Call *calls;
    struct When {
        NRSindex counter;    // index into C[] or -1 if no counter
        NRSindex signal;    // index into S[]
        NRSindex pc;        // pc to assign on expiration
    };
    NRSRequire require;
    NRSlength nwhens;
    When *whens;
    struct Handle {
        NRSindex exception; // index into E[]
        NRSindex pc;        // pc to assign on raise
    };
    NRSlength nhandles;
    Handle *handles;
    NRSindex pc1;    // the tcwc the nth time through (phase 1)
    NRSindex pc2;    // the tcwc the nth time through (phase 2)
    NRSindex pcN;    // the next pc
    NRSindex u;
};
```

NRSTcwc::Phase

The **TCWC** has three phases corresponding to: a **TAIL** when it is invoked for the first time in an instant; a **SYNC** when it is invoked multiply in an instant while its children compute further; and **HEAD** when it possibly begins an instant after its children did not complete in some previous instant.

NRSTcwc::ncalls

NRSTcwc::calls

The subprocesses called by the **TCWC**.

NRSTcwc::require

A **TCWC** has an implicit requirement that the status of the signals for which it holds guards be known before any further progress be made. This field is that implicit 'require.'

NRSTcwc::nwhens
NRSTcwc::whens

The 'when' clauses of the signal guards of the *TCWC*.

NRSTcwc::nhandles
NRSTcwc::handles

The 'handle' clauses of the exception handlers of the *TCWC*.

NRSTcwc::pc1

The program counter on phase 1.

NRSTcwc::pc2

The program counter on phase 2.

NRSTcwc::pcN

The program counter of the successor instruction.

NRSTcwc::u

The local signal potential of this instruction.

D.1.4.4 Potential

```
struct NRSPotential {  
    NRSlength nlocal;  
    NRSindex *local;  
    NRSlength ndelegated;  
    NRSindex *delegated;  
};
```

NRSPotential::nlocal
NRSPotential::local

The signals which have the potential to be emitted from within the current process.

NRSPotential::ndelegated
NRSPotential::delegated

The child processes which must be explored as well. This is a vacuous list except for the *TCWC*.

D.1.4.5 Signal

```
struct NRSSignal {  
    enum Presence { UNKNOWN, ABSENT, PRESENT, EMITTED };  
    Presence presence;  
    bool potential;  
};
```

NRSSignal::Presence

The sorts of status that a signal can have.

NRSSignal::presence

The actual status of the signal.

NRSSignal::potential

Whether the signal has the potential to be emitted in some future microstep.

D.1.4.6 Exception

```
struct NRSException {
    enum Status { LOWERED, RAISED };
    Status status;
    NRSdepth depth;
};
```

NRSException::Status

The sorts of status that an exception can have.

NRSException::status

The actual status of the exception.

NRSException::depth

The depth in the process tree of the instantiated handler of the exception.

D.1.4.7 Counter

```
struct NRSCounter {
    unsigned value;
    unsigned initial;
    bool expired() const;
    void init();
    void dec();
};
```

NRSCounter::value

The current value of the counter.

NRSCounter::initial

The initializable value of the counter.

NRSCounter::expired

Is the counter expired?

NRSCounter::init

Reset the counter to its initial value.

NRSCounter::dec

Decrement the counter.

D.1.4.8 Value

The value is a descriptor which is used to inform the input and output system the size and location of the actual signal values. This allows a generic input and output routine to be created which reads in the inputs and writes out the outputs without reference to any particular network instance.

```
struct NRSValue {
    NRSlength size;      // may be 0
    NRSvalue *value;    // is 0 if size is 0
};
```

NRSValue::size

The size of the value.

NRSValue::value

The locations of the values.

D.1.4.9 Input

```
struct NRSInput {
    NRSindex index; // index into S[];
    void (*read)(NRSNetwork&, bool& p, NRSlength n, NRSvalue *v);
    NRSvalue desc;
};
```

NRSInput::index

The signal corresponding to this input. It is an index into the **S** vector.

NRSInput::read

The read routine for that particular sort of input.

NRSInput::desc

The value descriptor.

D.1.4.10 Output

```
struct NRSOutput {
    NRSindex index; // index into S[];
    void (*write)(NRSNetwork&, bool p, NRSlength n, NRSvalue *v);
    NRSvalue desc;
};
```

NRSOutput::index

The signal corresponding to this output. It is an index into the **S** vector.

NRSOutput::read

The write routine for that particular sort of output.

NRSOutput::desc

The value descriptor.

D.1.4.11 Sensor

```
struct NRSSensor {  
    void (*read)(NRSNetworks, NRSlength n, NRSvalue *v);  
    void (*write)(NRSNetworks, NRSlength n, NRSvalue *v);  
};
```

NRSSensor::read

The read operation for the sensor.

NRSSensor::write

The write operation for the sensor.

D.2 Support Code

The minimal support code for NRS is 346 lines of C++. These support routines use the data structures defined in Section C.1. The size for the various routines in lines of code and bytes of object is given in Table 1.

Source file	Lines of C++	Object File Size		
		text	data	bss
EVAL.cc	41	356	0	0
MARK.cc	27	216	0	0
REQUIRE.cc	24	88	0	0
TCWC.cc	226	748	0	0
WAIT.cc	28	416	0	0
Total	346	1524	0	0

Table 1. Size in Bytes of the NRS Runtime^a

a. Compiler is g++ Cygnus reno-1.3, on i486 running Linux 1.0.x

D.2.1 -EVAL

```
void
NRSNetworkInstance::EVAL()
{
    unsigned i;
    for (i=0; i<NPROCESSES; i++)
        P[i].stable = FALSE;
    // Input Phase
    for (i=0; i<NSIGNALS; i++)
        S[i].presence = NRSSignal::UNKNOWN;
    (this->*read_inputs)();
    NRSProcess::State s;
    do {
        // Execute Phase
        s = (*P[0].CODE)(*this, P[0]);
        if (NRSProcess::RAISED == s) {
            // This condition is a codegen error but if it is not taken care
            // of then an infinite loop occurs: pretend it was an 'exit'
            s = NRSProcess::TERMINATED;
        }
        // Mark Phase
        for (i=0; i<NSIGNALS; i++)
            S[i].potential = FALSE;
        MARK(0);
        for (i=0; i<NSIGNALS; i++) {
            if ( ! S[i].potential ) {
                if (NRSSignal::UNKNOWN == S[i].presence)
                    S[i].presence = NRSSignal::ABSENT;
                if (NRSSignal::EMITTED == S[i].presence) {
                    S[i].presence = NRSSignal::PRESENT;
                }
            }
        }
    } while (NRSProcess::TCLOSE[s]);
    // Output Phase
    (this->*write_outputs)();
}
```

D.2.2 MARK

The MARK algorithm determines the dynamic signal potentials.

```
void
NRSNetworkInstance::MARK(NRSindex process)
{
    if (P[process].stable)
        return;
    NRSindex u = P[process].u;
    if (u < 0) // the nil potential function
        return;
    unsigned i;
    for (i=0; i<U[u].nlocal; i++)
        S[ U[u].local[i] ].potential = TRUE;
    for (i=0; i<U[u].ndelegated; i++)
        MARK( U[u].delegated[i] );
}
```

D.2.3 TCWC

```
NRSProcess::State
NRSNetworkInstance::TCWC(NRSProcess& process,
                          const NRSTcwc& tcwc, NRSTcwc::Phase phase)
{
    if (process.stable)
        return NRSProcess::STABLE;
    unsigned i;
    switch (phase) {
        case NRSTcwc::TAIL:
            for (i=0; i<tcwc.ncalls; i++) {
                NRSTcwc::Call& call = tcwc.calls[i];
                NRSProcess& proc = P[call.process];
                proc.pc = call.interface;
                proc.u = -1; // gratuitous
            }
            for (i=0; i<tcwc.nwhens; i++) {
                NRSTcwc::When& when = tcwc.whens[i];
                if (when.counter >= 0)
                    C[when.counter].init();
            }
            edepth = NRS_DEPTH_FLOOR;
            for (i=0; i<tcwc.nhandles; i++) {
                NRSTcwc::Handle& handle = tcwc.handles[i];
                E[handle.exception].status = NRSException::LOWERED;
            }
            break;
        case NRSTcwc::SYNC:
            // This is the intermediate SYNCING phase which is not the first
            // invocation of the instruction and its not the guarding case
            // when the instruction is invoked in a subsequent instant.
            break;
        case NRSTcwc::HEAD:
            if (NRSProcess::SYNCING == REQUIRE(process, tcwc.require))
                return NRSProcess::SYNCING;
            for (i=0; i<tcwc.nwhens; i++) {
                NRSTcwc::When& when = tcwc.whens[i];
                if (NRSSignal::PRESENT == S[when.signal].presence) {
                    if (when.counter >= 0)
                        C[when.counter].dec();
                }
            }
            for (i=0; i<tcwc.nwhens; i++) {
                NRSTcwc::When& when = tcwc.whens[i];
                if (NRSSignal::PRESENT == S[when.signal].presence) {
                    if (when.counter < 0 || C[when.counter].expired()) {
                        process.pc = when.pc;
                        process.u = -1; // gratuitous
                        return NRSProcess::CONTINUING;
                    }
                }
            }
            break;
    }
    bool raised = FALSE;
    bool stable = FALSE;
    bool syncing = FALSE;
    NRSProcess::State s;
    for (i=0; i<tcwc.ncalls; i++) {
        NRSTcwc::Call& call = tcwc.calls[i];
        s = (*P[call.process].CODE)(*this, P[call.process]);
        switch (s) {
```

```

    case NRSProcess::RAISED:
        raised |= TRUE;
        break;
    case NRSProcess::STABLE:
        stable |= TRUE;
        break;
    case NRSProcess::SYNCING:
        syncing |= TRUE;
        break;
    case NRSProcess::CONTINUING:
    case NRSProcess::TERMINATED:
    default:// this should never happen
        break;
}
}
if (syncing) {
    process.pc = tcwc.pc1;
    process.u = tcwc.u;
    return NRSProcess::SYNCING;
}
if (raised) {
    if (edepth > process.depth)
        return NRSProcess::RAISED;
    for (i=0; i<tcwc.nhandles; i++) {
        NRSTcwc::Handle& handle = tcwc.handles[i];
        if (NRSException::RAISED == E[handle.exception].status) {
            process.pc = handle.pc;
            process.u = -1; // gratuitous
            return NRSProcess::CONTINUING;
        }
    }
}
if (stable) {
    process.stable = TRUE;
    process.pc = tcwc.pc2;
    process.u = tcwc.u;
    return NRSProcess::STABLE;
}
process.pc = tcwc.pcN;
return NRSProcess::CONTINUING;
}

```

D.2.4 REQUIRE

```
NRSProcess::State
NRSNetworkInstance::REQUIRE(NRSProcess& process,
                              const NRSRequire& require)
{
    for (unsigned i=0; i<require.nsignals; i++) {
        // Can be either PRESENT or ABSENT but not unknown
        if (NRSSignal::UNKNOWN == S[require.signals[i]].presence) {
            // process.pc stays the same
            process.u = require.u;
            return NRSProcess::SYNCING;
        }
    }
    // process.pc is set by the caller
    return NRSProcess::CONTINUING;
}
```

D.2.5 WAIT

```
NRSProcess::State
NRSNetworkInstance::WAIT(NRSProcess& process, const NRSWait& wait)
{
    NRSProcess::State s = REQUIRE(process, wait.require);
    if (NRSProcess::CONTINUING != s)
        return s;
    // No signal will have an UNKNOWN presence
    for (unsigned i=0; i<wait.require.nsignals; i++) {
        if (NRSSignal::PRESENT == S[wait.require.signals[i]].presence) {
            // process.pc set in the caller by virtue of continuing on
            return NRSProcess::CONTINUING;
        }
    }
    // process.pc remains unchanged
    return NRSProcess::STABLE;
}
```

D.3 Debugger Support

The debugger for NRS consists of a Tcl/Tk [571] user interface which allows for the relevant aspects of the runtime system to be displayed: program counters, signals, registers, exceptions, sensors and the potential sets. To facilitate this, there are a few extra runtime data structures which are generated at compile time. These data structures map lines and names to locations in the runtime system. The user interface uses these data structures to present a user-friendly picture of the runtime system. The following sections

define these data structures.

D.3.1 NDBDebugger

The debugger consists of a pointer to a Tcl interpreter , a symbol table “database,” the actual NDAM network and interfaces to the trace file reader and writer:

```
struct NDBDebugger {
    Tcl_Interp *interp;
    NDBStab *stab;
    NRSNetworkInstance *network;
    NDBTraceFileReader *reader;
    NDBTraceFileWriter *writer;
    NDBDebugger(Tcl_Interp *interpreter,
                NRSNetworkInstance& network_instance,
                NDBStab& instance_stab);
    ~NDBDebugger();

    void INITIAL();
    void UPDATE();
    void RESET();
    void EVAL();

    void read_inputs();
    void write_outputs();
};
```

NDBDebugger::INITIAL

Initialize the debugger

NDBDebugger::UPDATE

Update the user interface with values from the network

NDBDebugger::RESET

Reset the debugger

NDBDebugger::EVAL

Evaluate the network with the previously-defined inputs

NDBDebugger::read_inputs

Read the input values that have been defined in the user interface.
Set them into the network in preparation for an **EVAL**.

NDBDebugger::write_outputs

Write the output values back to the user interface.
The output values were produced by the network during **EVAL**.

D.3.2 The Stab Record

The Symbol Table “database” manages all of the instruction source code line number and register, signal and exception name mappings. The majority of the structure consists of tables which are created at compile-time with the debug option.

```
struct NDBStab {
    struct Process {
        struct Label {
            char *name;
            int line;    // zero offset; -1 means undefined
            int start;  // -1 means undefined
            int end;    // -1 means undefined
        };
        char *name;
        unsigned nlabels;
        Label *L;
    };
    struct Signal {
        char *name;
        NRSValue desc;
    };
    struct Exception {
        char *name;
        NRSValue desc;
    };
    struct Register {
        char *name;
        NRSValue desc;
    };
    struct Sensor {
        char *name;
    };
    struct Counter {
        char *name;
    };
    struct Type {
        char *name;
        long low;
        long high;
    };
    unsigned NPROCESSES;
    Process *P;
    unsigned NSIGNALS;
    Signal *S;
    unsigned NEXCEPTIONS;
    Exception *E;
    unsigned NREGISTERS;
    Register *R;
    unsigned NSENSORS;
```

```

Sensor *X;
unsigned NCOUNTERS;
Counter *C;
char *sourcefile;
NDBStab(unsigned _NCOUNTERS, Counter *_C,
         unsigned _NEXCEPTIONS, Exception *_E,
         unsigned _NPROCESSES, Process *_P,
         unsigned _NREGISTERS, Register *_R,
         unsigned _NSENSORS, Sensor *_X,
         unsigned _NSIGNALS, Signal *_S,
         char *filename);
-NDBStab();
void UPDATE(Tcl_Interp *interp, NRSNetworkInstance *N);
};

```

NDBStab::UPDATE

The user interface is updated with the values from the network instance.

