

INFORMATION TO USERS

This reproduction was made from a copy of a document sent to us for microfilming. While the most advanced technology has been used to photograph and reproduce this document, the quality of the reproduction is heavily dependent upon the quality of the material submitted.

The following explanation of techniques is provided to help clarify markings or notations which may appear on this reproduction.

1. The sign or "target" for pages apparently lacking from the document photographed is "Missing Page(s)". If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting through an image and duplicating adjacent pages to assure complete continuity.
2. When an image on the film is obliterated with a round black mark, it is an indication of either blurred copy because of movement during exposure, duplicate copy, or copyrighted materials that should not have been filmed. For blurred pages, a good image of the page can be found in the adjacent frame. If copyrighted materials were deleted, a target note will appear listing the pages in the adjacent frame.
3. When a map, drawing or chart, etc., is part of the material being photographed, a definite method of "sectioning" the material has been followed. It is customary to begin filming at the upper left hand corner of a large sheet and to continue from left to right in equal sections with small overlaps. If necessary, sectioning is continued again—beginning below the first row and continuing on until complete.
4. For illustrations that cannot be satisfactorily reproduced by xerographic means, photographic prints can be purchased at additional cost and inserted into your xerographic copy. These prints are available upon request from the Dissertations Customer Services Department.
5. Some pages in any document may have indistinct print. In all cases the best available copy has been filmed.

**University
Microfilms
International**

300 N. Zeeb Road
Ann Arbor, MI 48106

8513124

Hooper, Richard Preston

AN APPLICATION OF KNOWLEDGE-BASED SYSTEMS TO ELECTRONIC
COMPUTER-AIDED ENGINEERING, DESIGN, AND MANUFACTURING DATA
BASE TRANSPORT

University of California, Los Angeles

PH.D. 1985

**University
Microfilms
International** 300 N. Zeeb Road, Ann Arbor, MI 48106

Copyright 1985

by

Hooper, Richard Preston

All Rights Reserved

UNIVERSITY OF CALIFORNIA

Los Angeles

An Application of Knowledge-Based Systems to
Electronic Computer-Aided Engineering, Design,
and Manufacturing Data Base Transport


A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy
in Computer Science

by

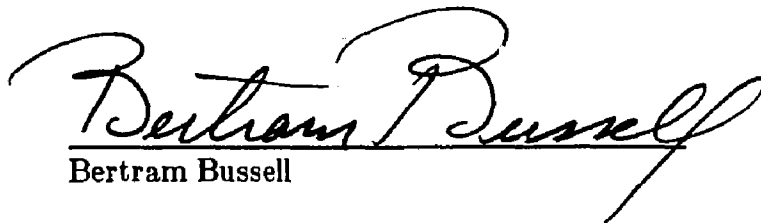
Richard Preston Hooper

1985


The dissertation of Richard Preston Hooper is approved.




Kirby A. Baker



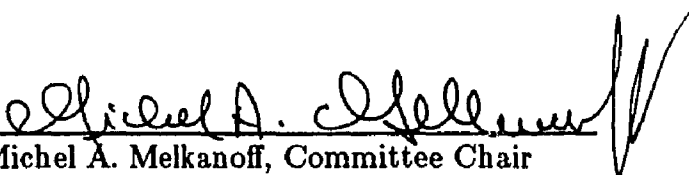
Bertram Bussell



Rakesh K. Sarin



Chand R. Viswanathan



Michel A. Melkanoff, Committee Chair

University of California, Los Angeles

1985

© Copyright by
Richard Preston Hooper
1985

DEDICATION

To my wife,

Denise

whose love and companionship
have sustained me, and
who has shared this experience
with me.

To my children,

Jacquelyn and Douglas

who have brought joy and
laughter into my life.

To my parents,

Fred and Shirley

who have always had faith in me,
and who have been a never ceasing
source of encouragement.

TABLE OF CONTENTS

	Page
1 INTRODUCTION	1
2 CAE/CAD/CAM ENVIRONMENT	8
2.1 Custom VLSI or Gate Array	9
2.2 Printed Circuit Boards	15
2.3 CAE/CAD/CAM Tools	18
2.4 CAE/CAD/CAM Data for Electronic Design	22
2.5 CAE/CAD/CAM Data Exchange Standards	31
2.5.1 Initial Graphics Exchange Specification (IGES)	33
2.5.2 ANSI/IPC-D-350	38
2.5.3 Electronic Design Interchange Format (EDIF)	40
2.5.4 Commercial Systems	41
3 DATA TRANSPORT METHODOLOGY	43
3.1 Media Difficulties	43
3.2 Differences in Electronic Design Data Representation	44
3.3 Database Organization Differences	46
3.4 General Approach	52
3.5 System Architecture	74
3.5.1 Compiler / DBIF	77
3.5.2 Master Data Schema: Generic Predicates	78
3.5.3 Translate Engine	80
3.5.4 Rules for Translation: Knowledge Base	82
3.5.5 Formatter	83
4 PROTOTYPE SYSTEM	85
4.1 Introduction to Prolog	85
4.2 Prototype Elements	92
4.2.1 Process Modules	92
4.2.2 Data Elements	108
4.3 Operational Scenario	116
5 TEST CASES	123
5.1 Hypothetical Cases: DR1 and DR2	124
5.2 TDL to PCB CAD Data Base	138
5.2.1 Data Mapping	139
5.2.2 Forward Rules	144
5.2.3 Reverse Rules	153
5.2.4 Forward Translation Results	157
5.2.5 Reverse Translation Results	161
5.3 CALMA to CIF	164
5.3.1 Data Mapping	164
5.3.2 Forward Rules	177
5.3.3 Reverse Rules	187
5.3.4 Forward Translation Results	189
5.3.5 Reverse Translation Results	193

6 CONCLUSION	198
Appendix A GLOSSARY	203
Appendix B ALTERNATIVE PROTOTYPE IMPLEMENTATIONS	208
Appendix C CALMA STREAM FORMAT DATA	211
Appendix D DBIF REPRESENTING CALMA STREAM FORMAT	217
Appendix E TDL DBIF	225
Appendix F DBIF REPRESENTING CIF FILE	227
Appendix G SAMPLE CIF FILE	233
Appendix H TDL PREPROCESSOR BNF EXCERPT	235
Appendix I SOURCE DBIF FOR CALMA—CIF DATA TRANSPORT CASE	243
Appendix J RULES FOR THE CALMA—CIF FORWARD TRANSPORT TEST CASE	246
Appendix K RULES FOR THE CALMA—CIF REVERSE TRANSPORT TEST CASE	254
Appendix L GENERIC FACTS CREATED FROM SOURCE TDL	260
Appendix M GENERIC FACTS CREATED FROM SOURCE CALMA DATA	262
Appendix N CIF DATA OUTPUT FROM GENERIC FACTS	264
Appendix O GENERIC DATA FROM CIF DURING REVERSE TRANSLATION	266
Appendix P CALMA OUTPUT DATA FROM REVERSE TRANSLATION	269
Appendix Q REVERSE TRANSLATION SYSTEM LOG, CIF TO CALMA	271
Appendix R KEPT FACTS FROM CALMA TO CIF TRANSLATION	273
References	275

LIST OF FIGURES

	Page
Figure 2.1. Sample System Level Block Diagram.	10
Figure 2.2. Typical Electronic Design Flow.	11
Figure 2.3. Multi-level (Hierarchical) Design.	12
Figure 2.4. Sample Schematic Diagram.	16
Figure 2.5. CAE/CAD/CAM Functions.	19
Figure 2.6. CAE/CAD/CAM Data Categories.	23
Figure 2.7. Build-Design Data Base Structure.	25
Figure 2.8. Build-Design Data Base for Sample Schematic.	26
Figure 2.9. Hewlett-Packard DTS-70 Pin-Signal Data Input Syntax.	27
Figure 2.10. Hughes DEC-20 CAD System Data Base.	29
Figure 2.11. Computervision.	30
Figure 2.12. Comparison of DEC-20 and CV CAD Data Bases.	32
Figure 3.1. The Delta Problem.	45
Figure 3.2. Hierarchical Model.	47
Figure 3.3. Network Model.	48
Figure 3.4. Relational Model.	49
Figure 3.5. Use of DBIF to Aid in Data Base Transport.	55
Figure 3.6. Sample Schematic Network.	58
Figure 3.7. Data Base Representation 1 (DR1).	59
Figure 3.8. Data Base Representation 2 (DR2).	60
Figure 3.9. ELKA Diagram for the DR1 Sample Schematic Information Model	62
Figure 3.10. ELKA Diagram for the DR2 Sample Schematic	65
Figure 3.11. Alternative Translation Schemes.	67

Figure 3.12. Translators Needed Using a Generic Data Schema.	67
Figure 3.13. DBIF Encoding of DR2 Data.	70
Figure 3.14. DR2 Relational Data Base.	75
Figure 3.15. System Architecture for Data Transport.	76
Figure 3.16. Sample DBIF Encoding.	78
Figure 3.17. Generic Predicates for Logical Data.	80
Figure 3.18. Generic Predicates for Physical Data.	81
Figure 3.19. Sample Formatter Results.	84
Figure 4.1. Prolog Terms.	86
Figure 4.2. Pseudo-BNF description of the CALMA Stream Format.	94
Figure 4.3. Sample CALMA Layout.	95
Figure 4.4. TDL Preprocessor Compiler Commands.	97
Figure 4.5. Sample TDL Input.	98
Figure 4.6. Translate Engine Processing Flow.	100
Figure 4.7. Translate Engine Processing Flow with Kept Facts.	102
Figure 4.8. CIF Syntax Description.	103
Figure 4.9. Sample DBIF for DR1.	105
Figure 4.10. Prolog Code for the DR1 Formatter.	106
Figure 4.11. Tabular Form of DR1 Data.	106
Figure 4.12. Prolog Encoding of Generic Predicates for Logical Data.	109
Figure 4.13. Prolog Encoding of Generic Predicates for Physical Data.	110
Figure 4.14. DBIF for Sample DR2 Data Base.	111
Figure 4.15. Mapping of DR2 Terms onto Generic Predicates.	112
Figure 4.16. Rules for Transforming DR2 into a Generic Form.	113
Figure 4.17. Different Representations Between DR2 and the Generic Form.	115

Figure 4.18. System Log of the Prototype Execution.	117
Figure 4.19. Generic Data Generated by the Prototype Translate Engine.	119
Figure 4.20. DR1 Target Output Rules.	120
Figure 4.21. Target DR1 DBIF.	120
Figure 4.22. Kept Facts Generated in Transporting Data from DR2 to DR1.	121
Figure 5.1. DR2-DR1 Transport Test Case.	125
Figure 5.2. Prolog Output for DR2 to DR1 Translation.	126
Figure 5.3. Source DR1 DBIF (dr1.dat).	127
Figure 5.4. Source Input Rules for DR1 (dr1in.rul).	127
Figure 5.5. Mapping Between DR1 and Generic Predicates.	128
Figure 5.6. Mapping Between Generic and DR1 Predicates for	129
Figure 5.7. DR1 Data Model.	131
Figure 5.8. Target Output Rules for DR2 (dr2out.rul).	133
Figure 5.9. Mapping Between DR2 and Generic Predicates for	135
Figure 5.10 Target DBIF for DR2.	136
Figure 5.11. Kept Data from the Generic to DR2 Translation.	137
Figure 5.12. Source TDL File in Native Format.	139
Figure 5.13. TDL Schematic Diagram.	140
Figure 5.14. Source TDL DBIF.	141
Figure 5.15. TDL Data Model.	142
Figure 5.16. HPC Data Model.	143
Figure 5.17. Mapping Between TDL and the Generic Form.	145
Figure 5.18. Mapping Between Generic and HPC Formats.	146
Figure 5.19. TDL Source Input Rules.	147
Figure 5.20. Generic Equivalent to TDL Schematic Diagram.	149

Figure 5.21. Rules for <i>Connected</i> in Translating TDL into Generic Form.	152
Figure 5.22. Target Output Rules for TDL.	154
Figure 5.23. <i>Connect</i> Rules from the TDL Target Output Rules.	156
Figure 5.24. Prototype System Log for Forward Translation (TDL to DR1).	158
Figure 5.25. TDL and Generic Kept Facts.	159
Figure 5.26. Target HPC (DR1) DBIF.	160
Figure 5.27. Prototype System Log for Reverse Translation (DR1 to TDL).	162
Figure 5.28. Target TDL DBIF Produced by the Reverse Translation.	163
Figure 5.29. CALMA GDS II Stream Format Data Model.	165
Figure 5.29. CALMA Structure Definition.	167
Figure 5.31. CALMA Array Reference.	168
Figure 5.32. CIF Data Model.	170
Figure 5.33. Mapping from CALMA to Generic Predicates.	172
Figure 5.34. Angular Rotation and Reflection Applied to a CALMA Array Reference.	174
Figure 5.35. Mapping from Generic to CIF Predicates.	175
Figure 5.36. Octagonal Approximation of a CIF Flash.	176
Figure 5.37. Hierarchical Representation of CALMA Source Data.	178
Figure 5.38. Hierarchical Representation of CIF Target Data.	179
Figure 5.39. System Log for the Forward Translation (CALMA to CIF).	190
Figure 5.40. Kept Facts Generated in Translating CALMA into Generic Data.	192
Figure 5.41. The Original Item I_14 and its 90° Rotation.	197
Figure B.1. Original DR2 Representation as a Relation.	209

ACKNOWLEDGEMENTS

There can be no doubt that a doctoral dissertation is the result of more than the work of the degree recipient. I, for one, recognize this and would like to acknowledge the contributions of those who helped me.

I am most grateful to Prof. Michel Melkanoff for his time, interest, support, and encouragement over the many years that I have been his student. No doubt this work wouldn't have been completed were it not for his contribution. I found his advice during the course of the research to be indispensable. He was always quite patient and a very good listener. Throughout the development of the concepts and the prototype, he reinforced the significance of the research, and this was a strong motivating factor in my persevering to the end. I especially appreciated his counsel on the editing of the dissertation, during the final days of its preparation.

I also found the comments and encouragement of the other doctoral committee members to be quite helpful. Many of their suggestions were incorporated into the approach taken in the research. In particular, Prof. Bussell suggested, back in 1981, that I consider a knowledge-based approach to solving this problem. That suggestion had a dramatic effect on the outcome of this research.

There have been many in the UCLA School of Engineering and Applied Science who have assisted me over the years with the administrative matters of graduate study. Special thanks to Brenda Ramsey, Verra Morgan, Rosetta Lindsey, Jamie Brodie, and Rosemarie Murphy for their help. Also, I would like to acknowledge the fine work of Doris Sublette in organizing and maintaining the holdings of our Computer Science Archives.

Also, Marilyn Caro assisted in developing part of the methodology. She built some of the formatters and compilers which were actually used to construct some of the test cases.

There were many people at Hughes Aircraft Company, Radar Systems Group, who gave me the opportunity to do this work. Furthermore, specific individuals shared their insights into this problem and to related issues. I am grateful for their technical contributions. Specifically, I would like to acknowledge Gloria Wilson and Grace Chen-Ellis for their assistance. Above all, I acknowledge the outstanding support of Abe Ansari, my department manager at Hughes. Not only did he share his many years of technical experience as a pioneer in the field of electronic CAD/CAM, but he made it possible for me to continue my studies while working. Were it not for he and the Hughes fellowship program, I would not have been able to pursue this degree.

Both Addison-Wesley and the Calma Company should be acknowledged for their support of academic research. Both granted permission for me to use their copyrighted material as noted in the body of this text.

Lest we forget the importance of computers in our research, I must thank our systems managers and programmers for their assistance. The last year of research was performed using Prolog on the UCLA Center for Experimental Computer Science. I am most appreciative of the outstanding support by Dr. Terry Gray and his staff of the Center. Specifically, Doris McClure and Anne Finestone were quite helpful in assisting me with disk space and in answering system questions. At Hughes, Mike Kimura was quite helpful with the VAX VMS system and in getting me linked to UCLA via phone line.

Pursuing this degree was quite a burden on my family. Words cannot express the sacrifices made by my wife, Denise. She took care of the children, ran the household, and tried to keep order amidst the chaos. Even still, she had time and energy to encourage and reassure me and to share in the excitement when I would make a big break-through in this work. I will be eternally grateful for all of her hard work. Also, a special thanks to Liz and Rex Harris and Marge Fisher who often cared for the children when Denise and I were busy and needed help.

Finally, and most important, I acknowledge that were it not for God paving the way and giving me the strength, this work would never have been possible. Truly, it was a miracle that many of the obstacles were overcome. I shall be thankful to Him all of the days of my life.

VITA

- June 3, 1951 — Born, San Diego, California.
- 1972 — B.A., Mathematics-Computer Science,
University of California, Los Angeles
- 1972-1975 — Teaching Associate, Department
of Computer Science, University of
California, Los Angeles
- 1975 — M.S., Computer Science, University
of California, Los Angeles
- 1975-1977 — Member of the Technical Staff,
Technology Service Corporation,
Santa Monica, California
- 1977-1978 — Senior Scientific Programmer, ITT
Gilfillan, Van Nuys, California
- 1978- — CAE/CAD Systems Development at Hughes
Aircraft Company, Radar Systems Group,
El Segundo, California.
- 1982- — Head, CAD Systems Engineering Section,
Radar Design Automation Laboratory,
Hughes Aircraft Company.

PUBLICATIONS

Hooper, Richard P., "Artificial Pattern Generation," Proceedings of the
Conference on Computer Graphics, Pattern Recognition, and Data
Structures, May 14-16, 1975, with A. Klinger.

ABSTRACT OF THE DISSERTATION

An Application of Knowledge-Based Systems to
Electronic Computer-Aided Engineering, Design,
and Manufacturing Data Base Transport

by

Richard Preston Hooper

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 1985

Professor Michel A. Melkanoff, Chair

The proliferation of computer-aided engineering (CAE), design (CAD) and manufacturing (CAM) systems for electronic design has created a excess of CAE/CAD/CAM database formats. These databases vary from one system to the next, and yet they often carry common information, represented in different formats. In spite of this variation, databases must be transported between systems, since electronic design requires the use of features from several CAE/CAD/CAM systems.

The significance of this work is underscored by other attempts to define a transport method. Most notable is the IGES (Initial Graphics Exchange Specification) standard. Thus far, there are shortcomings with these methods. The IGES standard was primarily developed to handle CAE/CAD/CAM data used to describe mechanical designs not electronic designs. Consequently, IGES does not address all of the data elements of electronic CAE/CAD/CAM systems.

Commercial offerings which translate between CAE/CAD/CAM formats do not translate all of the data. This is because there is usually some data which is unique to each system and cannot be translated. Most commercial translators disregard this data. This is not satisfactory in all cases, since important data relationships are lost in translation.

This dissertation defines a methodology for the transport of databases between independent CAE/CAD/CAM systems. In order to demonstrate the feasibility of this methodology, a prototype system was developed. A non-traditional, expert systems approach was used to solve the problems which have plagued earlier attempts at a data transport method. The prototype was implemented using PROLOG, running under LOCUS/UNIX on a VAX network. The research has been limited in scope to CAE/CAD/CAM systems used for the development of electronic systems as opposed to mechanical systems.

The conclusion of this research is that a this approach can be used to provide a method for transporting data between distinct CAE/CAD/CAM system types. The prototype translation algorithm is driven by knowledge bases which describe the CAE/CAD/CAM data for-

mats and semantics. New systems can be added to the knowledge base with a relatively minor amount of effort.

CHAPTER 1

INTRODUCTION

The purpose of this dissertation is to develop a methodology for transporting data bases of differing form, schema, and content between distinct types of systems for computer-aided engineering (CAE), design (CAD), and manufacturing (CAM). The focus of this work is in the electronic design field in contrast with other fields (e.g., mechanical design). The dissertation describes electronic CAE/CAD/CAM and the requirement for data transport, defines a data transport methodology, presents a prototype system, and analyzes the application of the prototype to several test cases.

Background

The need for increased productivity through the automation of electronic systems development has brought about rapid development of CAE/CAD/CAM systems. Several companies have developed commercial systems providing one or more CAE/CAD/CAM functions for the development of digital/analog circuits, printed circuit boards, and VLSI chips. New systems are constantly becoming available.

Since CAE/CAD/CAM provides a highly competitive market, most companies have worked independently, often without regard for the prob-

lem of interfacing their system with other systems. However, the complete cycle of electronic system development requires that features of several systems be used. No system provides all features necessary for the design, implementation, and fabrication of an electronic system. Several CAE/CAD/CAM functions are necessary, including

- graphics/drafting,
- design capture,
- analysis/simulation,
- placement/route,
- layout and artwork generation, and
- manufacturing aids generation.

Usually these functions will reside on several different computer systems.

Without the benefit of a method for automatically transporting data between systems in order to use each function, the only alternative is to manually re-encode the data necessary to drive each CAE/CAD/CAM system. This approach is expensive since the data must be verified each time the encoding process is repeated. This approach also results in unnecessary delay which defeats a major purpose behind the use of CAE/CAD/CAM. Consequently, there is a strong motivation to translate databases between systems, automatically.

Another motivation for database transport between CAE/CAD/CAM systems is to set up libraries of common data (e.g., graphic symbol libraries). Once established, a single library can be shared by different vendor systems providing that some data translation method is available. Maintenance of libraries is thus reduced by only requiring the update of a single common library.

However, since each brand of CAE/CAD/CAM system utilizes a different database schema/organization, there has been no direct, general data mapping approach available. Typically, specific point-to-point translators are developed by users as needed. The difficulty with this approach is that when the source and/or target data formats are revised by the system developers, the translators must be rewritten. Also, it is difficult to integrate new systems into a distributed CAE/CAD/CAM system network, since translators must be written to move data between the new system and each other system with which it interfaces.

Goal

An alternative data transport methodology is the goal of this doctoral research. The research began about six years ago, motivated by the requirement for data transport within the CAD environment at the Radar Systems Group (RSG) of Hughes Aircraft Company. By investigating the flow of information from the inception of an electronic design to its fabrication, it was clear that several CAE/CAD/CAM systems were required, and that the individual data bases of these systems were unique in both form and content. Six years ago at RSG, there were a few Computervision stations, two CALMA's, a DECsystem-10 (running Hughes-developed CAD

software), and a Hughes schematic capture (graphics) system. At that time, data was not easily transported between systems. For example, data entered on the Hughes schematic capture system was used to facilitate changes to the drawing and subsequent re-draw. Once the design was final, the drawing was released. In order to get a schematic database onto the DECsystem-10 CAD system, the data on the schematic drawing was re-encoded for data entry directly onto the DECsystem-10.

In the years which followed up to the present, this requirement remains. CAE/CAD/CAM technology has undergone many improvements, but the need for a data transport methodology has not been eliminated. The same CAD organization at Hughes RSG has grown and in addition to the systems available six years ago, there are more CV's, more CALMA's, two DECsystem-20's (the DECsystem-10 was converted), two VAX's (running VLSI/VHSIC CAD system), 16 VALID Logic Workstations, and two DAISY systems. Because the number of distinct types of CAE/CAD systems has increased, the data transport problem has become more complex. Attempts at identifying standards within classes of CAE/CAD/CAM data (e.g., schematic, layout, artwork) have succeeded on a limited basis.

Indeed, within the last five years there has been an effort to establish an international CAD data standard, "Initial Graphics Exchange Specification (IGES)." This standard has developed slowly and the major emphasis has been on graphics information and the mechanical CAE/CAD/CAM application. Finally, in 1980, version 1.0 of this standard was accepted and released as an ANSI standard. The intent of the standard is that all systems will provide interfaces to and from their internal

formats into the IGES format. There is no date yet projected for when this might be feasible. Also, there is uncertainty as to whether all commercial systems will adhere to this standard for economic reasons. The emergence of alternative CAD data standards add to the uncertainty.

This year, a new format was introduced for CAE/CAD/CAM data transport. As yet, version 1.0 of this Engineering Design Interchange Format (EDIF) has not been formally released. The inclusion of EDIF in this dissertation occurred recently. The contribution of this potential standard as described in its preliminary release will still not address many of the problems described in Chapter 3. However, as new standards arise, they may add to the growing superset of all data entities which describe electronic designs.

This dissertation will attempt to further clarify the problems of CAE/CAD/CAM data transport, define a solution to the problems, and describe a prototype which demonstrates the feasibility of the concepts developed. To begin with, the CAE/CAD/CAM environment is described in Chapter 2. This includes a description of how data flows between CAE/CAD/CAM processes in both microelectronics (chip) design and printed circuit board design. The categories of this data and differences in its representation between systems are presented. This chapter concludes with a review of existing standards for CAE/CAD/CAM data exchange.

Chapter 3 defines the proposed data transport methodology. First, the difficulties in transporting data between systems are described. Then, the general approach to overcoming the difficulties is presented. Finally, a systems architecture is defined.

Chapter 4 describes the prototype system. The proposed system has been implemented using PROLOG. This chapter contains a brief introduction to PROLOG, a description of the PROLOG programs used to prototype the proposed system, and an explanation of how the prototype operates.

Chapter 5 describes several test cases which demonstrate the feasibility of the prototype. One case is the transport of a hypothetical schematic data base into another, different hypothetical format. The second case involves translating an actual schematic data base (TEGAS description language - a.k.a. TDL) into a Hughes PCB CAD data base format. In this second case, not only is the data base translation demonstrated, but also, a compiler was written and is presented to demonstrate the feasibility of translating an arbitrary CAE/CAD/CAM data language into a generic format. Finally, a third example, two widely recognized, but distinct data formats used for VLSI layout description (CALMA Stream Format and CalTech Intermediate Form) are transformed both ways.

Chapter 6 summarizes the work and states the conclusions which were drawn during the conduct of the experiments.

Appendix A contains a glossary of terms to aid the reader and to clarify the intent of the author.

Appendix B discusses an earlier prototype implementation approach which was attempted using data base technology. This approach was subsequently abandoned due to inherent problems with translation rule representation. This decision led to the use of a knowledge-based systems

approach using Prolog.

Appendices C through G show sample CAE/CAD data bases which were used as test cases for the prototype described in Chapter 4.

Appendix H shows an excerpt of the syntax for the TEGAS Design Language (TDL). The excerpt is included so that the TDL sample of Chapter 4 can be better understood.

Appendices I, J, and K provide a data base and rules used as inputs in a CALMA to CIF test case described in Chapter 5.

Finally, Appendices L, M, N, O, P, Q, and R provide the various outputs from the prototype implementation for the CALMA to CIF test case.

CHAPTER 2

CAE/CAD/CAM ENVIRONMENT

To illustrate the data transport problem, consider a representative CAE/CAD/CAM environment for electronic design. The design process consists of several stages of definition, simulation, analysis, and mechanization until the design goal is reached. The process is iterative and CAE/CAD/CAM tools are utilized at all stages of this design process to perform computationally complex tasks, improve the quality and reliability of the design, and to verify consistency between the various stages of design. Each CAE/CAD/CAM tool/system has unique data input/output formats. A more in-depth look at the electronic design process, the CAE/CAD/CAM tools, the data used by the tools, and data standards will make this more apparent.

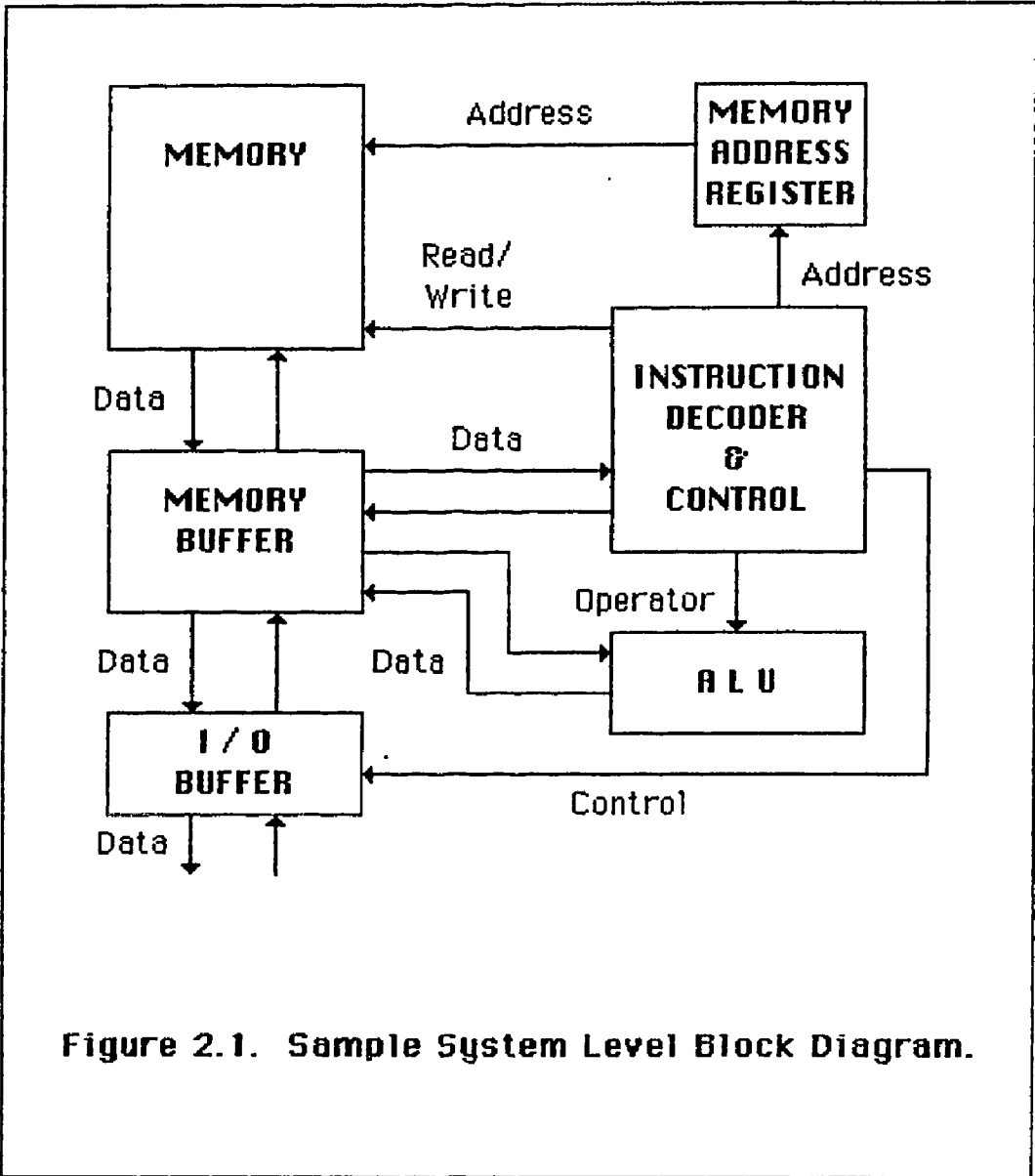
The first step in the electronic design process is the conceptual design of the system. This is a top-level definition which identifies the major functions to be performed by the system. All of the requirements are allocated to one of the various functions. System performance parameters are related to the functions which affect their outcome. Conforming to performance requirements and environmental considerations, the selection of electronic circuit technology will be made (e.g., TTL, Shottky, ECL, CMOS, Custom VLSI, Gate Array, etc.) and the physical packaging will be

established. Once the functional specification is established, each function is defined in terms of processes to be performed, inputs, outputs, and the functions with which they interface. Usually this relationship is illustrated with a system-level block diagram. Figure 2.1 shows an example of a block diagram for a hypothetical computer system. This conceptual definition of the system and top-level functional decomposition is generally performed without the use of a CAE system, although several computer simulations and other analyses are used.

Each system function to be performed is sufficiently described so that the next level of detail can be defined. In the case of the hypothetical computer system (Figure 2.1), the next step of refinement might break down the "Instruction Decoder & Control" into sub-functions. Refinement and functional decomposition continue until a level is reached which can be implemented on a standard physical device such as a gate array, custom wafer, or a printed circuit board (PCB). At this point, the development of the device begins. The development flow consists of the several steps shown in Figure 2.2: logic/circuit design, physical design, manufacture, and test.

2.1 Custom VLSI or Gate Array

When a logic function is to be implemented into a gate array or custom VLSI chip, the design may have several levels of hierarchy. The lowest level refers to primitive cells (macros). These are implemented using analog devices which are built into the semiconductor medium. This hierarchical design style is shown in Figure 2.3. In this example the functions "NAND" and "INV" are primitive cells.



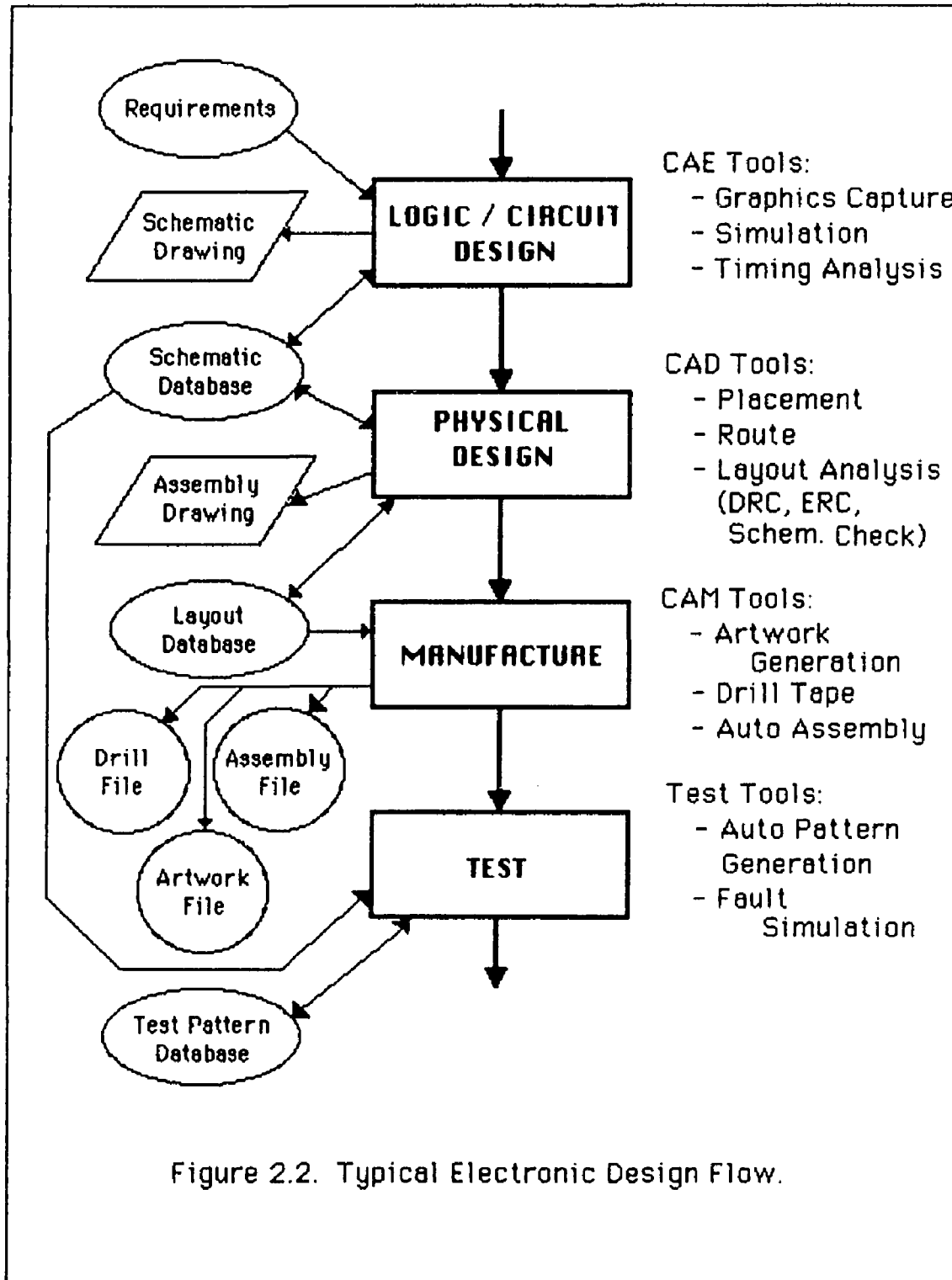
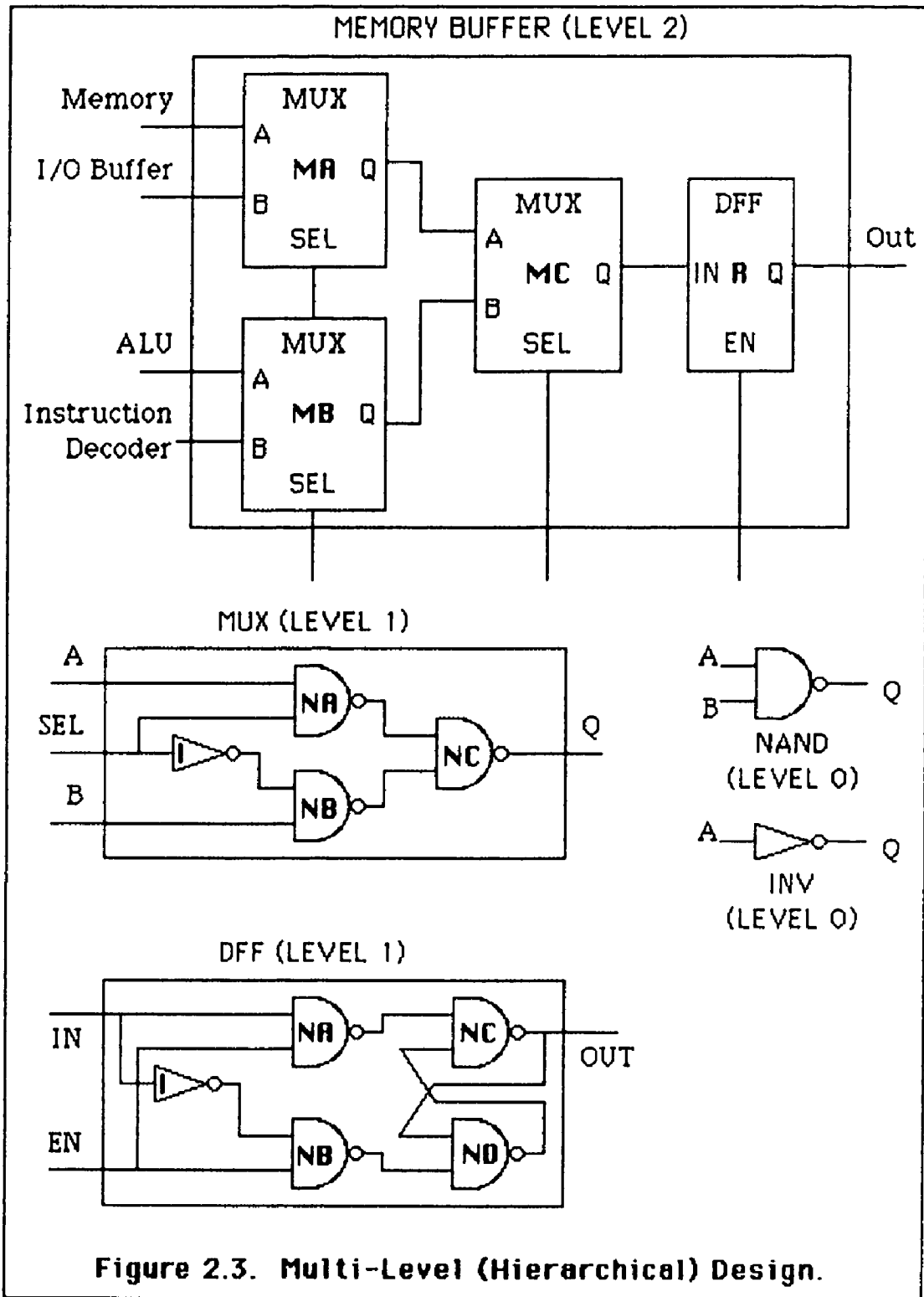


Figure 2.2. Typical Electronic Design Flow.



Typically, when a gate array or custom VLSI design begins, a library of standard primitive cells is formed from basic logic functions as those in the preceding example ("NAND" & "INV"). In some cases, the primitives may be quite intricate, utilizing > 100 transistors. These standard primitives are analog designs whose functions can be verified by a circuit simulator. This simulation will verify the proper voltage levels, voltage transitions, and timing. At the end of this activity, the library of primitive cells will be available as a foundation upon which logic design will be built. The circuit simulation will also generate the necessary data to formulate logic models for higher level logic simulations, working with digital (Boolean) signals rather than analog simulations.

Once the primitive cell library is available, attention turns back to the top-level block diagram. Each function at this level is decomposed into sub-functions, and these sub-functions are sub-divided again. This functional decomposition repeats until the primitive cell level is reached. Once a sub-function can be expressed in terms of primitive cells which have logic simulation models, the sub-function can be verified using a logic simulator.

As sub-functions are simulated and verified, they can be combined to form larger functions. The larger functions can then be verified by simulation. This process is repeated until the entire gate array or custom VLSI device is verified.

In conjunction with logic simulation, timing verification is used at each level in the hierarchy. The signal propagation delays are computed using timing models for all cells and the signal media itself. A network analysis of these propagation delays will verify that signals arrive at the

proper time at the various nodes in the path.

When the gate array or custom VLSI device has a completed logic design which has been verified, then the logic design must be mechanized into interconnected physical cells within a wafer geometry. The CAD tools identified in Figure 2.2 are used to achieve this. First, the primitive cells are placed according to routability criteria, taking into consideration any thermal restrictions and/or timing constraints. Cell placement can be accomplished either manually or semi-automatically.

With the cells placed, signal routing proceeds. Using the available conductor layers in a gate array or custom VLSI device, paths are selected according to the routing algorithm deployed. After automatic routing is finished, there are often signals for which a path could not be found by the router. These "route fails" must be resolved by ripping up routed lines, and manually re-routing the "route fails" and any ripped-up lines. After manual re-route, a design rule check is performed to determine if any geometric spacing constraints, imposed by the circuit and process technology (e.g., CMOS), have been violated. Electrical rules must also be checked to determine layout correctness with regard to electrical parameters, such as loading. Another check performed after any re-routes is a schematic check. This verifies that all signals in the logic design have a physical counterpart in the layout. Once the layout passes all of the post-route checks, a pattern generator tape is produced. This tape is used to guide the various steps in the fabrication process.

When all of the gate-array and custom VLSI devices have been designed and fabricated, they can be combined to construct a printed circuit board, implementing still a larger scale function.

2.2 Printed Circuit Boards In the case of a PC board implementation, this level of functional decomposition constitutes the first level above the components which have been selected by the circuit technology decision at the conceptual design step. Examples of components might be logic elements (e.g., NAND gates, counters, and registers), integrated circuit chips (IC's), or discrete analog parts (e.g., resistors, transistors, capacitors, etc.).

Each PC board is finally defined in sufficient detail so that a completed design can be carried through to fabrication. The PC board is documented in terms of a block diagram showing subfunctional areas (i.e., macros). As the PC board is refined, various subfunctions are designed using components from the set of the selected circuit technology. During this refinement a logic schematic is developed, detailing all of the logical elements and how they are interconnected to perform the specified function (see Figure 2.4).

As portions of the logic are defined, simulation is performed to insure the proper logic functioning of the design before it is implemented using actual components. Timing analysis is performed to insure that there were no incorrect assumptions made about the delays in signal propagation through various logic elements or errors in clocking. Signal loading and drive capacity are also checked to insure compliance with electrical rules. Any problems detected are corrected before the logic design is released for physical design.

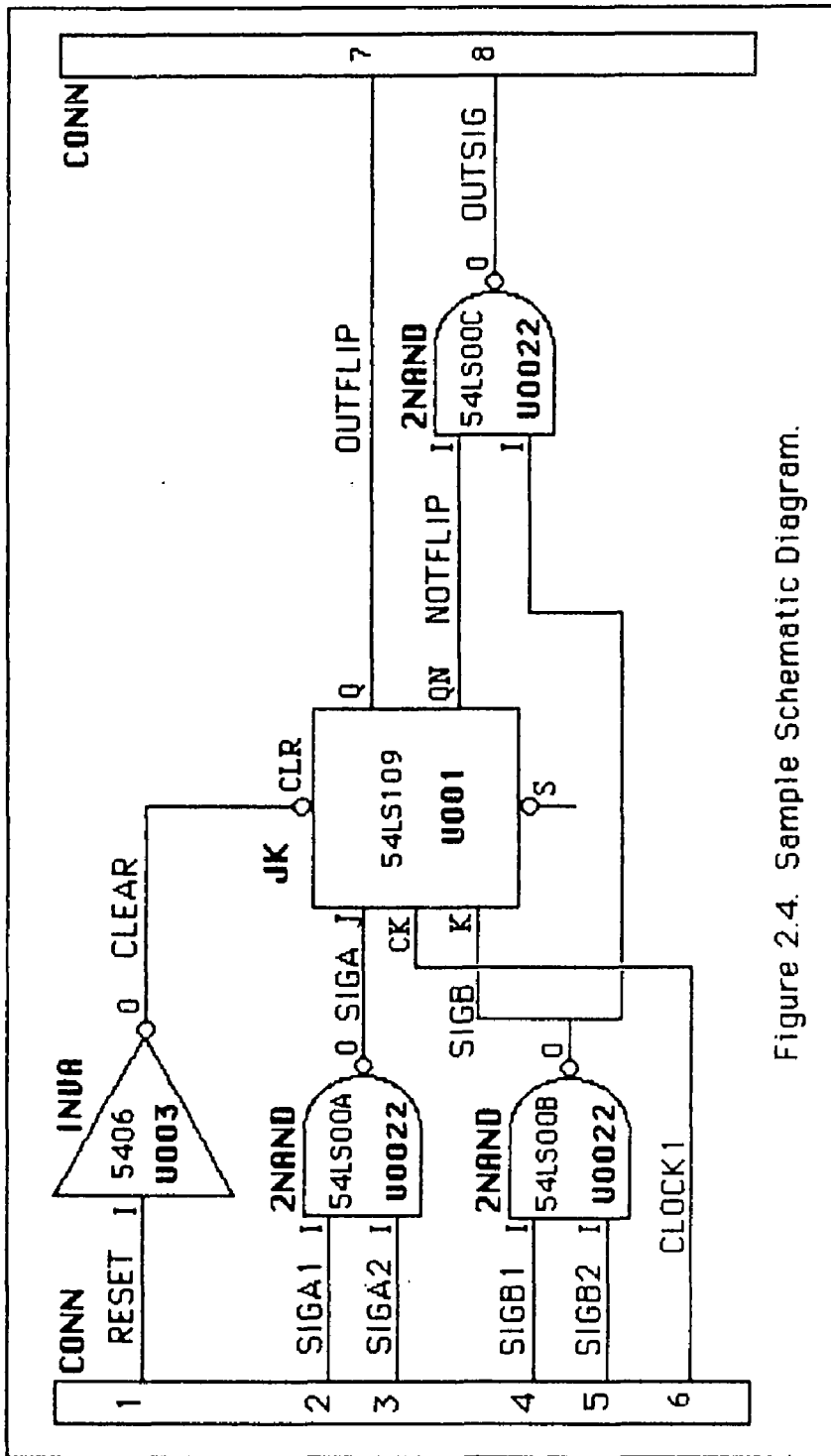


Figure 2.4. Sample Schematic Diagram.

When the PCB design activity is completed, each logic element is assigned to a physical component or chip. In small scale integrated circuitry, chips often contain multiple logic elements of the same type (e.g., 4 NAND gates). After the assignment of logic elements to components, the components are located on the PC board. At this time the placement is analyzed before signal routing to insure correct thermal distribution over the PC board. Once the design passes these validation tests, the next step is to route the interconnections between components upon the remaining PC board area not taken up by components and other obstacles. After route, a final analysis is performed to determine if the paths chosen violate signal length requirements thus causing timing problems or whether there is too much parallelism among adjacent signals to constitute inductive noise problems. Assuming the selected route paths pass these critical tests, the design is complete and ready for manufacture.

Manufacturing consists of etching and drilling the printed circuit boards, generating continuity tests for the etched board, inserting components upon the board, soldering the components in place, and wiring the backplane which contains slots for the individual printed circuit boards. In addition to these processes associated just with the electronic aspects of the design, there are many other fabrication processes necessary to assemble all of the mechanical parts into a final system.

The use of CAE/CAD/CAM tools has had a significant impact on this overall design process. Figure 2.2 shows where data bases and automated tools are used to assist in the electronic design.

2.3 CAE/CAD/CAM Tools

The intent of CAE/CAD/CAM is to off-load any processes which do not require human judgement and experience onto automated processes, i.e., to effectively utilize man and machine resources. Automation is applied to the design and manufacturing cycle in several areas. Figure 2.5 lists several CAE/CAD/CAM functional areas with examples of tools in each area.

Logic Design Aids. From the time a designer receives a requirement specification until a prototype is developed, logic design aids facilitate decision making. To begin with, graphics editors allow a designer to enter, edit and display his evolving logic schematic. As the design proceeds, design aids software indicates any deviation from standards and warns the designer about potential logic errors, producibility problems, and testing difficulties. This feedback speeds up the development cycle by calling to the attention of the designer details that he might have discovered at a less opportune time during development.

Logic simulators and test pattern generators can also be used incrementally as a designer adds more of his logic to the emerging design. These tools help the designer evaluate trade-offs and select the best of design alternatives. In general, the designer gains confidence in his design because the tools identify logic errors and measure how easily the design can be tested once fabricated.

Analysis Tools. Along with logic design aids, analysis tools also raise confidence that the design is correct and will function properly once it is

Logic Design Aids

- . Graphic (Schematic) Editor
- . Logic Simulator
- . Test Case Generator

Analysis Tools

- . Thermal Analysis
- . Propagation Delay & Timing Analysis
- . Noise Margin Analysis
- . Loading Analysis

Physical Design Aids

- . Routing
- . Gate Assignment
- . Component Placement
- . Design Rule Checks
- . Post Route Analysis - Line Lengths, Signal Parallel Runs

Drawings & Reports

- . Assembly Drawings
- . Schematic Drawings
- . Pin/Signal Lists
- . Parts Lists
- . Test Case Data

Manufacturing Aids

- . Numerical Control Machinery Data - Drill, Auto Component Insertion
- . LSI Mask File
- . PCB Artwork File

Libraries/Data Bases

- . Graphics Capture
- . Logic Design (Schematic) Data Base
- . Configuration Management Status
- . Standard Components/Parts Layout Library
- . Layout/Route Data Base
- . LSI Macro Cell Library
- . Drawing Symbol Library
- . Simulation/Timing Models Library
- . Component Packages Library

Figure 2.5. CAE/CAD/CAM Data Functions.

implemented into a physical design. For example, loading analysis will indicate whether there are too many components being driven from a single signal source. Analysis of the physical design before fabrication will often prevent costly errors. For example, once components have been placed on a printed circuit board (PCB), a thermal analysis will indicate whether overheating is likely to occur from the placement of too many "hot" parts in close proximity of one another.

Physical Design Aids. Once the logic design is complete, the mechanical, geometrical, and other non-electrical constraints must be used to determine how to implement the design. The steps involved include placement and route. Once standards for board/wafer geometry and thermal requirements are established and defined in libraries, placement and route can proceed automatically, driven by the logic design data base and a few directives which define router strategies and priorities to be considered during these processes.

Drawing/Report Generation. The output of CAE/CAD/CAM processes is typically a report or a drawing. Following the schematic capture, a pin/signal report is written which contains all of the information pertaining to which components are interconnected. A schematic drawing is the best format to display and verify the logic design data following design capture. The designer can make design changes to the schematic and have a clean drawing produced with the new changes reflected. An assembly drawing can be generated automatically once the components have been placed. This can be used by manufacturing planners to provide instructions on how to build a PC board. In general, CAE/CAD/CAM

processes create, use, and update design data which is then presented in the form of drawings or reports.

Libraries/Data Bases The CAE/CAD/CAM processes require a large volume of data in order to describe the designs under development. Libraries of standard parts or cells and board/wafer geometries are maintained to minimize the amount of data that must be re-entered each time a new design is created. Libraries are created for simulation and timing models, drawing symbols, cell layout patterns, component package specifications, and component artwork (footprint) patterns. These libraries minimize errors and insure consistency, where necessary, from one design to the next.

A design data base is created for each design. This contains data such as the logic network, assignment of logic gates to physical components/cells, placement locations, and signal routing. The data base contains references to standard library parts/cells, board/wafer geometries, and the interconnection of parts. While the collection of data describing a design is referred to as a data base, the data actually may reside in a variety of files and data bases depending upon the CAE/CAD/CAM system. Examples include those listed in Figure 2.5.

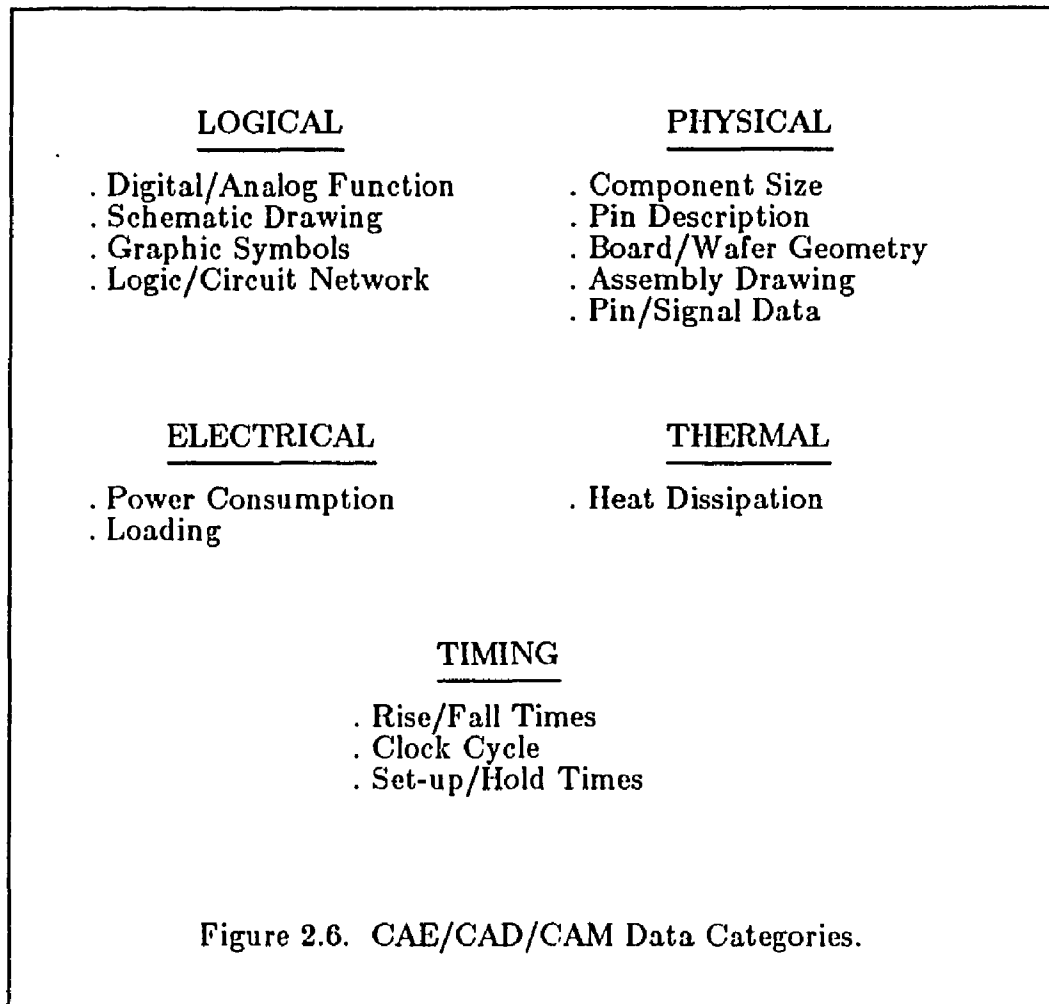
Because of the large number of parameters in the design data bases and libraries, changes in a design after its entry into a CAE/CAD/CAM system can have far reaching effects. For example, the decision to substitute parts in a design can affect several data bases. If a design has been released for manufacture, it means that a significant commitment has been made in time and money to the system configuration in which the design is

embedded. Still, changes are sometimes necessary after initial release and a method is necessary to keep track of which revision of one design is to be interfaced with other designs which together constitute a complete configuration. This is the role of a configuration management system, and some CAE/CAD/CAM systems provide this facility.

Manufacturing Aids. Once the design is complete and the physical implementation routines have obtained a feasible mapping of logic into components and interconnections within the established constraints, manufacturing aids are generated from the design data base. Computer-aided manufacturing relies on these tools to improve the throughput of the factory much the same way CAD tools improve the productivity of engineers. Examples of manufacturing aids include photo-plotter tapes for artwork generation/PCB etch, automatic assembly files, continuity check tapes, and drill tapes. Each of these aids represents a final output derived from the various CAE/CAD/CAM data elements describing the equipment to be built.

2.4 CAE/CAD/CAM Data for Electronic Design

CAE/CAD/CAM data for electronic equipment, such as a radar signal processor, can be broken into several categories of information: logical, physical, electrical, thermal, and timing. In each category, the data type may be textual, numeric, graphic, or special-purpose. Each data type may have several variations such as floating-point and integer numeric representations. Figure 2.6 lists the data categories, and examples of information in each category. Usually the format of a specific category of data varies from one CAE/CAD/CAM system to the next. There are numerous



approaches to establishing a structure for CAE/CAD/CAM data bases. Many of these have been documented, and are referenced here as background. [Ciam76a, Ciam76b, Gutt82, Hask82, Kawa78, Lacr81, Such79, Vall75, Wilm79, Wong79, TDL83, Mead80] At this time there are numerous commercial CAE/CAD/CAM systems, and each maintains a distinct data

base format: CAE Systems, CALMA, Computervision, Daisy, Mentor Graphics, Silvar-Lisco, and Valid Logic to name a few.

It is precisely the wide variety of data formats which has created the problem of CAE/CAD/CAM data transport. Two cases demonstrate the problem.

Case 1. Consider a data base consisting of schematic interconnect information. A sample schematic diagram was shown in Figure 2.4. A data base representing this information describes how the component pins are connected logically. This data can be represented in a variety of ways and is referred to as "pin-signal data", "from-to data", and by other names which connote the data content.

One example of the format this data takes is provided by the Hughes DECsystem-20 CAD system [Wong79]. Here, interconnect data is contained within the "Build-Design" data base. The structure of this data is indicated in Figure 2.7.

Figure 2.8 shows the data content of the build-design data base for the sample shown in Figure 2.4. This data base represents a PCB design by describing the components, their pins, pin function, signal names, and other data used by a variety of CAE/CAD/CAM processes.

Another format for schematic interconnect data is that of an often used test system at Hughes Aircraft: the Hewlett-Packard (HP) DTS-70 system [HP80]. The syntax for this data is shown in Figure 2.9. This form is man/machine readable (ASCII) since it is also designed for direct manual output into the system. In order to reduce error and minimize time, a spe-

ATTRIBUTE SIGNAL ABBREV S TEXT KEYED COL 1 8
 ATTRIBUTE FEED THRU ABBREV FT TEXT COL 9 9
 ATTRIBUTE TEST POINT ABBREV TP TEXT COL 10 10
 ATTRIBUTE CONNECTOR ABBREV CR TEXT COL 11 11
 ATTRIBUTE COMPONENT _ NAME ABBREV CN TEXT
 KEYED COL 12 15
 ATTRIBUTE COMPONENT _ PIN ABBREV CP TEXT
 KEYED COL 16 20
 ATTRIBUTE PAGENUMBER1 ABBREV PN1 TEXT
 COL 22 23
 ATTRIBUTE PIN SWAP ABBREV PS TEXT COL 24 24
 ATTRIBUTE SEQUENCE ABBREV SEQ TEXT COL 25 26
 ATTRIBUTE CRITICALITY _ CODE ABBREV CC TEXT
 KEYED COL 30 31
 ATTRIBUTE ELEMENT _ PIN ABBREV EP TEXT
 COL 32 32
 ATTRIBUTE ELEMENT _ ADDRESS ABBREV EA TEXT
 COL 33 34
 ATTRIBUTE MULTIPLE _ SOURCE ABBREV MS TEXT
 COL 35 35
 ATTRIBUTE SOURCE ABBREV SR TEXT COL 36 36
 ATTRIBUTE ELEMENT _ NAME ABBREV EN TEXT
 KEYED COL 37 42
 ATTRIBUTE ELEMENT _ TYPE ABBREV ET TEXT
 KEYED COL 47 56
 ATTRIBUTE LOAD ABBREV LD REAL COL 57 65
 ATTRIBUTE PIN _ FUNCTION ABBREV PF TEXT
 COL 66 70
 ATTRIBUTE NO CONNECT ABBREV NC TEXT COL 71 71
 ATTRIBUTE SIGNAL _ TYPE ABBREV ST TEXT COL 72 72
 ATTRIBUTE COMPONENT _ TYPE ABBREV CT TEXT KEYED
 COL 81 88
 ATTRIBUTE PINQUANTITY ABBREV PQ TEXT
 COL 105 106
 ATTRIBUTE DOC DATE OF CHANGE COL 142 149
 ATTRIBUTE COMPONENT _ ADDRESS ABBREV CA TEXT
 KEYED COL 161 170

Figure 2.7. Build-Design Data Base Structure.

S	CN	CP	CT	SR	EN	ET	EP	PF
CLOCK1	U001	13	54LS109		U001	JK	J	CK
CLOCK1	CONN	1	CONN	T	CONN	CONN	1	O
SIGA1	CONN	2	CONN	T	CONN	CONN	2	O
SIGA1	U002	1	54LS00		U0021	2NAND	A	I
SIGA2	CONN	3	CONN	T	CONN	CONN	3	O
SIGA2	U002	2	54LS00		U0021	2NAND	B	I
SIGB1	CONN	4	CONN	T	CONN	CONN	4	O
SIGB1	U002	4	54LS00		U0022	2NAND	A	I
SIGB2	CONN	5	CONN	T	CONN	CONN	5	O
SIGB2	U002	5	54LS00		U0022	2NAND	B	I
RESET	CONN	6	CONN	T	CONN	CONN	6	O
RESET	U003	13	5406		U0036	INV	A	I
SIGA	U002	3	54LS00	T	U0021	2NAND	Y	O
SIGB	U002	6	54LS00	T	U0022	2NAND	Y	O

Figure 2.8. Build-Design Data Base for Sample Schematic.

cial purpose translator has been written to dump the DECsystem-20 based CAD system Build-Design data base into the HP DTS-70 form suitable for output into the HP DTS-70 system.

Case 2. Another instance of similar data content in different formats is the DECsystem-20 CAD Build-Design data base versus the Compu-tervision (CV) Electrical Schematic-Printed Circuit (ES-PC) Data Base [CVPC83, CVDB83]. Ideally, the two systems should be capable of data interchange so that the CV system can be used for graphic data entry, display, and editing while the DEC-20 can be used for CPU-intensive computation involved in highly-dense digital printed circuit board physical

```

*HEADER
  MODULE: <module-name>          REVISED: <date>
*LIB <model-lib-1> <model-lib-2>
*MAIN <title>$
*INPUTS
  <input-connector-name>(<pin-loc-1>, <pin-loc-2>, ...
                        <pin-loc-n>)$
  ...
*OUTPUTS
  <output-connector-name>(<pin-loc-1>, <pin-loc-2>, ...
                        <pin-loc-n>)$
  ...
*GATES
<input-connector-name>(<connector-type>)
  <signal-name-1>.<pin-loc-1>
  <signal-name-2>.<pin-loc-2> ...
$
<output-connector-name>(<connector-type>)
  <signal-name-1>.<pin-loc-1>
  <signal-name-2>.<pin-loc-2> ...
$
* <component-name-1>
  <component-address>(<component-type>)
  <signal-name-1>.<pin-1> <signal-name-2>.<pin-2> ...
$
* <component-name-2>
  <component-address>(<component-type>)
  <signal-name-1>.<pin-1> <signal-name-2>.<pin-2> ...
$
.
.
.
* <component-name-n>
  <component-address>(<component-type>)
  <signal-name-1>.<pin-1> <signal-name-2>.<pin-2> ...
$
*END

```

Figure 2.9. Hewlett-Packard DTS-70 Pin-Signal Data Input Syntax.

design algorithms (e.g., routing).

The DEC-20 system data base (shown in Figure 2.7) is organized in a tabular format similar to the relational data model using System 1022™, a data base management system from Software House [Soft84]. In addition to the Build-Design data base, component information is stored in a library contained in two principle data tables as shown in Figure 2.10: the Component Index table and the Component Pin table. This data is used for analyzing designs and for implementing a logical design into a physical layout with associated interconnect etch lines.

The CV ES-PC data base is organized into a Master Index of entities which are represented in detail within a Part Data File (PDF). Each entity represents a graphic element within a larger drawing. Parts are comprised of these elements and may be nested in a hierarchical manner in order to form more complex drawings. In fact, each drawing is just another part which is stored into the PDF and pointed to by the Master Index. Figure 2.11 lists the entities associated with the ES-PC application ^(2.1) on the CV. The orientation of the CV data is toward graphic presentation of the data. Even though there are analysis and some physical design functions which can be performed on the CV system, the magnitude of complexity which the CV system is capable of handling is much more limited in comparison to the DEC-20, due to CPU power differences. The CV system is well suited to editing of logic circuits using graphic representations, with

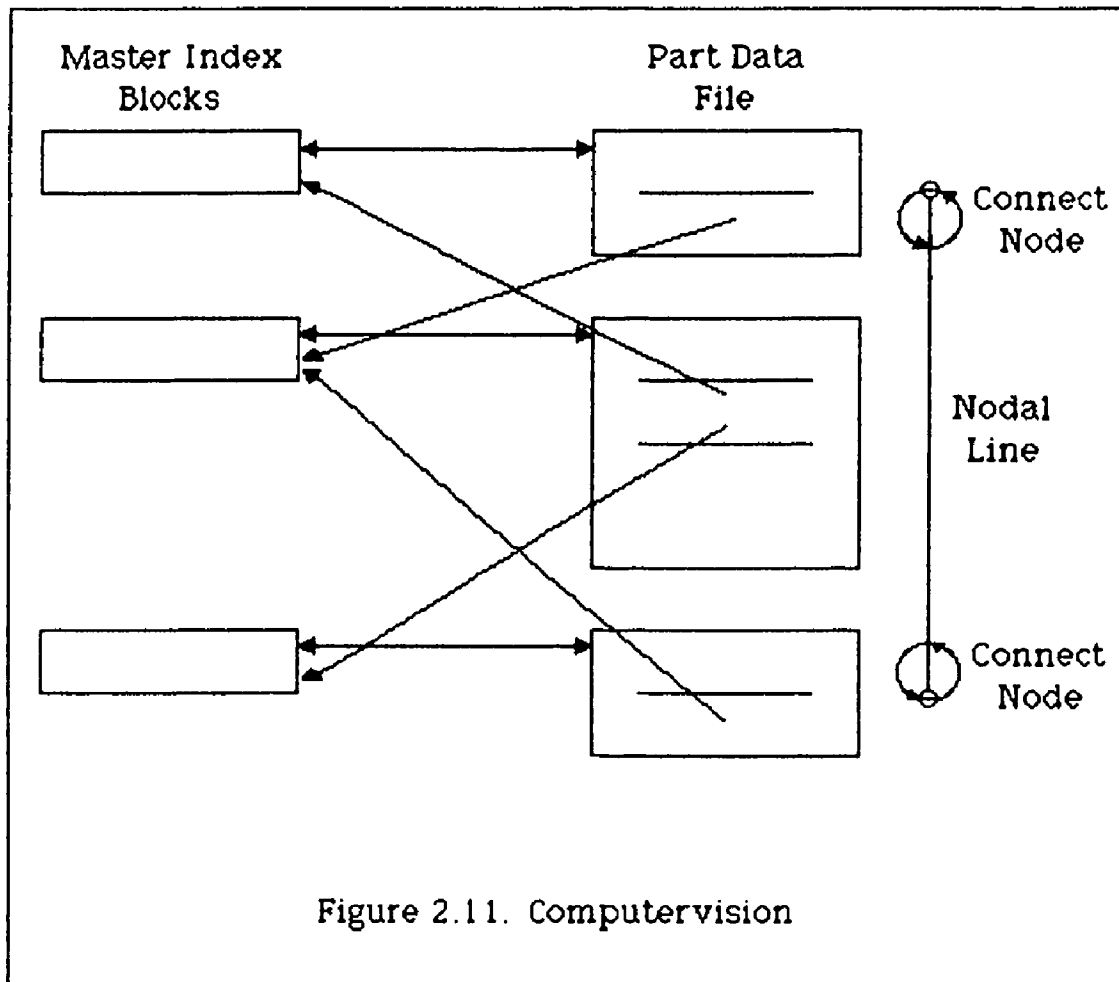
^(2.1) The CV system is general-purpose and supports many other applications such as mechanical parts drawing, surveying, architecture, etc. While each application uses the same data base concept, the entities may be different.

<u>Component Index Data</u>	<u>Component Pin Data</u>
DEFAULT SIGNAL NAME	COMPONENT TYPE
COMPONENT TYPE	PART DESIGNATOR
PART DESIGNATOR	COMPONENT PIN
DESIGNATOR TYPE	CONNECTOR
PACKAGE CODE	TEST POINT
DESCRIPTION	FEED THRU
REMARK	ELEMENT ADDRESS
	ELEMENT TYPE
	ELEMENT PIN
	PIN FUNCTION
	SOURCE
	PIN SWAP
	LOAD
	OVERRIDE CODE
	SIGNAL TYPE
	NO CONNECT

Figure 2.10. Hughes DEC-20 CAD System Data Base.

a straight-forward manipulation language entered via stylus and menu tablet.

The mapping from DEC-20 to CV data bases is complex in that the elements of a component or a schematic are not represented by unique entities on the CV system. Most of the data which should be shared between the two systems is textual in nature (e.g., signal names, pin functions, component names, etc). The use of text on the CV system is rather unrestricted, and as a result, it is only by context that a textual data element on the CV system can be associated with a data element in the DEC-20 data base.



The difficulty is shown graphically by the intersection of the sets of data which each system maintains. This intersection represents only a fraction of either system as shown in Figure 2.12.

As these two cases illustrate, the same conceptual CAE/CAD/CAM data is represented in a variety of ways on different systems. The number of systems is increasing. In fact, it is quite difficult to list all of the commercial CAE/CAD/CAM systems. Adding to the complexity of the problem are custom, in-house systems which many electronics firms have developed to meet their special requirements.

2.5 CAE/CAD/CAM Data Exchange Standards

Because the data transport problem is pervasive and wide-spread, a number of attempts have been made to standardize on neutral data base formats. However, these data standards do not completely address the data transport problem. The job of translating data to and from a standard is left up to the originator and receiver of the CAE/CAD/CAM data format being transported. The problem of what to do with data which cannot be expressed in the standard is not addressed.

Among the attempts at standardization are the Initial Graphics Exchange Specification (IGES), ANSI/IPC-D-350, and most recently the Electronic Design Interchange Format (EDIF). The preface to the EDIF specification [EDIF84] identifies the problems encountered by those attempting to define a standard:

"While many interchange formats and hardware description languages have been developed over the past decade, each has

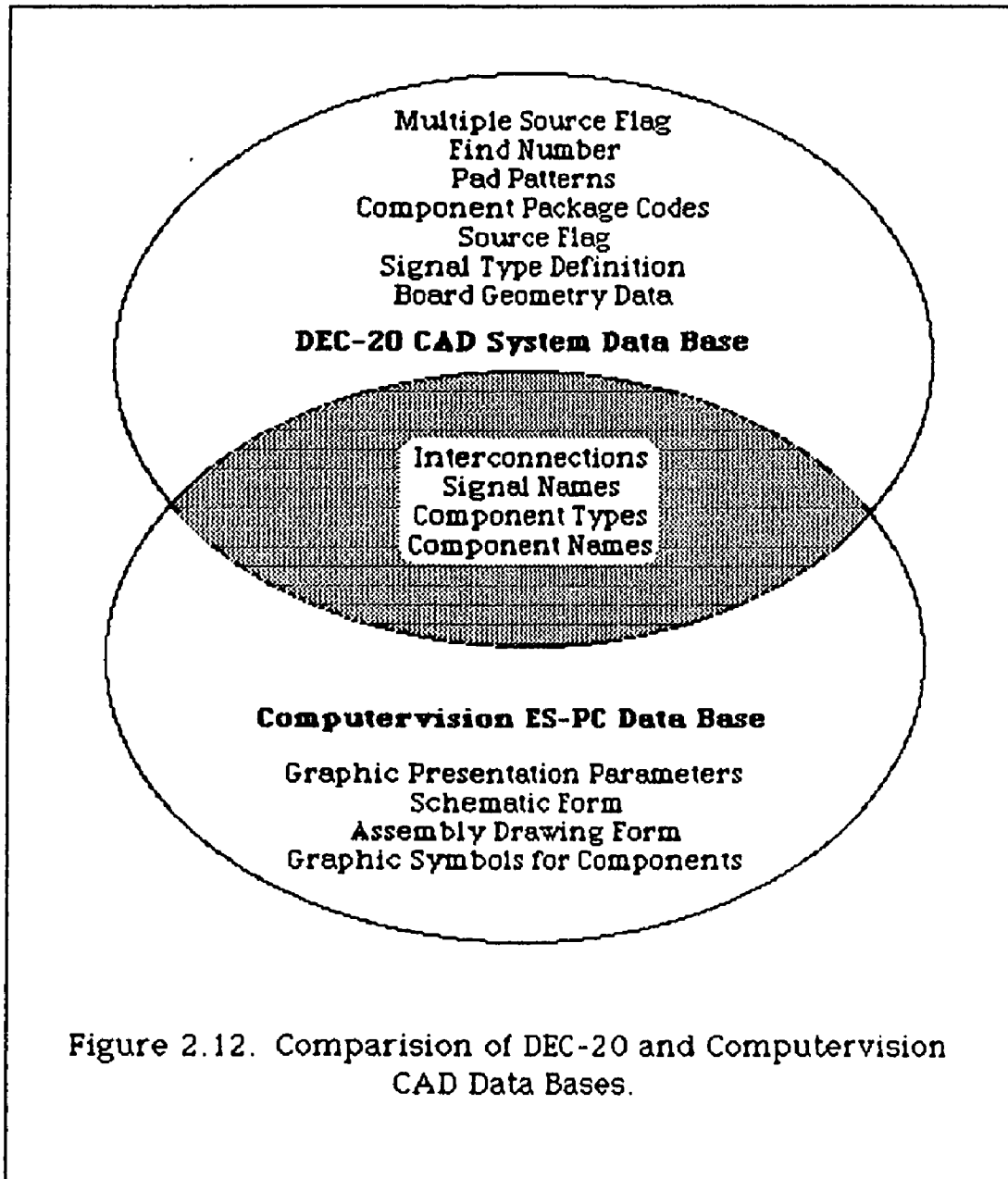


Figure 2.12. Comparison of DEC-20 and Computervision CAD Data Bases.

suffered from one or more of the following drawbacks:

Narrow focus, ...
Proprietary, ...
Difficult to Implement, ...
Difficult to Extend, ..."

A brief description of these standards will bring to light these drawbacks and illustrate others as well.

2.5.1 Initial Graphics Exchange Specification (IGES)

The formulation of IGES dates back to late 1979. The first IGES concepts stemmed from a joint meeting of the Air Force, Army, Navy, and the National Bureau of Standards (NBS) held at the National Academy of Sciences on October 11, 1979. Presentations by CAD/CAM systems vendors, corporate systems designers, and standards groups pointed to problem in data transport between CAD/CAM systems. At this time and since then, the emphasis has been on 3D graphics CAD/CAM systems for mechanical design. It was generally agreed that an initial graphics exchange specification was needed immediately. [IGES80]

Later in 1979, a committee was formed between representatives from the NBS, Boeing, and General Electric. The Boeing and General Electric corporate CAD/CAM exchange systems were selected as initial systems upon which IGES was formed. Subsequent to its first release, IGES became an ANSI standard (ANSI Y14.26M-1981) issued on April 15, 1982.

Due to the rapid formulation of IGES, several misconceptions about the intent of IGES developed even prior to its first release. In an attempt to clarify IGES, the technical committee offered a list of statements describing IGES in the Introduction of the first release. Several of these

statements point to inherent limitations built into the standard:

- IGES is designed with the technical aspects of several CAE/CAD/CAM systems in mind. Thus the translation from vendor systems to IGES and vice versa will not be one for one, but should be feasible.
- IGES is not a complete spec of all the data in all CAE/CAD/CAM systems. Thus, there may be a loss of data or structural information in the translation to and from IGES.
- IGES is based on the Boeing CAD/CAM Integrated Information Network, the General Electric Neutral Data Base, and a variety of other data exchange formats which were given to the committee.
- IGES is the best specification that could be produced in the time frame permitted. While it is not a copy of any of the exchange formats presented to the committee, it has the advantage of the experience and knowledge gained in their production and use.
- IGES is a set of geometrical, drafting, structural, and other entities. Thus it has the capability to represent a majority of the information in CAD/CAM systems.
- IGES is extensible. Several definition mechanisms have been provided to permit IGES to be expandable. A working com-

mittee has been set up to coordinate expansions and to correct errors.

- IGES is not designed for the technical aspects of any one of the currently available CAD/CAM systems.
- IGES is not perfect, or the solution to all data exchange problems between CAD/CAM systems.
- IGES is not a carbon copy of any of the exchange formats given to the technical committee.

It is clear, that the committee was attempting to deliver a spec in a short time-frame, and that they hoped to provide an extensible framework which would allow future additions to cover any initial shortcomings.

The initial format of an IGES file was a numerically sequenced set of 80-column card images in ASCII. As would be expected with graphics data bases, this file format resulted in very large IGES files. This problem resulted in the second release of IGES including an optional binary representation of an IGES file. The structure of the binary format is essentially the same as the ASCII format.

An IGES file consist of 5 sections: Start Section, Global Section, Directory Entry Section, Parameter Data Section, and Terminate Section. The Start Section contains comments about the IGES file and the part represented. The Global Section describes machine dependent characteristics used in an IGES file, various global delimiters, and units. The Directory Entry Section contains entity descriptions and pointers to parameters, other entities, line fonts, levels, views, translation/rotation matrices, status

flags. Entities are of three categories:

- Geometry: Circles, Lines, Conics, Points, etc.
- Annotation: Dimensions, Notes, Arrows, Witness Lines
- Structure: Associativity, Line Font, Macro, Subfigure, Text Font, Drawing, Property, and View.

The Parameter Data Section contains end points, transformation matrix values, angles, and pointers. In general, all parametric data needed to describe entities defined in the Directory Entity section is contained in the Parameter Data section. The Terminate Section contains the card counts for each section and serves as a checksum of sorts to verify that the file has been transmitted properly.

While the basic structure of an IGES file doesn't appear too complicated, there are 20 geometric entities, 13 annotation entities, 12 structural entities, plus a macro capability. The majority of the entities (> 75%) are geared for the mechanical CAD/CAM field, where the emphasis has been since the early days of the IGES inception. The specification as issued (both versions 1.0 and 2.0) gives only one, incomplete example of how IGES is applied to electronic CAD/CAM systems. Much of IGES is oriented toward describing drawings and not to describing the content of the drawings. There is virtually no support for CAE data used in simulation and analysis models. Presumably, CAE is treated as a user extension.

There is an IGES subcommittee which has been actively addressing extending IGES for electronics data as recently as February 1984. Working documents and meeting minutes indicate that there are shortcomings in

IGES version 2.0 which could be resolved in a number of ways, including adding more entities to describe schematics. It appears that no single approach has been approved and that further committee discussion is required.

As regards the integration of IGES with other standards, the IGES Electrical/Electronic Subcommittee is aware of the existence of ANSI/IPC-D-350. In February 1984, the Subcommittee suggested that the IPC standard should be regarded as any other vendor. ^(2.2) Also, it was suggested that IGES should consider its role as a super-set of CAD/CAM data (including IPC). At the other extreme, there seems to be no formal or informal recognition of EDIF by the IGES Subcommittee to date. The EDIF effort began in late 1983 with the 0.8 version of the specification being released May 14, 1984 and version 0.9 on July 16, 1984.

Aside from the lack of development in the area of electronic design data, there are a number of other IGES deficiencies. Scowen [Scow82] indicates a fundamental problem with all systems which are based upon the transfer of drafting data. The information defines drawings of a product and not the product itself. Scowen argues that the "full benefits" of a standard can be realized only

"by complete integration based on a model of the product itself rather than a model of drawings of the product."

(2.2) Unpublished minutes of the IGES Electrical/Electronic Subcommittee Meeting of February 9, 1984.

Another problem Scowen identified is that some objects can be represented in several ways in IGES. For example, a rectangle may be a composite curve, a bounded plane, or a subfigure. The lack of a canonical representation for objects leads to difficulty in translating data to another system and then back again. In fact, it is difficult to check whether two IGES files are equivalent and describe the same object.

An early criticism of IGES was that IGES files were large. This was to a large extent a problem with the ASCII 80-column card image format. With version 2.0, IGES now has a binary format as well. Another problem with the IGES file format is that there is no facility for making easy corrections since all internal pointers are absolute. Consequently, if a change needs to be made to a design, a new IGES file must be written. If an IGES translator is expensive to run (which is likely), then the purpose of facilitating data transport is defeated.

It is clear that while IGES is a tremendous accomplishment, it is not a panacea.

2.5.2 ANSI/IPC-D-350

The purpose of the ANSI/IPC-D-350 [IPC77] standard is to provide a uniform means of describing printed wiring boards in a digital (80-column card) form. All aspects of printed wiring board designs can be documented using this standard including copper type, dielectric used, plating thickness, locations of lines, line widths, orthogonality restrictions, pads, drill holes, and layering schemes.

An IPC file contains 5 sections: Parameter Records, Comment Records, Feature/Location Records, Complex or Composite Records, and an End of Job Record. Parameter records include JOB, DIM, UNITS, TOL, LAYER, and IMAGE records which describe global values needed to interpret the IPC file. Within the comment record are included

- List of Features,
- Finished part descriptions - fabrication materials and special instructions,
- Line Record Identification - structure of line records, and
- References to specification documents.

The Feature/Locations records use a pseudo-assembler language to describe

- Op codes for continuation records, line records, point records, and annotation or lettering records;
- Features Description Areas which depend upon the op-code but include dimensions, layer codes, signal identification, hole size, annotation character height; and
- Location Description Areas which contains x-y data.

The Complex or Composites records contain op-codes for subroutine definitions and subroutine calls.

While the IPC standard is comprehensive in its treatment of PC board related data, the bulk of the electronic CAE/CAD data is not

covered by intent. Thus, this standard suffers from the "Narrow focus" drawback identified by the EDIF formulating Committee, although ANSI/IPC 350B was not specifically referenced.

2.5.3 Electronic Design Interchange Format (EDIF)

This format is the newest attempt to develop a universally acceptable standard for the transport of electronic design data. As indicated at the beginning of this section on data standards, EDIF was undertaken because shortcomings were perceived with all previous efforts. Several predecessors were mentioned by name, but IGES wasn't included. It is unlikely that IGES would be unknown to a committee with CAE/CAD/CAM system houses represented. It can only be assumed that the IGES effort in the area of electronic data is not regarded to be of importance to the EDIF committee.

The version 0.9 EDIF is quite comprehensive in its inclusion of all classes of electronic design data. The classes described in section 2.4 are addressed and to a more extensive level of detail in EDIF. The basic element of design is the "cell". Several "views" of a cell provide data regarding the schematic, symbolic layout, masks, behavior, and documentation. Provisions are made for cell libraries and multiple technologies.

From a structural point of view, the EDIF file is expressed in LISP. Provisions are included for defining variables and macros, as well as conditionals. The EDIF syntax is quite extensible and allows for growth.

EDIF is oriented toward gate arrays and custom VLSI designs and provides for CAE as well as CAD/CAM data. However, some of the CAE data (e.g., simulation models) are expressed as comments. Clearly, EDIF is new and there are a number of features which are only developed superficially (e.g., the documentation view). Also, there have been a number of criticisms about some of the features of EDIF.

The source of the criticisms are the minutes of the July 16, 1984 public meeting on EDIF. Some have objected to the LISP syntax for efficiency reasons. The gate array description features of EDIF are unclear and confusing. The number of Test States is inadequate. The Behavioral description lacks power. There are weaknesses in the mapping among the logical, physical, and behavioral interfaces. In some cases information is duplicated as in the redundant definition of ports between views. There are other minor and subtle flaws which were discussed at the meeting. The intent of the discussion was to identify problems which need correction before the version 1.0 EDIF is released.

Because the EDIF effort is new, the issue of what to do in cases when data cannot be represented in EDIF has not really been discussed. The assumption made by the committee is that the user extension feature will cover these cases.

2.5.4 Commercial Systems

In addition to standards which are provided to assist in data transport, there has been at least one commercial offering. Octal, Inc. of Mountain View, CA announced as early as October 1981 a product which con-

verts CALMA drawing data bases into Applicon data bases. That particular system was reported ^(2.3) to log components of the data base which were "not exactly converted" due to differences in data representation.

Again as in the case of data standards, the problem of data which doesn't translate is acknowledged, but no obvious treatment prescribed. This problem among others is described in the next chapter and a methodology for CAE/CAD/CAM data transport is presented.

(2.3) The Harvard Newsletter on Computer Graphics -- Vol. 3,
No. 20 - October 19, 1981.

CHAPTER 3

DATA TRANSPORT METHODOLOGY

The limitations in the present data exchange standards and in commercial translation software underscore the difficulty in solving the CAD/CAM data transport problem. There are several obstacles to be overcome in providing a data path between CAD/CAM systems.

3.1 Media Difficulties

Since each CAD/CAM system usually resides on a different computer system, data must be transported between computer systems. Disks are problematic since there are several format standards. Also, not all systems have a removable disk media. Punched cards are cumbersome and outdated; many modern computer systems do not handle punched cards. A third option would be to use networks. This is by far the most direct method, but as with disks, there are many network standards. Not all computer systems have compatible network interfaces with other vendor computer systems. The final option is to use magnetic tape. While there are a variety of tape formats and densities, most candidate systems support industry compatible format tapes.

3.2 Differences in Electronic Design Data Representation

While there is, in theory, commonality across systems which perform the same CAD/CAM function, electronic design attributes are represented differently in different CAD/CAM systems. The design attributes describe all aspects of the electronic system to be developed. As described in section 2.4, this data can be categorized as logical, physical, electrical, thermal, or timing-related. Examples of attributes with differing representations are logical pin names and functions, the names of logical elements and signals, and generic component types. Specific cases of differing representation were presented in section 2.4.

The Delta Problem

Also, in comparing any two similar CAD/CAM systems, there is usually data which is stored in one system without counterpart (in any format) in the other system. We can refer to this anomaly as the "delta" problem (delta, meaning difference). This problem is not easily solved and has been often overlooked in translators to date. Consider the scenario represented in Figure 3.1. As this example shows, sometimes CAE/CAD/CAM data needs to be transported from system A to B and then back again to system A. When this is done, it is important that data relationships not be lost. If a given data element D_1 is related to D_2 in system A, $R_A(D_1, D_2)$, and D_2 cannot be translated to system B, then we need to store the relationship $R_A(D_1, D_2)$, separately, before translating. Upon return from system B, item D_2 should again be associated with item D_1 according to $R_A(D_1, D_2)$.

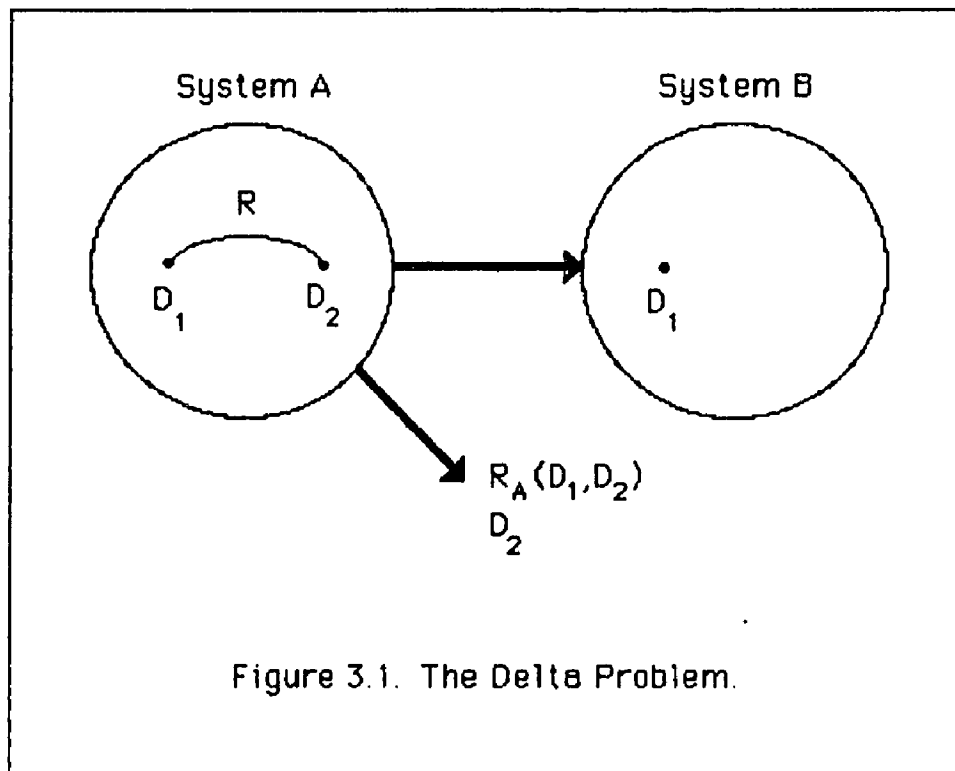


Figure 3.1. The Delta Problem.

This problem is not straight-forward since once data is sent to system B from system A, it may be operated upon, resulting in D_1 being deleted or changed. Consequently, it is not obvious what meaning does $R_A(D_1, D_2)$ have when the database is transported back from system B to A.

An example of the delta problem was shown graphically in Figure 2.12, which compares the data elements of the Computervision system with the Hughes PWB CAD system. This problem will be addressed again in Chapter 5 when actual transport test cases are presented.

3.3 Database Organization Differences

Another problem is that each CAD/CAM system has a unique schema or organization for design databases. Conventional data base theory [Date82] has defined 3 basic schema types: the hierarchical, network, and relational models. The *hierarchical* model is represented in Figure 3.2. Each node has exactly one parent (except the top or root node) and may have zero or more child nodes. The *network* model resembles a graph (Figure 3.3) and is less restrictive than the hierarchical model. The *relational* model takes a different view of data. Data is organized into tables with columns which represent different data types and rows which contain instances (or -tuples) of data (see Figure 3.4).

In addition to these basic schema types, there are other database organizations. In fact, CAE/CAD/CAM system vendors often use the term *data base* when, in reality, the correct term would be *file*. Consequently, there exist a multitude of custom data base types in this field. Further-

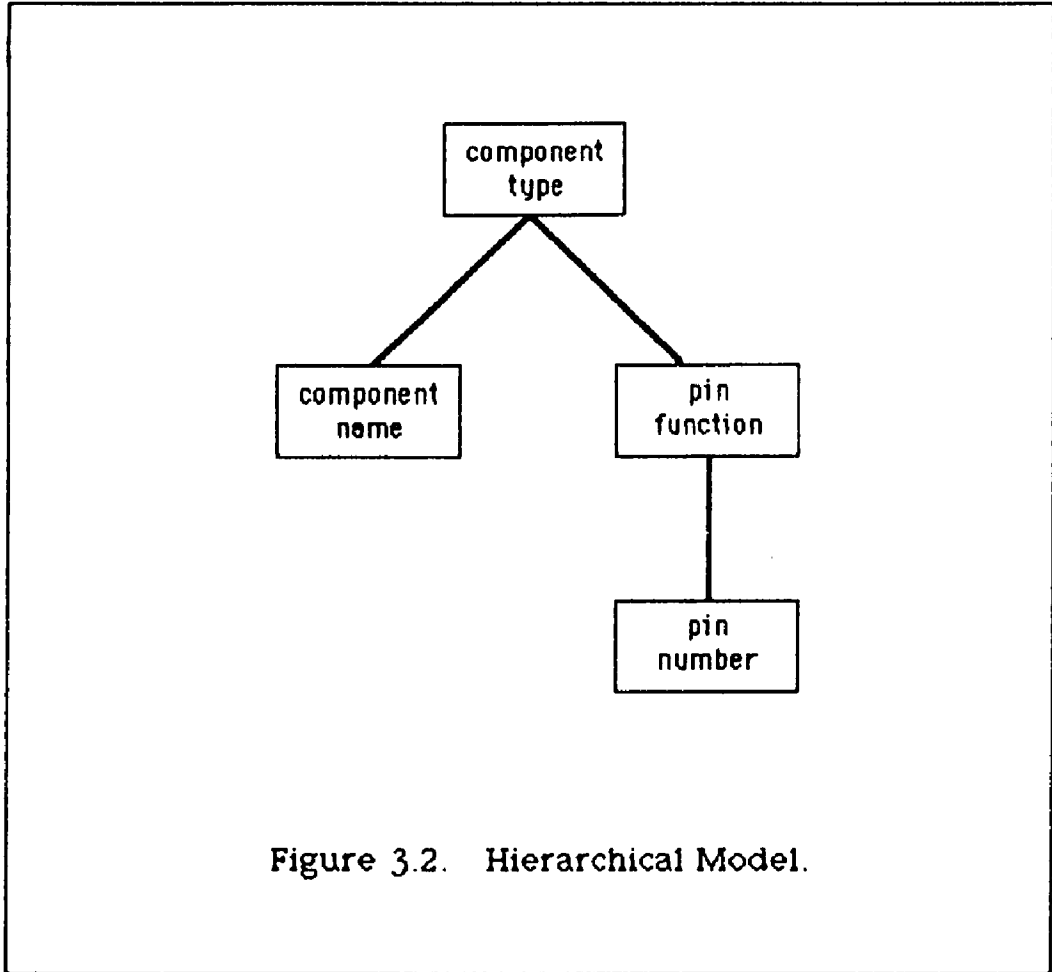
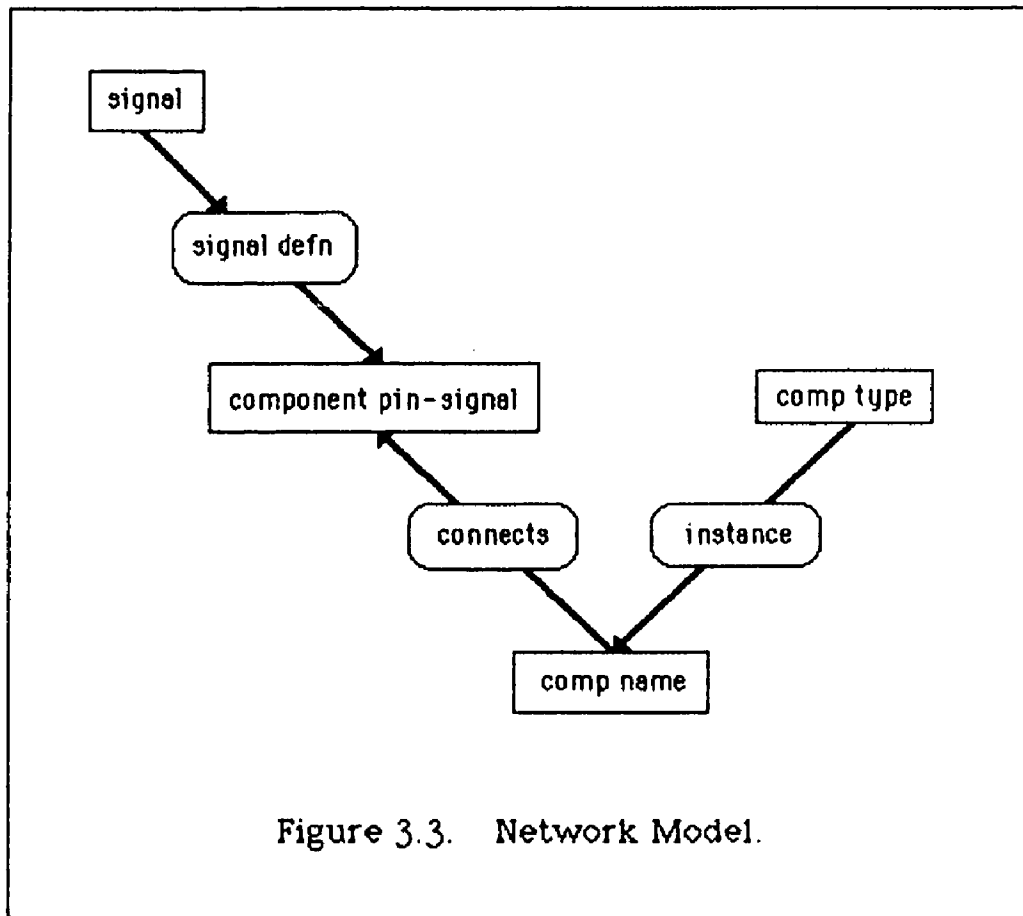


Figure 3.2. Hierarchical Model.



Component Name	Component Type
~~~~~	~~~~~
~~~~~	~~~~~
~~~~~	~~~~~

Component Type	Pin #	Pin Function
~~~~~	~~~~~	~~~~~
~~~~~	~~~~~	~~~~~
~~~~~	~~~~~	~~~~~
~~~~~	~~~~~	~~~~~

Component Name	Pin #	Signal
~~~~~	~~~~~	~~~~~
~~~~~	~~~~~	~~~~~
~~~~~	~~~~~	~~~~~
~~~~~	~~~~~	~~~~~

Figure 3.4. Relational Model.

more, each of these models can be implemented (i.e., physically stored) in a variety of ways. This variety adds to the problem of data base transport between distinct CAE/CAD/CAM system types.

Also, the complete semantics or meaning of a data base is not always represented in just the data model alone. Consider an example based upon the relational data model. Figure 2.4 showed a sample schematic diagram. A non-normalized relational data base representing this information is shown in Figure 2.8. One important constraint on schematic data is that no two signals be tied (shorted) together. This restriction is referred to as an *integrity constraint*. In terms of the relational data base (Figure 2.8), it means that no two rows can have the same values for CN (component name) and CP (component pin) and have different values for S (signal). There is no provision for storing this constraint in the relational data base management system (DBMS), but it is still a part of this data base's semantics. To overcome this inadequacy in many DBMS's, a data base verification program is written to insure that the data complies with all integrity constraints. Following updates to the data base, the verification program is executed to determine if integrity violations exist.

To show how the same conceptual information is represented differently on different systems, consider two examples: 1) logic interconnection databases in both the Hughes Aircraft CAD system and Computer-  
vision, and 2) layout databases in both the Hughes CAD system and the CALMA system.

*Hughes CAD System Build-Design Database [Wong79].* This database describes both the logical and physical data associated with a logic network and its printed circuit board (PCB) implementation. This is essentially a relational data model. This CAD systems runs on a DECsystem-20 using the System 1022™ database management system.

*Computervision Schematic Database [CVPC83, CVDB83].* Like the Hughes Build Design Database, this database describes the logical and physical data associated with a circuit. However, unlike the Build Design Database this database also includes graphic schematic information. All components used in the circuit are represented in terms of graphic elements (e.g., line segments, arcs, circles). Signals represent not only named interconnections between component pins, but also lines which appear on the schematic. This database uses a hierarchical model with all of the logic network data intermingled with the graphic data.

*Hughes CAD PCB Layout Data [Wong79].* The layout data for the CAD PCB system consists of a matrix representing grid points in both  $x$  and  $y$  dimension of the printed circuit board. For each grid point, there is an indicator designating whether that grid point contains etch, a via, a drill hole, or is vacant. There is a third dimension to the matrix representing the layer of the printed circuit board. A board measuring 5" x 9" with a 50 mil grid would have 100 ( $=5/.05$ ) by 180 ( $=9/.05$ ) elements in its matrix representation.

*CALMA IC Layout Data [CALM82].* Unlike the Hughes PCB Layout matrix, the CALMA IC layout representation uses rectangular boundaries and paths to describe the layout pattern. Interconnections are represented

as paths; vias, pads, or vacant regions are represented as boundaries. This database stores actual coordinate endpoints for these geometric entities; essentially a vector representation. The Hughes PCB Layout database, on the other hand, uses a pixel or raster data representation to store the layout patterns.

### **3.4 General Approach**

It has been shown that the CAE/CAD/CAM data transport problem consists of transport media difficulties, difficulties arising from differences in data representation, and difficulties due to data base organization differences. In order to overcome these difficulties, several elements in a transport methodology are necessary:

- media conversion,
- database intermediate format (DBIF),
- compilers/formatters,
- generic/neutral data base, and
- a translation mechanism.

Basically, the problem can be approached by providing mechanisms for 1) translating the various formats into a standard format and 2) translating between different sets of data base content, each represented in standard form.



Conventional translations schemes have been implemented with procedural languages. Consequently, they were cumbersome to produce the first time and difficult to change. This is because much of the knowledge of how to translate between data bases is embedded into procedural language statements which must be re-coded, often affecting the logic flow of the program as well. An alternative translation scheme is a knowledge-based systems (KBS) approach.

This approach is also referred to as an expert systems approach. Both terms apply equally, but KBS emphasizes the collection of rules and the architecture of the system. *Expert systems* emphasizes the fact that the content of the knowledge base is obtained from experts who are very familiar with the problem area. This dissertation emphasizes the architectural approach of the prototype system, therefore the KBS designation has been used. The complexity of the subject data also suggests that the methodology could be described as an expert system. Henceforth, the system and methodology will be referred to as a knowledge-based system.

A knowledge-based system differs from conventional procedural systems in that the algorithm for performing a task is characterized by a database of knowledge rather than by procedural statements. Using a KBS the procedure is to apply knowledge in the forms of rules or assertions (e.g., "If x then y") and to produce the desired results by inference. There are numerous references on the subject of knowledge-based systems (KBS) [McCa83, Wood83, Brac83a, Brac83b, Mylo83].

A knowledge-based systems approach to the data transport problem improves most elements of the transport methodology.

*Media conversion.* This is the very basic problem of overcoming computer word length differences, 7-bit ASCII, 8-bit ASCII, and EBCDIC character code conversions, magnetic tape formats, etc. While it is necessary that this problem be addressed, it has been solved routinely in the past. This subject will therefore not be addressed in this dissertation.

*Database intermediate format (DBIF).* Assuming that the media conversion problem has been addressed, a common *format* is necessary so that when data base *content* (including logic schema, integrity constraints, etc.) is translated it can be accomplished without regard for format differences. There are many possible formats which could be used as the DBIF. For example, a relational data base could be used for this purpose. For each CAD/CAM data base type there would be a set of relations which would constitute its DBIF representation. The DBIF content (e.g., relation domains, number of tuples, etc.) would be different for each CAD/CAM data base type. The translation between CAD/CAM database types would require rules which transform one set of relations into another. Having each data base represented in DBIF (in this case, a set of relations), translations would be performed independent of specific CAD/CAM database formats. Figure 3.5 illustrates the use of the DBIF as an intermediate translation step. In translating from data base type 1 to data base type 2, first type 1 is converted into DBIF 1. Next, a generic translation engine is used to translate from DBIF 1 into a generic DBIF and then into DBIF 2. Then, DBIF 2 is converted into native data base type 2 format. In a like manner,

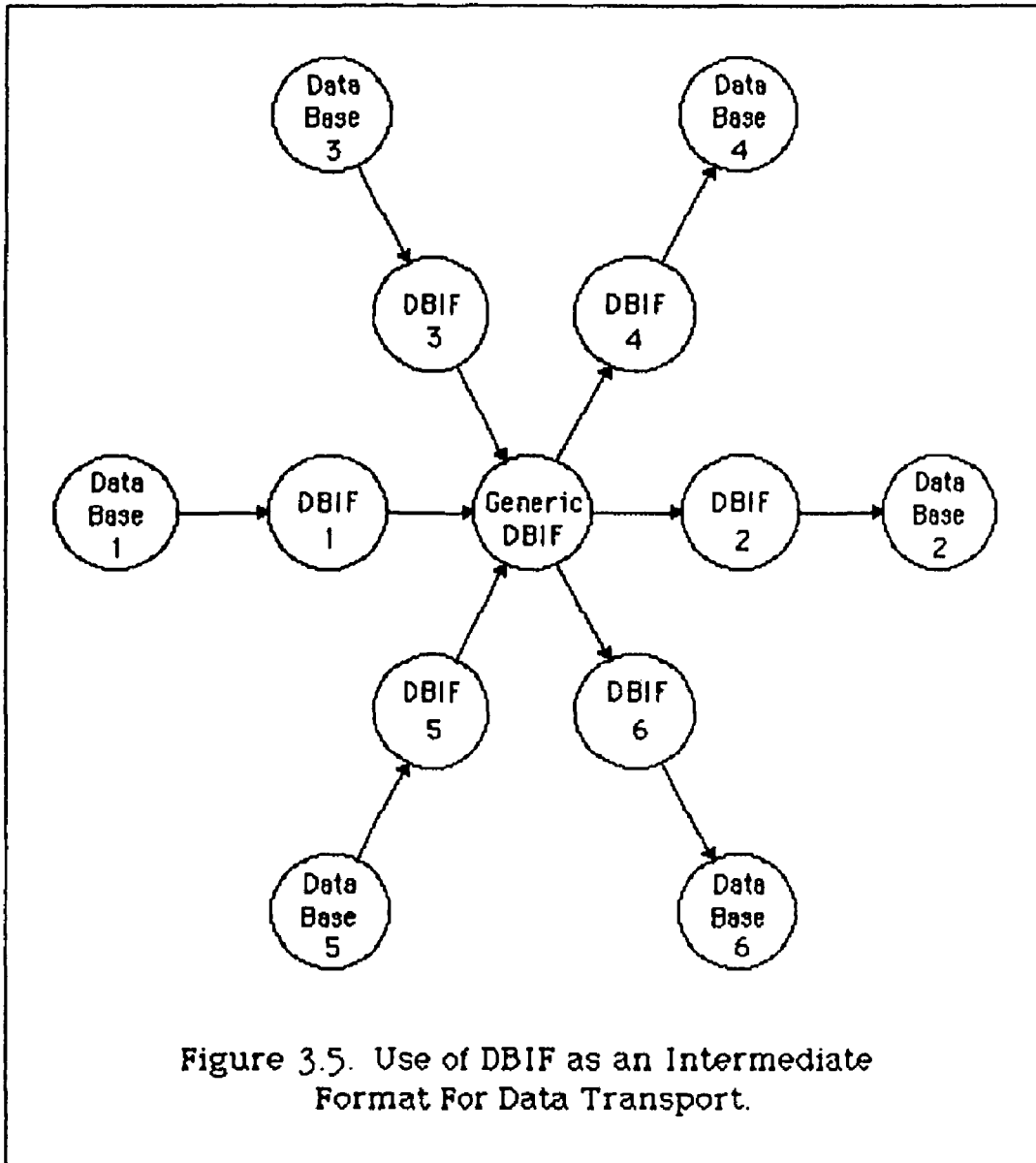


Figure 3.5. Use of DBIF as an Intermediate Format For Data Transport.

data base type 3 can be transformed into data base type 4, and type 5 to 6.

Of course, a relational data base is only one possible DBIF candidate. There are a number of other potential DBIF's: for example, a neutral language or other common data base format. Using a knowledge-based approach, the DBIF would consist of facts which express the data base content. The syntax of the knowledge base depends upon the particular translation mechanism and its preferred input/output format.

*Compilers/formatters.* Compilers are necessary to translate CAD/CAM data bases from their native form into DBIF. Likewise, formatters take data in DBIF and repackage it into a native CAD/CAM data base format. For each CAD/CAM database type, only one compiler and one formatter needs to be generated. ^(3.1)

Working in conjunction with compilers and formatters, a CAE/CAD/CAM data description language can be used to provide language-specific information. This can be used to identify where in a CAE/CAD/CAM database a particular data element is found. A general compiler might be written which maps data between a native format (after media conversion) and DBIF using the CAD/CAM data language description of the native format. Similarly, a formatter can identify where a DBIF item belongs in a native format using the CAE/CAD/CAM data description language.

---

^(3.1) This is very similar to the idea behind IGES.

A knowledge-based approach to compilation has been tried with great success in the field of natural language understanding. Webber [Webb83] has shown how logic and deduction are applied to this field. Similarly for CAE/CAD/CAM data languages, syntax and semantic rules can be expressed in terms of logic predicates. These rules are then used by the compiler. Further examples of the techniques are given by Dahl [Dahl83] who has shown how a knowledge base and logic programming can be used for language processing (both French and Spanish).

Furthermore, a KBS approach allows the validity of a data base to be checked by comparing it against a model (schema) expressed in terms of logic predicates and their relationships. To see how a CAD data base can be modeled in terms of logic predicates, we shall consider below the schematic data base represented by the network in Figure 3.6

#### *CAD Data Base Modeling and Representation*

This simple network consists of four components A, B, C, and C1 of types T1, T2, T2, and CONNECTOR. Seven signals S1-S7 connect the components. Component type T1 has nodes R, S, Q, and QN. Component type T2 has pins (or nodes) I1, I2, and Y. The connector has pins I1-I3, O1, and O2. Two possible native representations of the network are 1) a relational data base and 2) a hardware description language (HDL) encoding. (see Figures 3.7 and 3.8 respectively). Call these two data representations DR1 and DR2.

Using the Entity-Link-Key-Attribute (ELKA) [Rodr81] information

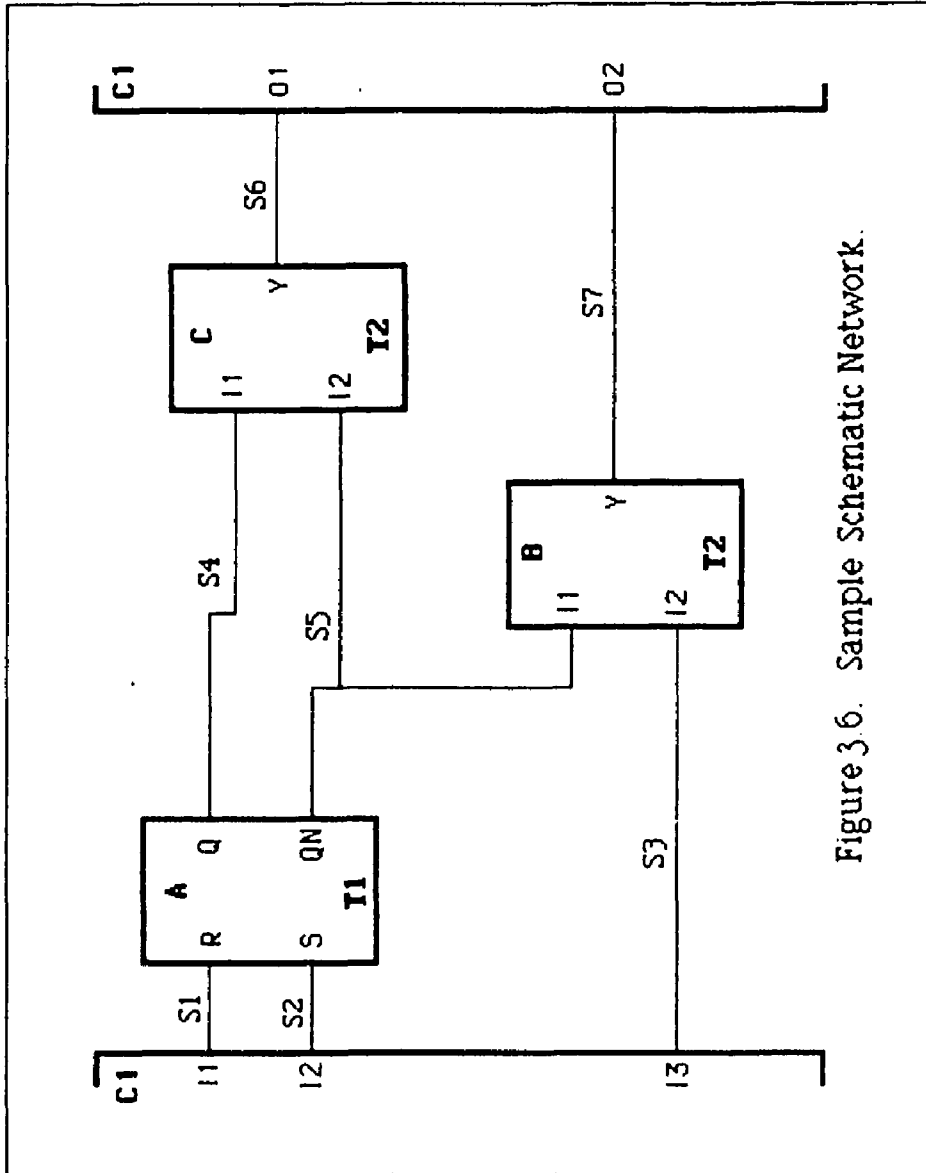


Figure 3.6. Sample Schematic Network.

SIGNAL	COMPONENT NAME	COMPONENT TYPE	PIN
S1	A	T1	R
S1	C1	CONNECTOR	I1
S2	A	T1	S
S2	C1	CONNECTOR	I2
S3	C1	CONNECTOR	I3
S3	B	T2	I2
S4	A	T1	Q
S4	C	T2	I1
S5	A	T1	QN
S5	B	T2	I1
S5	C	T2	I2
S6	C	T2	Y
S6	C1	CONNECTOR	O1
S7	B	T2	Y
S7	C1	CONNECTOR	O2

Figure 3.7. Data Base Representation 1 (DR1).

```
BODIES
  BODY A
    TYPE T1
    NODE R (1,9)
    NODE S (1,8)
    NODE Q (2,9)
    NODE QN (2,8)
  BODY B
    TYPE T2
    NODE I1 (3,4)
    NODE I2 (3,2)
    NODE Y (4,3)
  BODY C
    TYPE T2
    NODE I1 (5,8)
    NODE I2 (5,6)
    NODE Y (6,7)
SIGNALS
  SIGNAL S1 (0,9), (1,9)
  SIGNAL S2 (0,8), (1,8)
  SIGNAL S3 (1,2), (3,2)
  SIGNAL S4 (2,9), (5,8)
  SIGNAL S5 (2,8), (2,5,6), (5,6), (3,4)
  SIGNAL S6 (6,7), (7,7)
  SIGNAL S7 (4,3), (7,3)
```

Figure 3.8. Data Base Representation  
2 (DR2).



model ^(3.2), the semantics of this data (i.e., the schema) can be represented by the diagram shown in Figure 3.9. The entity classes defined are: SIGNAL, COMPONENT, PIN-INSTANCE, PIN, and TYPE. The attribute classes are "signalName", "compName", "compType", and "pinName". Note that some of the attribute classes are underlined. These correspond to the key classes for the entity classes in which they are contained. The links in this diagram are of two types, strong many-to-one (m-to-1) and many-to-one (m-to-1) links. These links are depicted graphically by a line joining two entity classes with a diamond on one end. The front of the link is the end without the diamond and the back is the end with the diamond.

The strong m-to-1 links (those with a solid diamond ) indicate that for every member of the back entity class there exists exactly one member of the front entity class. Also, for every member of the front entity class, there exists one or more members of the back entity class. The m-to-1 link (with the un-filled diamond) is like the strong m-to-1 link except that in an m-to-1 link, for every member of the front entity class there maybe *zero*, one, or more members of the back entity class.

For each representation, the essential data elements and their relationships can be expressed in terms of logic predicates or statements. As Figures 3.7 and 3.9 indicate, for DR1, the following statements hold:

---

^(3.2) The ELKA model was used extensively at Hughes for the US Department of Defense VHSIC program. Specifically, all of the data handled by the Hughes-developed VHSIC CAD system, called HERCULES, was modeled using the ELKA technique.

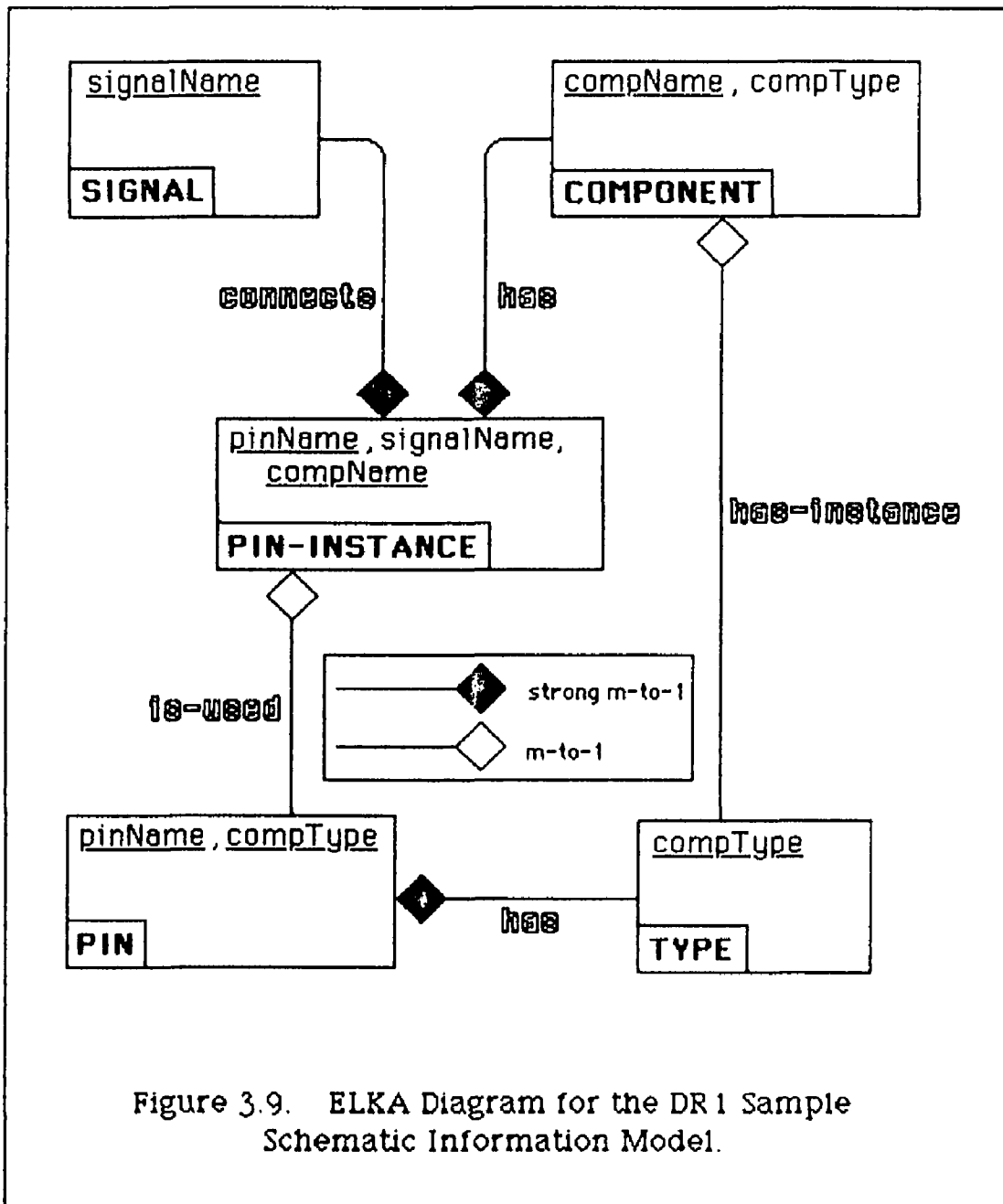


Figure 3.9. ELKA Diagram for the DR 1 Sample Schematic Information Model.

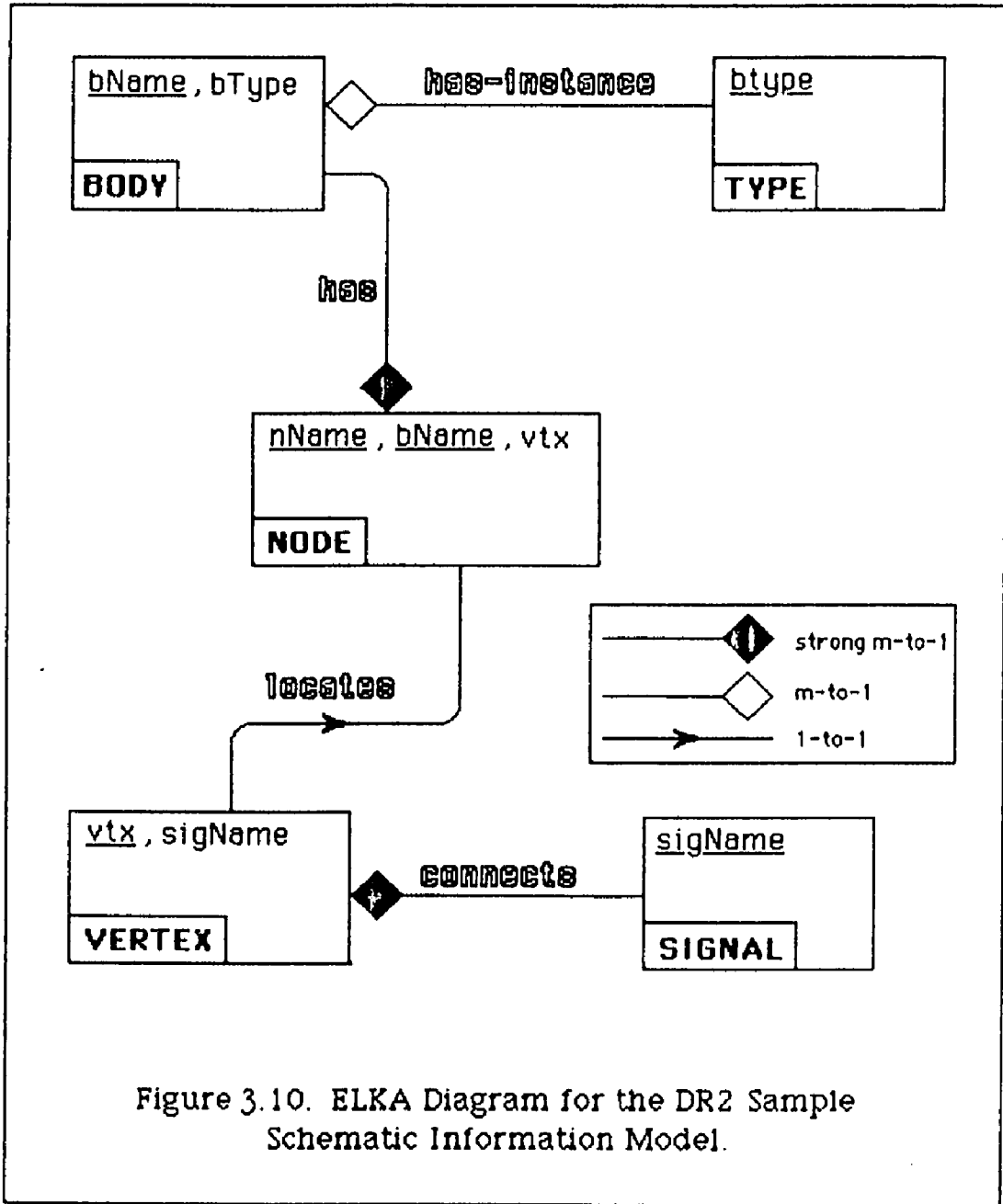
1. If **X** is a signal, then **X** connects one or more pin instances.
2. If **X** is a pin instance, then **X** is connected by one and only one signal.
3. If **X** is a component name, then **X** has one or more pin instances.
4. If **X** is a pin instance, then there is one and only one component name which has **X**.
5. If **X** is a component type, then **X** has one or more pins.
6. If **X** is a pin, then there is one and only one component type which has **X**.
7. If **X** is a component type, then **X** has zero or more instances of component name.
8. If **X** is a component name, then there is one and only one component type which has the instance **X**.
9. If **X** is a pin, then there are zero or more pin instances where **X** is used. If **X** is a pin instance, then there is one and only one pin which is used as **X**.

This information model for schematic data has been simplified to illustrate the point that a CAD data base can be characterized in terms of logic predicates. This will prove useful in Chapter 4 when a more realistic schematic data model is presented.

In a like manner, the data elements for DR2 and their relationships can be modeled as shown in Figure 3.10. Information Model." In this case, the entity classes are: BODY, TYPE, NODE, VERTEX, and SIGNAL. The attribute classes are "bName", "bType", "nName", "vtx", and "sigName". As in the previous ELKA diagram, the underlined attribute classes are also key classes. The links in this diagram are strong m-to-1, m-to-1, or 1-to-1 links. A 1-to-1 link indicates that for every member of the back entity class, there exists exactly one member of the front entity class. And, for every member of the front entity class there exists zero or one member of the back entity class.

As Figures 3.8 and 3.10 indicate, for DR2, the following statements hold:

1. If **X** is a body, then **X** has one or more nodes.
2. If **X** is a node, then there is exactly one body which has **X**.
3. If **X** is a body type, then **X** has zero or more body instances.
4. If **X** is a body, then there is exactly one body type which has instance **X**.
5. If **X** is a node, then there is exactly one vertex which locates **X**.
6. If **X** is a vertex, then there is zero or one node which is located by **X**.
7. If **X** is a signal, then there are one or more vertices connected



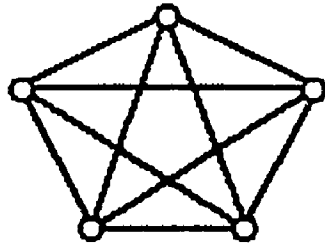
by **X**.

8. If **X** is a vertex, then there is exactly one signal which connects to **X**.

Although these logic assertions contain the necessary information, it is clear that they must be expressed in a more concise and regular format in order to be useful.

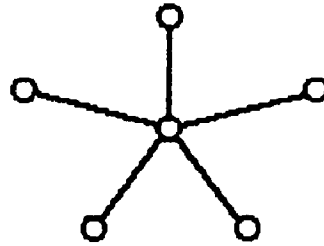
*Generic/neutral data schema.* Assuming that all data can be compiled into a DBIF representation, a master data schema is needed to represent a superset of all of the CAD/CAM data bases to be transferred. The schema needs to include each category of CAD/CAM data: logical, physical, electrical, thermal, and timing. Data from each native system is first compiled into its DBIF and then it is translated according to the master data schema into a generic DBIF. The rationale for the generic schema is so that for each native format only two translators are needed: one from the native format into the generic schema and one back from the generic schema into the native format. Without a generic schema, point-to-point translators would be needed between pair of native formats. For a collection of  $N$  native systems,  $2\Sigma^{(N-1)}$  translators need to be written (see Figure 3.11a). Using the generic schema, only  $2N$  translators need to be written. This corresponds to a "star" configuration (see Figure 3.11b).

For a five system collection this is a difference between 20 ( $=2*\Sigma_{i=1}^4$ ) translators and 10 ( $=2*5$ ) translators. An additional advantage is gained when a native format changes. Without a generic schema,  $N-1$  translators are affected. With a generic schema, only 2 translators need to be changed.



$$2\sum_1^{N-1}$$

Figure 3.11a.  
Point-to-Point  
Configuration.



$$2N$$

Figure 3.11b. Star  
Configuration.

Alternative Translation Schemes.

Number of Systems	Generic Data Schema	
	With	Without
3	6	6
4	8	12
5	10	20
6	12	30
7	14	42

Figure 3.12. Translators Needed Using a  
Generic Data Schema.

Figure 3.12 shows how this difference varies with the number of systems sharing data.

Assuming a generic data schema is used, it can be represented as a set of rules in a knowledge base just as native CAE/CAD/CAM data bases are described.

*Translation Mechanism.* While several methods can be used to translate CAE/CAD/CAM data content, a knowledge-based approach offers more flexibility and generality. Returning to the sample schematic network and DR1 and DR2, a translation scenario shows how rules can be used to transform data between different representations. This translation was developed manually to understand the algorithmic steps. In Chapter 4, a similar automatic translation is introduced. Assume that the sample schematic network of Figure 3.6 is to be translated from DR2 into DR1. The steps in this process are

1. Compile the native DR2 into DBIF.
2. Prepare DR2 to generic translation rules (input rules).
3. Prepare generic to DR1 translation rules (output rules).
4. Translate DR2 into generic data.
5. Translate Generic into DR1 data.
6. Format the resulting DR1 DBIF into DR1 native form.

These steps are described in more detail in the next section (3.5) and they are illustrated in Figure 3.15. To give a more thorough understanding of



these steps, the following DR2 to DR1 transport scenario is provided.

Figure 3.13 shows the DR2 data encoded in a DBIF, consisting of factual statements. (Note that each input fact is numbered  $I_x$  for future reference). The "TYPE" information (Figure 3.8) has been omitted in this example to simplify the explanation.

In order to be able to translate this data into an alternate representation, it is important that it be defined in terms of the generic data schema. The following is a set of assertions which translate DR2 into the generic data schema along with the previous set of rules which operate with DR1.

- R1. X is a DR2 body if X is a generic box.
- R2. X is a generic box if X is a DR2 body.
- R3. X is a DR2 node if X is a generic node.
- R4. X is a generic node if is a DR2 node.
- R5. X is a DR2 signal if X is a generic net.
- R6. X is a generic net if X is a DR2 signal.
- R7. X is a DR1 signal if X is a generic net.
- R8. X is a generic net if X is a DR1 signal.
- R9. X is a DR1 pin if X is a generic node.
- R10. X is a generic node if X is a DR1 pin.
- R11. X is a DR1 compname if X is a generic box.
- R12. X is a generic box if X is a DR1 compname.
- R13. X connects Y if X has vertex (A,B), Y has vertex (A,B),  
and X is a DR2 signal.

- |                             |                             |
|-----------------------------|-----------------------------|
| I1. A is a dr2 body.        | I29. C is a dr2 node.       |
| I2. A has R.                | I30. C.I2 has vertex (5,6). |
| I3. R is a dr2 node.        | I31. B has C.Y              |
| I4. R has vertex (1,9).     | I32. C is a dr2 node.       |
| I5. A has S.                | I33. C.Y has vertex (6,7).  |
| I6. S is a dr2 node.        | I34. S1 is a dr2 signal.    |
| I7. S has vertex (1,8).     | I35. S1 has vertex (0,9).   |
| I8. A has Q.                | I36. S1 has vertex (1,9).   |
| I9. Q is a dr2 node.        | I37. S2 is a dr2 signal.    |
| I10. Q has vertex (2,9).    | I38. S3 is a dr2 signal.    |
| I11. A has QN.              | I39. S4 is a dr2 signal.    |
| I12. QN is a dr2 node.      | I40. S5 is a dr2 signal.    |
| I13. QN has vertex(2,8).    | I41. S6 is a dr2 signal.    |
| I14. B is a dr2 body.       | I42. S7 is a dr2 signal.    |
| I15. B has B.I1             | I43. S2 has vertex (0,8).   |
| I16. B is a dr2 node.       | I44. S2 has vertex (1,8).   |
| I17. B.I1 has vertex (3,4). | I45. S3 has vertex (1,2).   |
| I18. B has B.I2             | I46. S3 has vertex (3,2).   |
| I19. B is a dr2 node.       | I47. S4 has vertex (2,9).   |
| I20. B.I2 has vertex (3,2). | I48. S4 has vertex (5,8).   |
| I21. B has B.Y              | I49. S5 has vertex (2,8).   |
| I22. B is a dr2 node.       | I50. S5 has vertex (2,5,6). |
| I23. B.Y has vertex (4,3).  | I51. S5 has vertex (5,6).   |
| I24. C is a dr2 body.       | I52. S5 has vertex (3,4).   |
| I25. B has C.I1             | I53. S6 has vertex (6,7).   |
| I26. C is a dr2 node.       | I54. S6 has vertex (7,7).   |
| I27. C.I1 has vertex (5,8). | I55. S7 has vertex (4,3).   |
| I28. B has C.I2             | I56. S7 has vertex (6,3).   |

Figure 3.13. DBIF Encoding of DR2 Data.

Now translation can occur using the input data encoded in DBIF and the translation rules. The following notation represents an output transaction "Ox" as a result of rules being applied to input record x, denoted "Ix". The arrow indicates *results in*. "Rx" means rule x. So,

I5 → R3: fido is a dog. → R10: fido is an  
animal.

says that by applying rule 3 to Input record 5, results in "fido is a dog."  
And, that by subsequently applying rule 10, "fido is a animal." is obtained.

The following example shows the translation process. The rules re-  
garding DR1 and DR2 are applied until the facts of Figure 3.13 result in a  
new set of facts (below) which have DR1 specified, where possible, instead  
of DR2.

O1: I1 → R2: A is a generic box.  
→ R11: A is a DR1 component name.

O2: I2 → no rule (NR): A has R

O3: I3 → R4: R is a generic node.  
→ R9: R is a DR1 pin.

O4: I4 → R13(I34,I36): S1 connects R.

O5: I5 → NR: A has S.

O6: I6 → R4: S is a generic node. → R9: S is a DR1 pin.

O7: I7 → R13(I37,I44): S2 connects S

O8: I8 → NR: A has Q.

O9: I9 → R4: Q is a generic node. → R9: Q is a DR1 pin.

O10: I10 → R13(I39,I47): S4 connects Q.

O11: I11 → NR: A has QN

O12: I12 → R4: QN is a generic node.  
→ R9: QN is a dr1.pin.

O13: I13 → R13(I40,I49): S5 connects QN.

O14: I14 → R9: B is a box.

→ R18: B is a DR1 compname.  
 O15: I15 → no rule (NR): B has B.I1  
 O16: I16 → R4: B.I1 is a generic node.  
       → R9: B.I1 is a DR1 pin.  
 O17: I17 → R13(I40,I52): S5 connects B.I1.  
 O18: I18 → NR: B has B.I2  
 O19: I19 → R4: B.I2 is a generic node.  
       → R9: B.I2 is a DR1 pin.  
 O20: I20 → R13(I38,I46): S3 connects B.I2.  
 O21: I21 → NR: B has B.Y  
 O22: I22 → R4: B.Y is a generic node.  
       → R9: B.Y is a DR1 pin.  
 O23: I23 → R13(I42,I55): S7 connects B.Y.  
 O24: I24 → R9: C is a box.  
       → R18: C is a DR1 compname.  
 O25: I25 → NR: C has C.I1  
 O26: I26 → R4: C.I1 is a generic node.  
       → R9: C.I1 is a DR1 pin.  
 O27: I27 → R13(I39,I48): S4 connects C.I1.  
 O28: I28 → NR: C has C.I2  
 O29: I29 → R4: C.I2 is a generic node.  
       → R9: C.I2 is a DR1 pin.  
 O30: I30 → R13(I40,I51): S5 connects C.I2.  
 O31: I31 → NR: C has C.Y  
 O32: I32 → R4: C.Y is a generic node.  
       → R9: C.Y is a DR1 pin.

O33: I33  $\rightarrow$  R13(I41,I55): S6 connects C.Y.

O34: I34  $\rightarrow$  R6: S1 is a generic net.  
 $\rightarrow$  R7: S1 is a DR1 signal.

O35: I35  $\rightarrow$  NR: S1 has vertex (0,9).

O36: see O4

O37: I37  $\rightarrow$  R6: S2 is a generic net.  
 $\rightarrow$  R7: S2 is a DR1 signal.

O38: I38  $\rightarrow$  R6: S3 is a generic net.  
 $\rightarrow$  R7: S3 is a DR1 signal.

O39: I39  $\rightarrow$  R6: S4 is a generic net.  
 $\rightarrow$  R7: S4 is a DR1 signal.

O40: I40  $\rightarrow$  R6: S5 is a generic net.  
 $\rightarrow$  R7: S5 is a DR1 signal.

O41: I41  $\rightarrow$  R6: S6 is a generic net.  
 $\rightarrow$  R7: S6 is a DR1 signal.

O42: I42  $\rightarrow$  R6: S7 is a generic net.  
 $\rightarrow$  R7: S7 is a DR1 signal.

O43: I43  $\rightarrow$  NR: S2 has vertex (0,8).

O44: see O7

O45: I43  $\rightarrow$  NR: S3 has vertex (1,2).

O46: see O20

O47: see O10

O48: see O27

O49: see O13

O50: I50  $\rightarrow$  NR: S5 has vertex (2.5,6).

O51: see O30

O52: see O17

O53: see O33

O54: I54  $\rightarrow$  NR: S6 has vertex (7,7).

O55: see O23

O56: I56  $\rightarrow$  NR: S7 has vertex (6,3).

A final DR1 representation is obtained from the resulting output facts. Figure 3.14 shows the relational data base. A few items of note are that not all facts expressed in DR2 can be translated into DR1 facts. Specifically O35, O43, O45, O50, O54, and O56 are not expressible in DR1. This is a case of the *delta problem* discussed previously. These facts which do not translate must be saved as part of the resulting DBIF. When a reverse translation is required, these untranslated facts must be used to derive the original data representation. A production-quality system needs to provide this capability.

This example shows the basic technique of using a knowledge base to perform translation. The six steps described at the beginning of this example have been executed manually and the desired result obtained.

### **3.5 System Architecture**

The elements needed for data transport can be integrated into a system architecture as shown in Figure 3.15. The elements are made up of both processes and data.

Record #	Signal	Component Name	Pin	Derived From
1	S1	A	R	02, 04, 01, 03, 034
2	S2	A	S	05, 01, 06, 07, 037
3	S4	A	Q	08, 01, 09, 010, 039
4	S5	A	QN	011, 01, 012, 013, 040
5	S5	B	B.11	014, 015, 016, 017, 040
6	S3	B	B.12	018, 014, 019, 020, 038
7	S7	B	B.Y	021, 014, 022, 023, 039
8	S4	C	C.11	024, 025, 026, 027, 039
9	S5	C	C.12	028, 024, 029, 030, 040
10	S6	C	C.Y	031, 024, 032, 033, 041
unused: 035, 043, 045, 050, 054, 056				
Figure 3.14. DR1 Relational Data Base.				

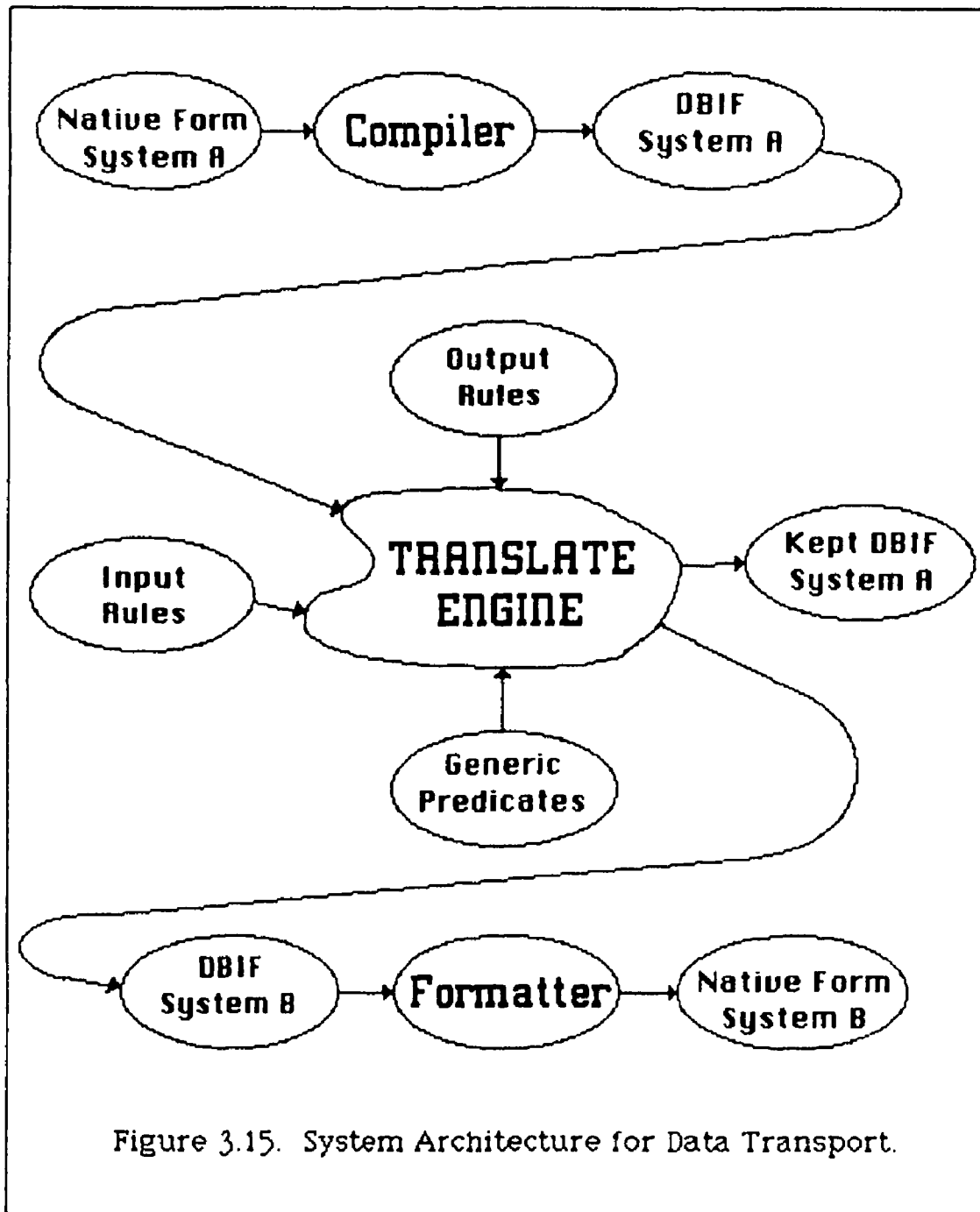


Figure 3.15. System Architecture for Data Transport.



### **3.5.1 Compiler / DBIF**

The first step in transporting data between systems is the process of compiling data from its native (source system) format. This process varies with each native system. When the source data is in the form of a language, then a compiler which parses the language and outputs DBIF is appropriate. A compiler needs to be constructed for each unique type of native language. Depending on the complexity of the language and the number of data elements to be recognized by the system, this can be a time-consuming process.

When the native form of the source data is itself a data base, then the appropriate data base query language statements can extract the data and format it into DBIF. This is usually easier than parsing a language.

As described earlier, the DBIF format consists of logical predicates which assert facts from the data base. The problem with the English-like logic statements is that they are not easily processed by a translation engine. A more regular syntax for the logic predicates alleviates this difficulty. Consider the schematic example of Figure 3.6. The DR2 representation of the schematic data can be encoded into DBIF as shown in Figure 3.16. This exact syntax for DBIF is not important as far as the system architecture is concerned. The actual implementation of the architecture into a working system will dictate the exact syntax required for DBIF.

### SCHEMATIC DATA BASE:

1. compname(A).
2. comptype(T1).
3. compname(B).
4. comptype(T2).
5. compname(C).
6. comptype(T2).
7. compname(C1).
8. comptype(CONN).
9. has(A,T1).
10. has(B,T2).
11. has(C,T2).
12. has(C1,CONN).
13. node(R).
14. node(S).
15. node(Q).
16. node(QN).
17. node(I1).
18. node(I2).
19. node(Y).
20. node(I3).
21. node(O1).
22. node(O2).
23. has(T1,R).
24. has(T1,S).
25. has(T1,Q).
26. has(T1,QN).
27. has(T2,I1).
28. has(T2,I2).
29. has(T2,Y).
30. has(CONN,I1).
31. has(CONN,I2).
32. has(CONN,I3).
33. has(CONN,O1).
34. has(CONN,O2).
35. signal(S1).
36. signal(S2).
37. signal(S3).
38. signal(S4).
39. signal(S5).
40. signal(S6).
41. signal(S7).
42. connects(S1,C1,I1).
43. connects(S1,A,R).
44. connects(S2,C1,I2).
45. connects(S2,A,S).
46. connects(S3,C1,I3).
47. connects(S3,B,I2).
48. connects(S4,A,Q).
49. connects(S4,C,I1).
50. connects(S5,A,QN).
51. connects(S5,C,I2).
52. connects(S5,B,I1).
53. connects(S6,C1,O1).
54. connects(S6,C,Y).
55. connects(S7,C1,O2).
56. connects(S7,B,Y).

Figure 3.16. Sample DBIF Encoding.

### 3.5.2 Master Data Schema: Generic Predicates

Assuming that all native formats can be translated into the same DBIF, they will all have similar format, but unique content (i.e., the facts or predicates will be different), as described in section 3.4. It was shown to be an advantage, using a generic data schema for translating between N

unique system types.

There must be a set of generic predicates for each category of data. Each asserts a fact about a data element or facts about data element relationships. Figures 3.17 and 3.18 show sets of generic predicates for two categories of data: logical and physical.

One problem that has been encountered by most attempts at a neutral data standard is that not all data elements are included in the standard. The goal of a standard is to have a superset of all data elements and data relationships that are encountered in a given category. This is somewhat impractical, since it is difficult to define the universe of all native systems having a given CAE/CAD/CAM data category. Furthermore, if a given collection of systems is augmented with a new system at a later time, then the neutral standard may need to be augmented as well. This is somewhat self-defeating, since the neutral data format is supposed to influence the data base definitions for CAE/CAD/CAM systems and not the other way around.

Rather than attempt to provide a universal, all-encompassing standard, a reasonable set of generic predicates for each CAE/CAD/CAM data category is sufficient. If experience indicates that additions should be made, then the set of generic predicates can be expanded accordingly. But, the *delta problem* described in section 3.2 must now be addressed since not all DBIF constructs from all native systems will be translatable into the generic DBIF. This is one of the roles of the translate engine

**LOGICAL:**

<u>predicate</u>	<u>meaning</u>
box(X)	X is a component or cell name
box_type(X)	X is a component or cell type
node(X)	X is a node
node_type(X)	X is a node type
node_dir(X)	X is a node direction
net(X)	X is a signal
net_type(X)	X is a signal type
has_(X,Y)	X has Y or Y is contained in X
connects(X,Y,Z)	Signal X is connected to Component Y at Node Z

Figure 3.17. Generic Predicates for Logical Data.

### 3.5.3 Translate Engine

Assuming that the various data files shown as inputs into the translate engine in Figure 3.15 are available, the next step is to translate one DBIF into another DBIF of similar content. The first step involves translating the predicates of the source system's DBIF into generic predicates for the type of data to be translated. This requires the source DBIF and the input rules for the source system. These rules are discussed in the next section in more detail. An intermediate output of the translation engine is the data content of the source system's DBIF expressed in terms of

PHYSICAL:

<u>predicate</u>	<u>meaning</u>
polygon(X)	X is a polygon
wire(X)	X is a wire
macro_def(X)	X is a structure or macro
macro_call(X,N)	X is a reference or call to macro definition N
scale_(X)	X is a scaling factor
layer_(X,Y)	X has layer Y
vertex_(S,X,Y,I)	S has vertex (X,Y) as its Ith coordinate
width_(X,Y)	X has width Y
orient_(W,X,Y,Z)	W is rotated X degrees about the x axis, Y degrees about the Y axis, and Z degrees about the Z axis.
has_(X,Y)	X has Y or Y is contained in X
magnif_(S,M)	Magnification M is applied to the elements of S
relative_orient(S)	Any orientation specified for S is relative to the calling frame of reference (otherwise is with respect to absolute zero).
relative_magnif(S)	Any magnification specified for S is relative to the calling macro's magnification (otherwise it is absolute).
text_(T)	T is a text block
textval_(T,Str)	Str is the character string for text block T
h_just(T,N)	N is the horizontal justification for text block T: left, right, or center.
v_just(T,N)	N is the vertical justification for text block T: upper, lower, or center.
tfont_(T,F)	F is the font type for text block T

Figure 3.18. Generic Predicates for Physical Data.

generic predicates. A by-product of the translation is a set of facts which cannot be translated into generic predicates. These are called *kept data* and they must be preserved so that reverse translation at a later time is possible.

Once the generic facts are available, they are translated into the destination DBIF using a new set of rules which indicate how to derive the destination system's predicates from generic predicates. Again, there may be facts which cannot be translated from the generic set into the DBIF of the destination system. These must also be preserved as *kept data* so that reverse translation is possible ^(3.4). The final output of the translate engine is the destination system's DBIF and the kept data. The output DBIF is reformatted for direct use by the destination system.

#### **3.5.4 Rules for Translation: Knowledge Base**

In order to perform any translations, a set of rules is necessary which specify how to interpret predicates from one system to another, including the set of generic predicates. These rules constitute a knowledge base. The knowledge base containing the rules for translation consists of several parts:

- rules, describing how to express generic predicates in terms of source predicates,

---

^(3.4) It is necessary that the existence of kept data be remembered when the time comes for reverse translation. It would be possible to insert a flag to this effect into the body of the output DBIF.

- rules, describing how to express target predicates in terms of generic predicates, and
- ordered lists of source predicates, target predicates, and generic predicates.

The knowledge base must contain rules pertaining to each CAE/CAD/CAM system to be recognized by the data transport system.

The rules are arbitrary in format, and they must be tailored to fit the CAE/CAD/CAM system under consideration. The rules used in section 3.4 to translate between DR1, DR2, and a generic set of predicates are an example. Additional examples of rules are presented in Chapter 5.

### **3.5.5 Formatter**

Once the knowledge base has been used to obtain the DBIF of the destination system, the formatter is used to write the content of the DBIF in the format that the native CAE/CAD/CAM system requires. This might be a data base or language. Using the same DBIF encoding of Figure 3.16, a relational data format can be derived to produce three tables "connects", "has", and "type" which are shown in Figure 3.19.

CONNECTS:		
Signal	CompName	No
S1	C1	I1
S1	A	R
S2	C1	I2
S2	A	S
S3	C1	I3
S3	B	I2
S4	A	Q
S4	C	I1
S5	A	QN
S5	C	I2
S5	B	I1
S6	C1	O1
S6	C	Y
S7	C1	O2
S7	B	Y

HAS:	
CompType	Node
T1	R
T1	S
T1	Q
T1	QN
T2	I1
T2	I2
T2	Y
CONN	I1
CONN	I2
CONN	I3
CONN	O1
CONN	O2

TYPE:	
CompName	CompType
A	T1
B	T2
C	T2
C1	CONN

Figure 3.19. Sample Formatter Results.



## CHAPTER 4

### PROTOTYPE SYSTEM

The system architecture for CAE/CAD/CAM data transport, presented in the previous chapter was implemented using Prolog. Alternative approaches were considered, but discarded because of the difficulties that arise in trying to represent a knowledge base. Also, an advantage to using Prolog is the built-in inference engine which provides for automatic translation, once the appropriate rules (knowledge base) are defined. Appendix B describes in detail one alternative implementation considered. The remainder of this chapter describes Prolog, the elements of the prototype implementation, and then a sample operating scenario.

#### **4.1 Introduction to Prolog**

This section presents the Prolog language. Readers, familiar with Prolog, can skip this digression and proceed on to section 4.2.

Prolog is a programming language which dates back to around 1970. Prolog has gained recent attention as a tool for artificial intelligence, particularly for Expert and Knowledge-based systems. Prolog programs consist of facts and rules. Facts express information about objects and their relationship to other objects. Rules contain the intelligence necessary to derive new facts. One good source for detailed information on Prolog is Clocksin & Mellish [Cloc81]. A brief description is provided here.

*Syntax:* Prolog programs are built from terms, which are either constants, variables, or structures. *Constants* are atoms or numbers. Atoms are any string of characters, enclosed in single quotes, or special Prolog symbols, or alpha-numeric strings beginning with a lower-case alpha character. Numbers can either be integers or real numbers. ^(4.1) *Variables* are alpha-numeric strings beginning with an upper-case alpha character or the underscore, "_". The underscore can also be inserted in the middle of atoms or variables to improve readability. Figure 4.1 shows examples of Prolog terms.

Constants		Variables
Atoms	Numbers	
mary '&/5*abc' 'Capital'	123 1.05 3.14e-5	G1 Dogs _x
Figure 4.1. Prolog Terms.		

*Structures* consist of a functor (or predicate) which is an atom, followed by one or more terms called arguments. A simple example of a structure is

owns(mary,Dog).

In this case, the functor is "owns", the first argument can be the constant atom, "mary", and the second argument is the variable, "Dog". Structures can be nested, since an argument is any term. Thus, an entry in a library card catalog might be

---

^(4.1) The Prolog definition presented in [Cloc81] doesn't include real numbers. However, the VAX CProlog does.

```
book(author('J. Doe'),
      publisher(name('McGraw-Hill'),
                loc('New York')),
      date(1983),
      title('Abstract Data Structures')).
```

It should be noted that the order of the arguments is important. The significance of the atoms and structures is strictly the decision of the programmer. But, consistency in definition is required in order to be able to obtain the expected results.

One special type of structure is that created by the special functor dot or "." . This structures is referred to as a *list*. This is the same as the list data structure in the programming language LISP. In Prolog, the lists containing members a, b, and c is represented as `.(a,.(b,.(c,[])))` , where [] represents the empty list. The same list can also be represented in Prolog using a special list notation: `[a,b,c]`.

Using these basic elements of Prolog syntax, not only can facts be asserted, but also rules can be defined. Rules are logic clauses which express an assumption and the conditions which must be true in order for the assumption to be true. For a more in depth discussion of the relationship of Prolog to predicate calculus and Horn clauses, consult Chapter 10 of Clocksin & Mellish [Cloc81].

To clarify the basic concepts, consider the following logic problem and the usage of Prolog to solve it.

Three students recently graduated from a college and want to attend graduate school. Each student will attend a different university next year, each in a different major field. The strong point in favor of each student being accepted is different. Determine the school, major, and strong point of each student, considering the following clues:

1. The student who applied to Yale will not be studying business.
2. Brown and the student going to Harvard, who is not Jones, had marginal G.P.A.'s.
3. Jones will not be attending UCLA.
4. The students with good references and high G.R.E. scores will not be studying History.
5. Smith, who had mediocre G.R.E.'s, doesn't intend to study Computer Science.

The following set of Prolog statements express the same information, and they can be used to arrive at an answer to the problem.

```

/* Facts */
name(brown). name(jones). name(smith).
sp(gre).      sp(gpa).      sp(refs).
sch(ucla).    sch(yale).    sch(harvard).
mjr(comp_sci). mjr(history). mjr(business).

/* Rules */
strong(B,J,S):-
    sp(B),
    B\==gpa, /* clue 2 */
    sp(J), J\==B,
    sp(S),
    S\==gre, /* clue 5 */
    S\==B, S\==J.

major(B,J,S,SB,SJ,SS):-
    mjr(S),
    S\==comp_sci, /* clue 5 */
    clue_4(S,SS),
    mjr(J), J\==S, clue_4(J,SJ),
    mjr(B), B\==S, B\==J,
    clue_4(B,SB).

clue_4(history,refs):-!,fail.
clue_4(history,gre):-!,fail.
clue_4( _, _).

school(B,J,S,MB,MJ,MS,SB,SJ,SS):-
    sch(B),
    B\==harvard, /* clue 2 */

```

```

    clue_1(B,MB),
    sch(J),J\==B,
    J\==ucla, /* clue 3 */
    J\==harvard, /* clue 2 */
    clue_1(J,MJ),
    clue_2(J,SJ),
    sch(S),S\==B, S\==J,
    clue_1(S,MS),
    clue_2(S,SS).

clue_1(yale,business):-!,fail.
clue_1( _, _).
clue_2(harvard,gpa):-!,fail.
clue_2( _, _).

```

The facts are self-explanatory. The rules can be interpreted as follows. The definition for "strong" indicates that three variables are to be instantiated. There is no significance to the letters chosen (B, J, and S) except that they are mnemonics chosen to stand for the last names of the students. The Prolog atom ":-" can be read "is true if." The definition for "strong" contains a series of structures on the right-hand side of the ":-" symbol,. This series represents a conjunction of clauses, each of which must be true in order for the entire definition to be evaluated as true. The first clause "sp(B)" will cause "B" to be instantiated to the first constant found such that "sp( )" is true. In this case, "sp(gre)" is a fact, and "B" is instantiated to "gre."

In each succeeding clause of the conjunction, all possible definitions are tried until one evaluates true. Any variables instantiated in this process remain with their set value and processing continues with the next clause in the series.

In this example, the next clause is "B\==gpa" which reads "B" is not equal to "gpa." If this check is true, processing continues. But, once a

clause cannot be evaluated true, backtracking begins. Working backward from the point of failure, clauses are re-examined to look for alternative definitions which evaluate true. In this case, "B\===gpa" is true since "B" is currently instantiated to "gre." Note that "/* clue 2 */" is merely a comment.

The next clause is "sp(J)" which is satisfied by "sp(gre)." The next clause, "J\===B", now checks to see if "J" (=gre) is not equal to "B" (=gre). This check evaluates false, and backtracking begins with "sp(J)." This clause also evaluates true by instantiating "J" with "gpa" instead of "gre." Now, "J\===B" is true since "gre" is not equal to "gpa." In this manner, the remainder of the conjunction series is evaluated. Upon invocation of "strong," the first successful evaluation returns "strong(gre,gpa,refs)."

The definition of "major" contains a reference to "clue_4." The first definition of "clue_4" assumes that both arguments have already been instantiated. In this case, if the two arguments match "history" and "refs", then "clue_4" fails. The "!" also called "the cut" prevents backtracking from proceeding backward over the "!" symbol. In this case, the cut prevents alternative definitions to "clue_4" from being evaluated. If the arguments do not match "history" and "refs", then the next definition is evaluated. The last definition for "clue_4" contains two instances of the anonymous variable "_". This variable is used when no further remembering of the variable is necessary. Each occurrence of "_" is a unique variable. The definitions for "clue_4" indicate that it is false if "history" is paired with "refs" or "gre" in the definition of "major." Any other pairing

is true; i.e., "clue_4( _,_)." is a true statement without any condition being imposed by the ":" symbol.

In order to solve the logic problem, a question needs to be posed once the facts and rules have been entered. Prolog prompts with "| ?-" and the question is posed:

```
| ?- strong(SB,SJ,SS),major(MB,MJ,MS,SB,SJ,SS),
      school(B,J,S,MB,MJ,MS,SB,SJ,SS).
```

In this question, the variables SB, SJ, and SS correspond to the strong points of Brown, Jones, and Smith, respectively; MB, MJ, and MS, their majors; and B, J, and S their schools. The answer Prolog returns is

```
CProlog version 1.4d.edai
| ?- ['logprob.pro'].
logprob.pro consulted 2120 bytes 0.8 sec.
```

```
yes
| ?- strong(SB,SJ,SS),
      major(MB,MJ,MS,SB,SJ,SS),
      school(B,J,S,MB,MJ,MS,SB,SJ,SS).
```

```
S = harvard
J = yale
B = ucla
MS = business
MJ = history
MB = comp_sci
SS = refs
SJ = gpa
SB = gre ;
```

```
no
| ?- halt.
```

```
[ Prolog execution halted ]
```

Prolog contains quite a few more built-in predicates and operators. As necessary to explain the knowledge base and translation process, these additional Prolog features will be described. This simple example merely illustrates how Prolog works.

## 4.2 Prototype Elements

With a proper understanding of Prolog, the knowledge-based approach to the prototype system and its elements can be described.

As presented in section 3.5, the system architecture for CAE/CAD/CAM data transport consists of three process modules: compiler, translate engine, and formatter. Several data bases feed these processes: source CAE/CAD/CAM data, generic predicates, source to generic rules, and generic to target rules. The resulting outputs are target CAE/CAD/CAM data and *kept data* in DBIF. Each of these elements will be described.

### 4.2.1 Process Modules

*Compiler:* This module may be implemented a number of ways. Conventional compiler techniques are appropriate for language translation or alternatively Prolog can be used. For a CAE/CAD/CAM data base, a data base dump routine can be written to produce a DBIF rendition of the data. For each native data source, a separate compiler is required.

To insure the feasibility of this methodology, two compilers were produced as part of this research. The first compiler was written to translate an industry standard, CALMA GDS II, data base into its appropriate DBIF. This work was performed by Marilyn Caro as part of a Master's Comprehensive Examination at UCLA [Caro83]. The binary input was read into Ms. Caro's compiler, written in FORTRAN, and the output was an early version of the data base intermediate format as described in section 3.5 and described in further detail in this section. The compiler



was written in FORTRAN on a VAX 11/780 running VMS. Approximately 600 lines of code were needed to implement the compiler [Caro83].

The input to the compiler was a standard CALMA GDS II Stream Format tape, which contains binary data. Appendix C shows a hex dump of a CALMA Stream Format file which was used for this purpose. The meaning of the Stream Format tape is described in Figure 4.2 which shows a pseudo-BNF description of its syntax. Further description of this data is contained in Chapter 5. The terminals in this description are actually binary records of variable length. Each binary record starts with 2 bytes containing a count of the total record length in bytes (8-bits). The third byte is the record type (e.g., BGNLIB = type 1). The fourth byte is the type of data contained within the record (e.g., 2-byte integer, 4-byte real, bit array, etc.). Starting with the fifth byte, begins any data associated with the record type.

Figure 4.3 shows a sample CALMA Layout that was used as input to the compiler. The first step involved taking the CALMA Stream Format data and reading it from the CALMA system onto a VAX 11/780. Appendix D shows the output of the compiler in the form of a file of assertions or facts that represent the data in the Stream Format file from the CALMA system (see Appendix C). A somewhat exhaustive, manual comparison of the annotated hex dump of the stream format file with the assertions file shows that there has been no loss of data in the translation. This example illustrates the feasibility of representing a native CAE/CAD/CAM data base in terms of DBIF.

```

<library> ::= HEADER BGNLIB LIBNAME [ REFLIBS ] [ FONTS ]
           [ ATTRTABLE ] [ STYPTABLE ] [ GENERATIONS ]
           UNITS { <structure> } * ENDLIB

<structure> ::= BGNSTR STRNAME [ STRCLASS ] [ STRTYPE ]
              { <element> } * ENDS

<element> ::= { <boundary> | <path> | <sref> | <aref> | <text> | <node>
              | <box> } { ELKEY { <link> } + } { <property> } * ENDEL

<boundary> ::= BOUNDARY | ELFLAGS | [ PLEX ] LAYER DATATYPE XY

<path> ::= PATH [ ELFLAGS ] [ PLEX ] LAYER DATATYPE [ PATHTYPE ]
           [ WIDTH ] XY

<sref> ::= SREF [ ELFLAGS ] [ PLEX ] SNAME [ <strans> ] XY

<aref> ::= AREF [ ELFLAGS ] [ PLEX ] SNAME [ <strans> ] COLROW XY

<text> ::= TEXT [ ELFLAGS ] [ PLEX ] LAYER <textbody>

<node> ::= NODE [ ELFLAGS ] [ PLEX ] LAYER NODETYPE XY

<box> ::= BOX [ ELFLAGS ] [ PLEX ] LAYER BOXTYPE XY

<textbody> ::= TEXTTYPE [ PRESENTATION ] [ PATHTYPE ]
              [ WIDTH ] [ <strans> ] XY STRING

<strans> ::= STRANS [ MAG ] [ ANGLE ]

<link> ::= LINKTYPE LINKKEYS

<property> ::= PROPATTR PROPVALUE

```

NOTE:    "|" denotes an entity occurring zero or one time.  
          "|" denotes pick one of the entities.  
          "|"*" denotes entities can occur an arbitrary number of times.  
          "|"+" denotes at least one of the entities must be present.

"Reprinted with permission by the Calma Company"

Figure 4.2. Pseudo-BNF description of the CALMA Stream Format.

The second example of using a compiler to produce DBIF, reformats the TEGAS Design Language (TDL) into DBIF format. TDL is an industry recognized format for describing interconnections and simulation models

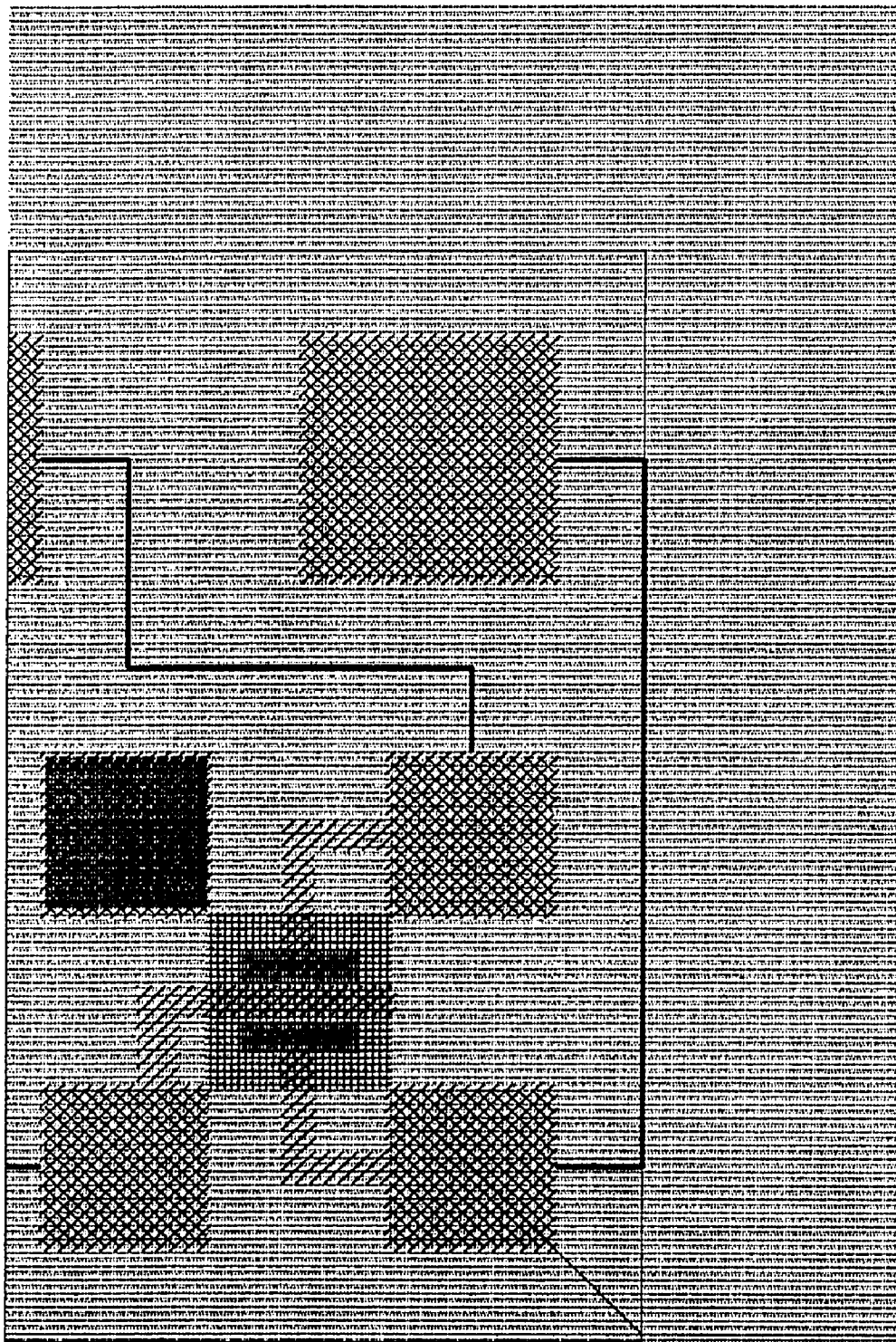


Figure 4.3. Sample CALMA Layout.

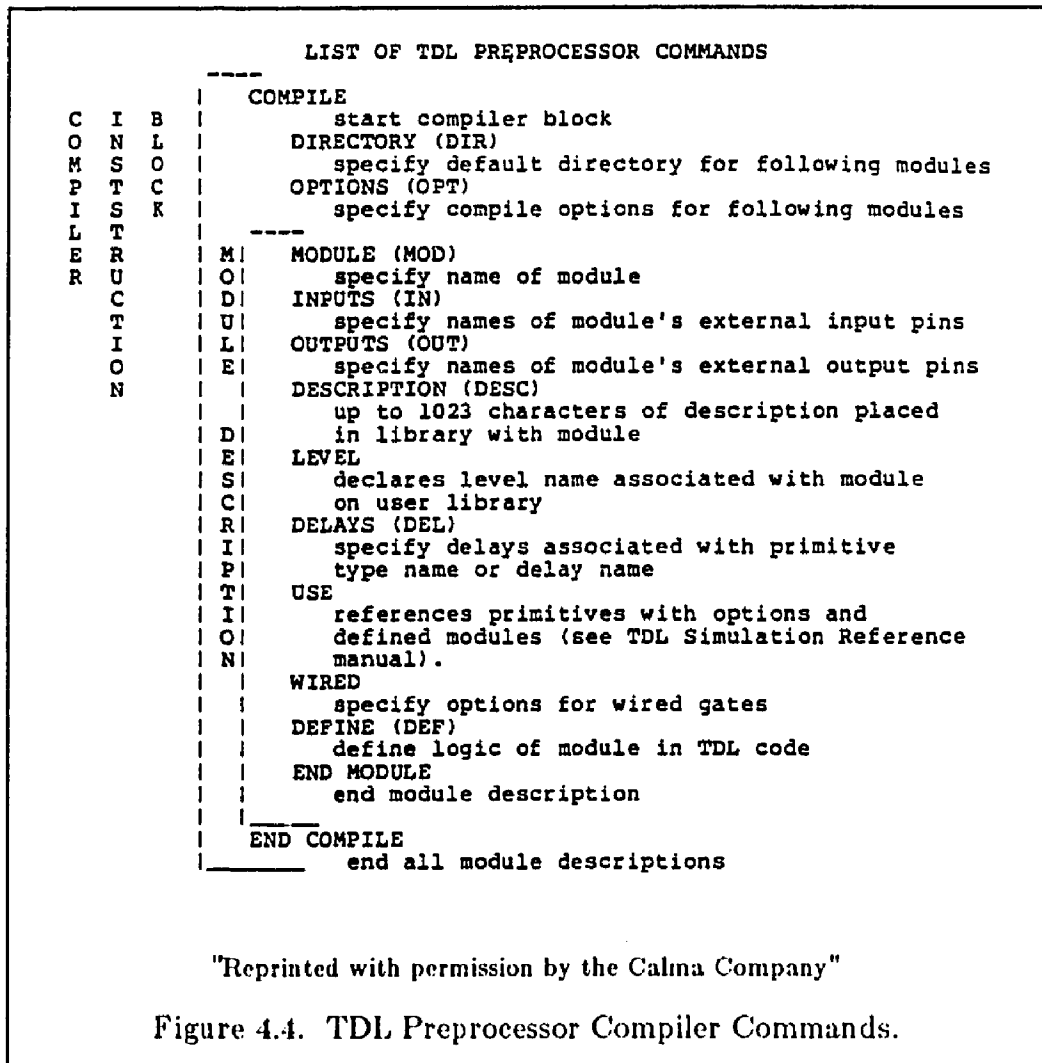
for performing logic simulation. In this example, a schematic data base represented in the TDL format is compiled into a set of assertions or facts that represents the same data content. To perform this, a compiler was produced using Prolog which contained the necessary syntax and semantic knowledge in terms of Prolog statements. Approximately 550 lines of Prolog code were written to perform the compilation.

The TDL preprocessor language has four major parts:

1. Compiler Instructions
2. File Manager Instructions
3. Linker Instructions
4. End

The part of interest in this example is the Compiler and its instructions, which contain a description of the schematic net list along with any models used. Figure 4.4 shows the structure of the TDL Preprocessor Compiler commands. The BNF descriptions of these commands are complex. An excerpt from the August 1983 release of the TDL Preprocessor User & Reference Manual [TDL83] is provided in Appendix H. This excerpt describes the syntax of the DEFINE command which is used to specify signals and the interconnectivity of logic elements.

Figure 4.5 shows a sample TDL input to the compiler. The resulting DBIF is shown in Appendix E. As in the preceding compilation example involving CALMA data, this example show the feasibility of compiling a native CAE/CAD/CAM data format into DBIF. In this case, however,



rather than a binary data base, a man-readable language has been compiled into DBIF.

*Translate Engine:* Once compiled DBIF exists it is input into the translate engine as shown in Figure 3.15. The basic translate engine consists of the following nine steps:

1. Initialize
2. Read in and load source facts and input rules into Prolog da-

```

COMPILE;
OPTIONS CATALOG, XREF;
DIRECTORY RPH;
MODULE JKFF/GATE/1/RPH;
INPUTS CLOCK, J, K, PS, PC;
OUTPUTS OQ, OQB;
DESCRIPTION THE MODULE IS A MASTER/SLAVE JK FLIP-FLOP
          WITH PRESET AND PRECLEAR LINES.  ;
          "(SEE TDL REF. MANUAL P.71)"
DELAYS NANDEL/3,2,4/, NOT/3,2,4/;

      "THE FOLLOWING TWO LINES CREATE TWO DIFFERENT TYPES
      BASED ON THE PRIMITIVE ELEMENT NAND.
      3-NAND IS THE SAME AS NAND.
      2-NAND IS DECLARED TO BE A 2 INPUT NAND. "

USE 3-NAND = NAND(3,1) /NANDEL/,
    2-NAND = NAND(2,1) /NANDEL/;

DEFINE
DEV1(NAND-A) = 3-NAND(J,QB,CLOCK);
DEV2(NAND-B) = 3-NAND(K,Q,CLOCK);
DEV3(NAND-C) = NAND(PS,NAND-A,NAND-D);
DEV4(NAND-D) = NAND(PC,NAND-B,NAND-C);
DEV5(I) = NOT(CLOCK);
DEV6(NAND-E) = 2-NAND(NAND-C,I);
DEV7(NAND-F) = 2-NAND(NAND-D,I);
G-NAND(Q) = NAND(NAND-E,QB);
H-NAND(QB) = NAND(NAND-F,Q);
DEV8(OQ) = NOT(Q);
DEV9(OQB /1/) = NOT(QB);
END MODULE;
END COMPILE;

```

Figure 4.5. Sample TDL Input.

tabase.

3. Generate generic facts.
4. Create kept facts from the input to generic form translation.
5. Reset fact/rule (Prolog) database.

6. Read in and load generic facts and output rules into Prolog database.
7. Generate target facts.
8. Add kept facts from the output translation to those created in step 4.
9. Clean-up and reset Prolog database.

Figure 4.6 shows this process and the interface with the various data files.

It is important to note that there is no knowledge of any native CAE/CAD/CAM data format or the generic predicates embedded within the translate engine. In fact, the same translate engine code (in Prolog) was used with distinct sets of source/target DBIF's and two different classes of generic predicates. Chapter 5 describes these test cases and the results of translation. This translate engine requires approximately 120 lines of Prolog code.

A slightly different sequence of steps is necessary if a translation is performed on not only source DBIF, but also uses previously kept facts. This might be needed if a data base were transported from system A to system B and then back again. Assuming kept facts were generated in going from A to B, then in order to reverse the process the same kept facts would be needed to result in the complete database of system A. In this case, two additional steps are added to the translate engine:

- 2A. Read in and load previously kept facts into the Prolog database.

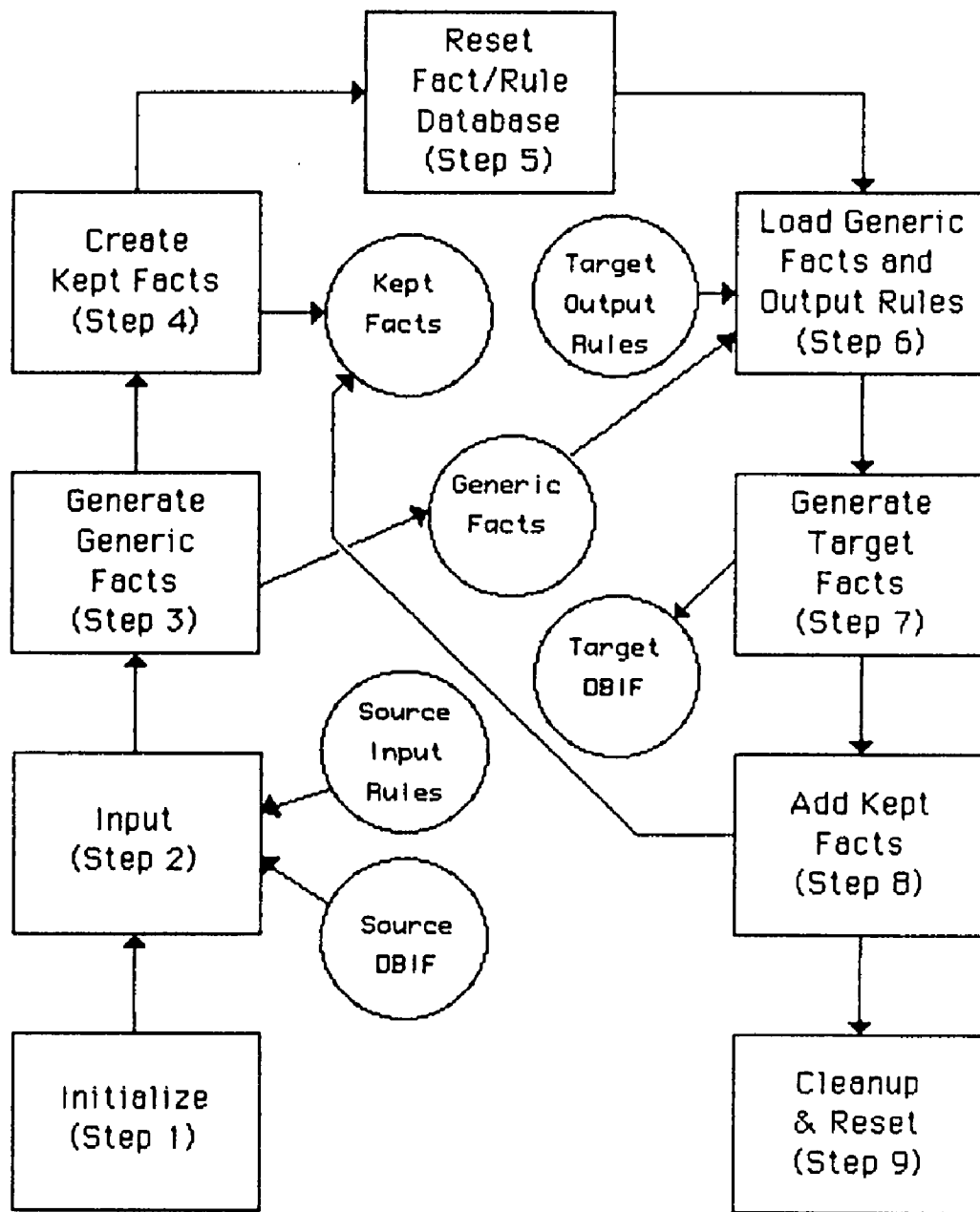


Figure 4.6. Translate Engine Processing Flow.



6A. Read in and load previously kept facts into the Prolog database.

These previously generated kept facts are removed before steps 4 and 8 in order to avoid having the previously saved data from adding to the newly created kept facts. Figure 4.7 shows this process flow utilizing previously kept facts. This modified translate engine consists of approximately 130 lines of Prolog code.

*Formatter:* Once the translate engine provides target DBIF, formatter modules (one for each unique target system type) convert DBIF into the native CAE/CAD/CAM format. Two formatters were produced in the course of this study. Again the goal was to show feasibility.

The first formatter converts DBIF into CalTech Intermediate Form (CIF). CIF is a file format which describes graphic features of VLSI design layouts [Mead80]. The basic CIF commands are

- P - Polygon, defined by its vertices.
- B - Box, defined by its length, width, center, and orientation.
- R - Round flash, defined by its diameter and center.
- Wire - Wire, defined by its vertices and width.
- L - Layer specification.
- DS - Define symbol.
- DF - Finish symbol definition.
- DD - Delete symbol definition.

---

Mead and Conway, INTRODUCTION TO VLSI SYSTEMS,  
© 1980, Addison-Wesley, Reading, Massachusetts. Pgs. 115  
through 126. Reprinted with permission.

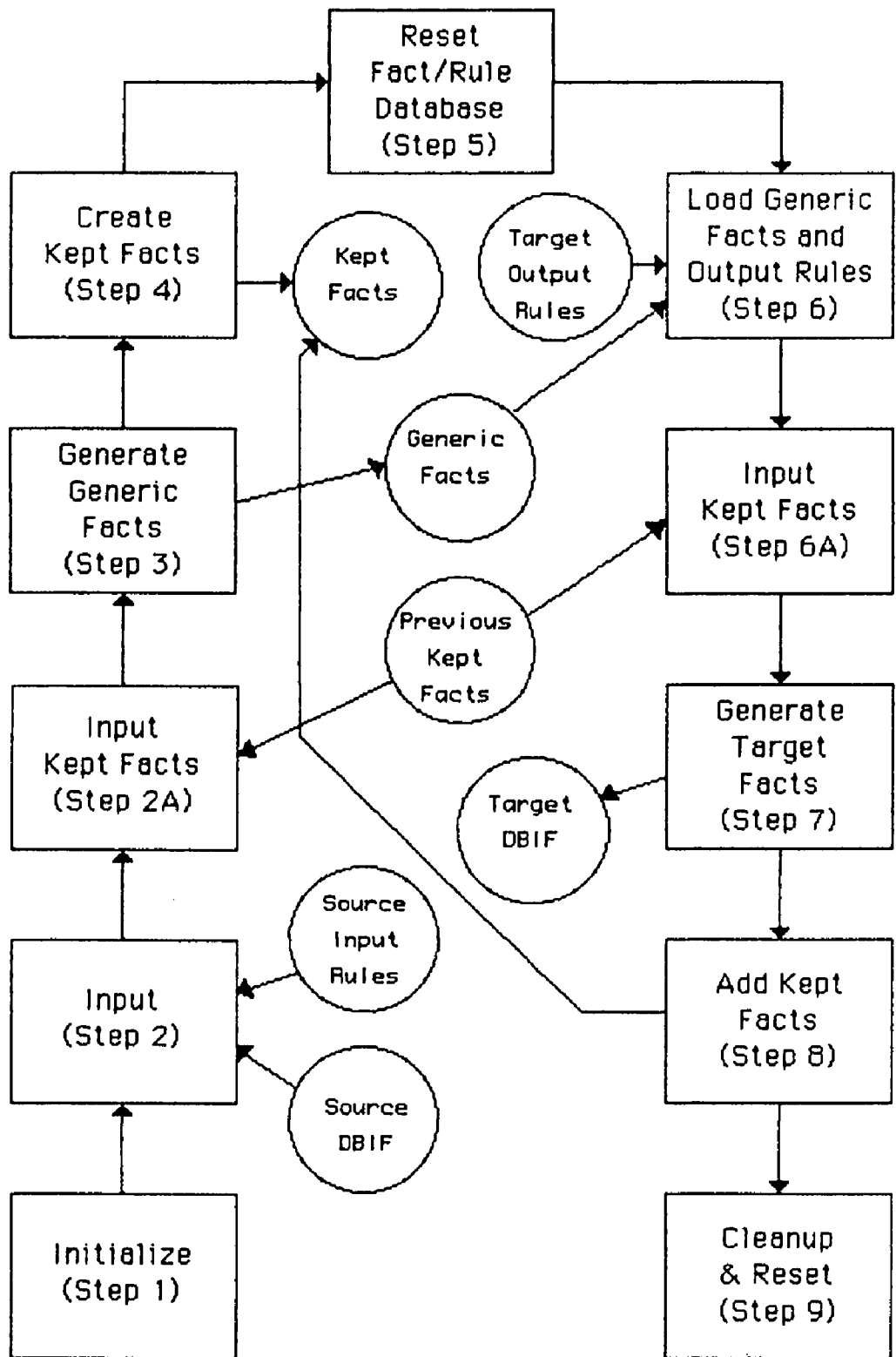


Figure 4.7. Translate Engine Processing Flow with Kept Facts.

C - Call symbol and provide transformation.

In addition to these basic commands are facilities for commenting the file and extending the command repertoire with user extensions. A complete syntax definition is provided in Figure 4.8 as published in [Mead80].

cifFile	= blank ; command ; semi ; endCommand blank .
command	= primCommand   defDeleteCommand : defStartCommand semi ; blank ; primCommand semi ; defFinishCommand .
primCommand	= polygonCommand   boxCommand   roundFlashCommand   wireCommand   layerCommand   callCommand   userExtensionCommand   commentCommand .
polygonCommand	= 'P' path .
boxCommand	= 'B' integer sep integer sep point sep point .
roundFlashCommand	= 'R' integer sep point .
wireCommand	= 'W' integer sep path .
layerCommand	= 'L' blank shortname .
defStartCommand	= 'D' blank "S" integer sep integer sep integer .
defFinishCommand	= 'D' blank "F" .
defDeleteCommand	= 'D' blank "D" integer .
callCommand	= 'C' integer transformation .
userExtensionCommand	= digit userText .
commentCommand	= "!" commentText "
endCommand	= "E" .
transformation	= blank   "T" point "M" blank   "X" "M" blank   "Y" "R" point .
path	= point sep point . .
point	= vInteger sep vInteger .
vInteger	= sep " " integerD .
integer	= sep integerD .
integerD	= digit   digit . .
shortname	= c   c   c   c . .
c	= digit   upperChar .
userText	= userChar . .
commentText	= commentChar . commentText "!" commentText "!" commentText .
semi	= blank " ; " blank .
sep	= upperChar blank .
digit	= "0"   "1"   "2"   "3"   "4"   "5"   "6"   "7"   "8"   "9" .
upperChar	= "A"   "B"   "C"   . . .   "Z" .
blank	= any ASCII character except digit, upperChar, " ", "!", "(", ")", or " ; " .
userChar	= any ASCII character except " ; " .
commentChar	= any ASCII character except "!" or "!" .

Figure 4.8. CIF Syntax Description.

The CIF formatter based upon this syntax, developed by M. Caro [Caro83], consists of approximately 350 lines for FORTRAN code. The sample CIF DBIF used to demonstrate the formatter is included in Appendix F. This is an early version of the DBIF and is slightly different than the DBIF that

was used in experiments with the translate engine. The native CIF output by the formatter is shown in Appendix G.

A second formatter was written for the hypothetical relational database native format, DR1, described in Chapter 3. The sample DBIF for DR1 is shown in Figure 4.9. The formatter is quite trivial due to the power of Prolog and the simplicity of the DR1 tabular form. The Prolog code, shown in Figure 4.10, consists of approximately 20 lines of code. The resulting tabular form for DR1 is suitable for loading into a relational DBMS (see Figure 4.11). Note that the order of the facts in Figure 4.9 is immaterial. All of the facts are stored in the Prolog database and can be accessed by the code of Figure 4.10.

A walk through the Prolog code for "dr1wrt" in Figure 4.10 illustrates the formatter process and also how Prolog operates in general. The execution of the formatter begins in line 1 with the "[dr1.dat]" clause which consults the file "dr1.dat". This means that Prolog facts and rules contained in this file will be added to the Prolog database. Continuing with line 1, the "tell" clause directs any further output to the file "dr1out.dat". The next clause, "fail", causes Prolog to backtrack over previously encountered clauses, looking for instantiated variables which could be associated with a new value. In this case, there were no instantiated variables in this line, so backtracking regresses back to the head of line 1, "dr1wrt". Since this entire rule evaluates to false (due to the "fail" clause), Prolog looks for another rule for "dr1wrt" which might evaluate to true.

```

dbid(schemex1,dr1,'1.0','1/11/84:17:06').
content([signal(s7),comp__name(c1),pin(o2),comp__type(conn),
  connect(s7,c1,o2),has(c1,conn),
signal(s7),comp__name(b),pin(y),comp__type(t2),
  connect(s7,b,y),has(b,t2),
signal(s6),comp__name(c1),pin(o1),comp__type(conn),
  connect(s6,c1,o1),has(c1,conn),
signal(s6),comp__name(c),pin(y),comp__type(t2),
  connect(s6,c,y),has(c,t2),
signal(s5),comp__name(b),pin(i1),comp__type(t2),
  connect(s5,b,i1),has(b,t2),
signal(s5),comp__name(c),pin(i2),comp__type(t2),
  connect(s5,c,i2),has(c,t2),
signal(s5),comp__name(a),pin(qn),comp__type(t1),
  connect(s5,a,qn),has(a,t1),
signal(s4),comp__name(c),pin(i1),comp__type(t2),
  connect(s4,c,i1),has(c,t2),
signal(s4),comp__name(a),pin(q),comp__type(t1),
  connect(s4,a,q),has(a,t1),
signal(s3),comp__name(b),pin(i2),comp__type(t2),
  connect(s3,b,i2),has(b,t2),
signal(s3),comp__name(c1),pin(i3),comp__type(conn),
  connect(s3,c1,i3),has(c1,conn),
signal(s2),comp__name(a),pin(s),comp__type(t1),
  connect(s2,a,s),has(a,t1),
signal(s2),comp__name(c1),pin(i2),comp__type(conn),
  connect(s2,c1,i2),has(c1,conn),
signal(s1),comp__name(a),pin(r),comp__type(t1),
  connect(s1,a,r),has(a,t1),
signal(s1),comp__name(c1),pin(i1),comp__type(conn),
  connect(s1,c1,i1),has(c1,conn),
dummy]).

```

Figure 4.9. Sample DBIF for DR1.

In a like manner, line 2 begins by trying to instantiate the variable F with something such that "content(F)" is a fact. Since the data from "dr1.dat" contains a clause "content([...])", F is associated with the list of terms contained between the parentheses of the "content" term (see Figure 4.9). Continuing with line 2 of the Prolog code (Figure 4.10), the "loadfact" term is evaluated with the current value for F, assigned in the "content"

```

/* 1 */ dr1wrt:-['dr1.dat'],tell('dr1out.dat'),fail.
/* 2 */ dr1wrt:-content(F),loadfact(F),fail.
/* 3 */ dr1wrt:-connect(S,C,P),signal(S),
/* 4 */         comp_name(C),pin(P),has(C,T),
/* 5 */         comp_type(T),putlft(S,0,12,P1),
/* 6 */         putlft(C,P1,12,P2),
/* 7 */         putlft(P,P2,12,P3),
/* 8 */         putlft(T,P3,12,P4),
/* 9 */         putstring("."),nl,fail.
/* 10 */        dr1wrt:-told.

loadfact([]).
loadfact([H|T]):-retr(H),assertz(H),loadfact(T).

retr(R):-retract(R),fail.
retr(R).

putlft(Dat,Cur,Leng,New):-name(Dat,D),length(D,L),putlft2(D,Cur,Leng,L),
                          New is Cur+Leng.
putlft2(D,C,L,L2):-L=L2,putstring(D).
putlft2(D,C,L,L2):-L<L2,firstN(D,L,D2),putstring(D2).
putlft2(D,C,L,L2):-L>L2,M is L-L2,putstring(D),tab(M).

putstring([]).
putstring([H|T]):-put(H),putstring(T).

```

Figure 4.10. Prolog Code for the DR1 Formatter.

s7	c1	o2	conn .
s7	b	y	t2 .
s6	c1	o1	conn .
s6	c	y	t2 .
s5	b	i1	t2 .
s5	c	i2	t2 .
s5	a	qn	t1 .
s4	c	i1	t2 .
s4	a	q	t1 .
s3	b	i2	t2 .
s3	c1	i3	conn .
s2	a	s	t1 .
s2	c1	i2	conn .
s1	a	r	t1 .
s1	c1	i1	conn .;

Figure 4.11. Tabular Form of DR1 Data.

clause. "Loadfact" merely picks each term in the list and appends it to the Prolog database. The "fail" term at the end of line 2 causes this line to evaluate false, and Prolog backtracks, looking for another "dr1wrt" rule which will evaluate to true.

Line 3 begins with the term "connect(S,C,P)", and the Prolog database is searched looking for a "connect" term. The first such term is "connect(s7,c1,o2)", and the variables "S", "C", and "P" are instantiated with the constants "s7", "c1", and "o2", respectively. Having satisfied the "connect" term, Prolog continues on to the "signal(S)" term. At this point, "S" has been instantiated with "s7", so Prolog looks into the database to see if "signal(s7)" is true. Since this is so, Prolog continues with the next term in the series beginning with line 3 (dr1wrt).

Terms continue to be satisfied in this instance through the term "nl" (new line). At this point, the first line of "dr1out.dat" has been output (see Figure 4.11). Upon encountering the next term, "fail" (line 9), backtracking resumes. Each term in this rule (lines 3-9) is re-examined in reverse order to determine if an alternative value can be used to re-instantiate a variable such that the term containing the variable is still true. The terms "putstring", "putlft", "comp_type", "has", "pin", "comp_name", and "signal" do not instantiate any variables which can be re-assigned new values. The backtracking process finally arrives at "connect(S,C,P)" where the variables "S", "C", and "P" were instantiated.

Searching through the Prolog database from where we left off, Prolog encounters an alternate definition for "connect". "Connect(s7,b,y)" instantiates these values to the variables "S", "C", and "P", backtracking

stops, and processing proceeds forward again with the new values assigned. The forward process continues until "fail" is reached. This time the second line of "dr1out.dat" is output (see Figure 4.11). The flow alternates backward and forward, each time using a different instance of "connect" in the Prolog database (see Figure 4.9). For each instance of "connect", a new line of "dr1out.dat" (Figure 4.11) is produced.

Once the last instance of "connect" has been used in this process, backtracking back to the "connect" term in line 3 will fail, and the entire "dr1wrt" rule will finally be evaluated false. Backtracking will continue to lead to the definition for "dr1wrt" in line 10. Here, the term "told" means to stop "telling", i.e., the file "dr1out.dat" is closed since this is the file opened with the last previously executed "tell" term in line 1. "Told" evaluates true, and so does "dr1wrt". The evaluation of "dr1wrt" stops and the processing is over.

#### **4.2.2 Data Elements**

*DBIF*: Thus far, the processes of the prototype have been described and examples presented. Complementing the processes are the data bases which provide the information to be transported and the rules to be used to accomplish this.

The DBIF, as described earlier, provides a standard format for processing. The syntax of DBIF in the prototype implementation is just the syntax of Prolog terms. Any fact expressible in Prolog can be used as part of a DBIF for a native CAE/CAD/CAM form.



In practice, each native format contains a number of basic data elements which are combined to form a complete native database or language segment. For example, the fields in a relational database are candidates for terms in Prolog. Several examples have been presented and more substantial test cases will be presented in Chapter 5.

*Generic Predicates:* As discussed in section 3.5.2 and in 4.2.1, describing the translate engine, generic predicates are used to assist in a general n-way translation scheme (refer to Figure 3.11). For each class of CAE/CAD/CAM data, a set of generic predicates is provided. Figures 3.17 and 3.18 show generic predicates for the "logical" and "physical" classes of data. The encoding of these generic predicates into Prolog is shown in Figures 4.12 and 4.13. These predicates provide a set of reference terms in which translate rules are expressed.

```
node(X).
net(X).
box(X).
box_type(X).
net_type(X).
node_type(X).
node_dir(X).
connected(X,Y,Z).
has_(X,Y).
```

Figure 4.12. Prolog Encoding of Generic Predicates for Logical Data.

```

polygon_(P).
wire_(W).
macro_def(S).
macro_call(S,N).
scale_(S).
layer_(X,L).
vertex_(S,X,Y,I).
width_(X,W).
orient_(X,AX,AY,AZ).
has_(X,Y).
magnif_(S,M).
relative_orient(S).
relative_magnif(S).
text_(T).
textval_(T,Str).
h_just(T,N).
v_just(T,N).
tfont_(T,F).

```

Figure 4.13. Prolog Encoding of Generic Predicates for Physical Data.

*Rules:* Using the generic predicates, rules are written to show how each DBIF is re-written into generic predicates. Likewise, rules are also prescribed for expressing each generic predicate in terms of DBIF predicates. Consider an example of the use of rules and generic predicates using the hypothetical DR2 database shown in Figure 4.14 in DBIF form. (The corresponding schematic diagram was shown in Figure 3.6).

In order to translate this DBIF into generic predicates, rules must be provided to indicate how the translation is performed. The set of generic predicates used for logical data (Figure 4.12) was actually used in experiments with the prototype implementation. Contrasting these predicates with the terms used in the DR2 DBIF, Figure 4.15 shows the mapping from

```

dbid(exsch1,dr2,'3.0','1/4/84:16:01').
content([body(a),body(b),body(c),body(c1),
        bt(t1),bt(t2),bt(conn),
        ndt(i1),ndt(i2),ndt(i3),ndt(y),
        ndt(o1),ndt(o2),ndt(r),ndt(s),
        ndt(q),ndt(qn),
        has(t1, qn),has(t1, q),has(t1, s),has(t1, r),
        has(t2, i1),has(t2, i2),has(t2, y),
        has(conn,i1),has(conn,i2),has(conn,i3),
        has(conn,o1),has(conn,o2),
        has(a, t1), has(b, t2), has(c, t2), has(c1, conn),
        signal(s1),signal(s2),signal(s3),signal(s4),signal(s5),
        signal(s6),signal(s7),
        vertex(a, r, [10,90], 1),vertex(a, s, [10,80], 1),
        vertex(a, q, [20,90], 1),vertex(a, qn, [20,80], 1),
        vertex(b, i1, [30,40], 1),vertex(b, i2, [30,20], 1),
        vertex(b, y, [40,30], 1),
        vertex(c, i1, [50,80], 1),vertex(c, i2, [50,60], 1),
        vertex(c, y, [60,70], 1),
        vertex(c1, i1, [0,90], 1),vertex(c1, i2, [0,80], 1),
        vertex(c1, i3, [10,20], 1),vertex(c1, o1, [70,70], 1),
        vertex(c1, o2, [60,30], 1),
        vertex(s1, [0,90], 1),vertex(s1, [10,90], 2),
        vertex(s2, [0,80], 1),vertex(s2, [10,80], 2),
        vertex(s3, [10,20], 1),vertex(s3, [30,20], 2),
        vertex(s4, [20,90], 1),vertex(s4, [50,80], 2),
        vertex(s5, [20,80], 1),vertex(s5, [25,60], 2),
        vertex(s5, [50,60], 3),vertex(s5, [30,40], 4),
        vertex(s6, [60,70], 1),vertex(s6, [70,70], 2),
        vertex(s7, [40,30], 1),vertex(s7, [60,30], 2)]).

```

Figure 4.14. DBIF for Sample DR2 Data Base.

DR2 terms to generic predicates. Note that the term "vertex" does not have an exact generic counterpart. Similarly, generic predicates, "node", "net_type", "node_type", and "node_dir" do not have equivalents in DR2. While the concept of a "node direction" (node_dir) may not be necessary to describe a logic network, certainly, the "node" concept is.

Indeed, there are implicit nodes in DR2; however, node types are used in DR2 to refer to the nodes.

DR2		GENERIC
signal(X)	→	net(X)
bt(X)	→	net(X)
ndt(X)	→	net(X)
body(X)	→	net(X)
vertex(W,X,Y,Z)	→	?
vertex(X,Y,Z)	→	?
has(X,Y)	→	has_(X)
?	→	node(X)
?	→	node(X)
?	→	node(X)
vertex(?)	→	connected(X,Y,Z)

Figure 4.15. Mapping of DR2 Terms onto Generic Predicates.

These mismatches between DR2 and a generic definition of "logical" class data are an example of the delta problem described in previous chapters. It is necessary that rules be written to deal with these mismatches. Figure 4.16 shows a set of rules which not only perform the transformation from DR2 DBIF into a generic form, but also generates generic data without DR2 equivalent. Also, a provision is made to keep DR2 data which cannot be represented in terms of generic predicates.

The rules which translate the DR2 terms, "signal", "bt", "ndt", and "body" are straight-forward, and merely reflect a one-to-one mapping. The rule for creating generic "nodes" is more complex. It relies upon DR2 terms and creates the missing generic relationships. the first "node(X)" rule says

```

net(X):-signal(X).

box(X):-body(X).

box_type(X):-bt(X).

connected(Box,Signal,Node):-vertex(Signal,V,1),vertex(Box,Ntype,V,1),
                               has_(Node,Ntype),node(Node),has_(Box,Bt),has_(Bt,Node).

keep(vertex(X,Y,V,1)):-vertex(X,Y,V,1).

node(X):-keep(node(X)).
node(X):-!,ndt(Y),bt(Z),has(Z,Y),gensym(node_,X),asserta(node(X)),
                               asserta(has_(X,Y)),asserta(has_(Z,X)),retract(has(Z,Y)).

node_type(X):-keep(node_type(X)).
node_type(Y):-!,ndt(Y).

net_type(X):-keep(net_type(X)).
net_type(X):-!,signal(Y),gensym(nt_,X),assert(has_(Y,X)).

node_dir(X):-keep(node_dir(X)).
node_dir(X):-!,ndt(Y),has_(Z,Y),node(Z),gensym(ndr_,X),assert(has_(Z,X)).

has_(X,Y):-body(X),has(X,Y),bt(Y).

```

Figure 4.16. Rules for Transforming DR2 into a Generic Form.

that  $X$  is a node if there is a kept fact that  $X$  is a node. This is for the case that not only a DR2 data base is being input but also a data base of previously kept data. This scenario corresponds to the process flow shown in Figure 4.7 as opposed to Figure 4.6 where previously kept facts don't exist. The second "node(X)" rule first looks for a node type  $Y$  ( $ndt(Y)$ ) and a body type  $Z$  ( $bt(Z)$ ) such that  $Z$  has  $Y$ . For each  $Y$  and  $Z$  meet this condition, the function "gensym" will generate a unique constant of the form "node_x" where  $x$  is 1,2,3,... . The constant will then be considered a node name and that fact inserted into the Prolog database ("asserta(node(X))").

Appropriate relationships between this new node, its node type, and the containing body type will then be asserted ("asserta(has_(X,Y))" and "asserta(has_(Z,X))").

This transformation from DR2 to generic form illustrates the different philosophy between the two systems. In DR2, each body type (bt) has a set of node types (ndt). A body has a body type (bt). Vertices connect signals to the node_types of a particular body (which is an instance of a body type). From the generic point of view, a box (= DR2 body) has a box type, a box type has a node, and a node has a node type. The different representations are illustrated in Figure 4.17.

One additional rule of interest is the one for the generic predicate, "connected". Since the generic predicates are evaluated in sequence (as shown in Figure 4.12), it is possible for one predicate's rule to rely upon the Prolog database facts generated by the rules for predicates processed earlier in the sequence. Such is the case for the "connected" rule. In this rule, the Prolog database is searched for a vertex held in common between a signal and a node type. Then the node type is mapped to a node generated earlier by the node rule. The previously asserted "has_" facts are used in the mapping.

In a like manner, each rule is processed and the data transformed from the source DBIF to the target DBIF. A complete scenario follows.

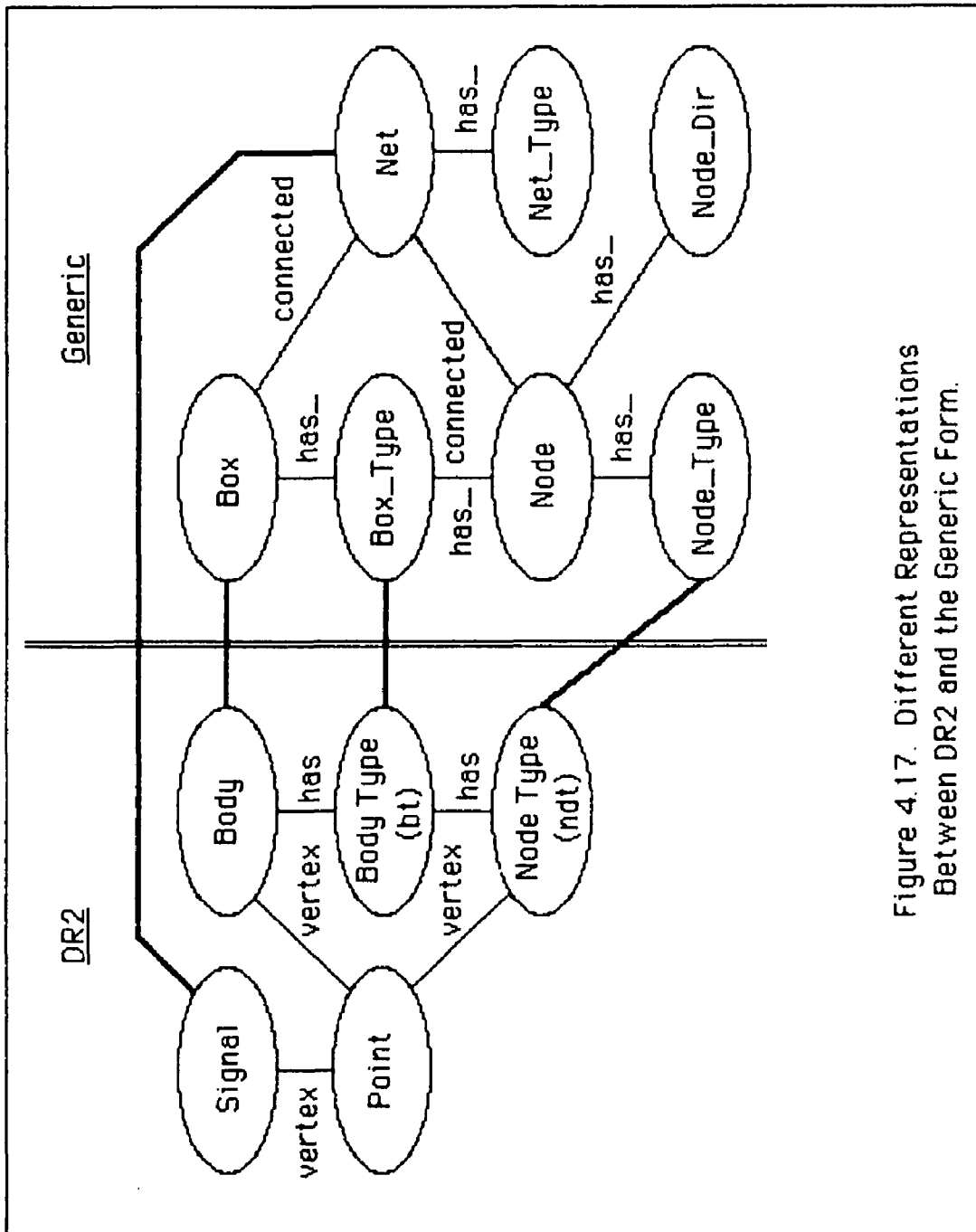


Figure 4.17. Different Representations Between DR2 and the Generic Form.

### 4.3 Operational Scenario

To illustrate how the various prototype processing elements function, consider the two native data formats DR2 and DR1. The native form for the DR2 data was shown in Figure 3.8 and for DR1 in Figure 3.7. By following the flow diagram in Figure 4.6, the DR2 DBIF can be transformed into DR1 DBIF. Executing the Prolog code which implements this flow diagram results in the system log shown in Figure 4.18.

The Prolog term "translate(...)" in this figure begins the execution. The arguments to "translate" indicate that the source DBIF is the file, "dr2.dat", and that the target DBIF will be written into file, "dr1.dat", the new data base identifier (dbid) in this file will be "exschT", version "10.0" written on 1/10/84 at 2:53pm. The target system is "DR1" (argument 5). The file "dr1K.dat": (arg. 3) will be used to store any kept facts generated in the process. The output lines in the system log which follow the "translate" term indicate the steps in the process.

As shown in Figure 4.6, step 1 involves initialization. Step 2 reads in the source DBIF and the source input rules. The system log (Figure 4.18) shows this action. The line "dr2.dat consulted ..." indicates that the source DBIF (Figure 4.14) was read in. The next line after the "T r a n s l a t e" banner in the system log echoes the "dbid" from "dr2.dat". The second argument of this "dbid" term indicates the source system "dr2". Based on this argument, the file (dr2)in.rul (Figure 4.16) is assumed to contain the source input rules. These are read in next. This initial start up segment encompassing steps 1 and 2 is not complete.



```

CProlog version 1.4d.edai
[ Restoring file /u/ua/hooper/tran10.env ]

yes
| ?- ['/u/ua/hooper/pro/gensym.pro'].
/u/ua/hooper/pro/gensym pro consulted 812 bytes 0.3 sec.

yes
| ?- translate('dr2.dat','dri.dat','dr1K.dat',exschT,dri,'10.0','1/10/84:14:53').
dr2.dat consulted 2832 bytes 0.65 sec.

>> Translate: V1.0 <<

dbid(exsch1,dr2,3.0,1/4/84:16:01)
dr2in.rul consulted 1472 bytes 0.46667 sec.
Start Up 2.42 sec.
T=node(_993) 12 facts.
T=net(_993) 7 facts.
T=box(_993) 4 facts.
T=box_type(_993) 3 facts.
T=net_type(_993) 7 facts.
T=node_type(_993) 10 facts.
T=node_dir(_993) 12 facts.
T=macro_call(_993) 0 facts.
T=macro_def(_993) 0 facts.
T=connected(_993,_994,_995) 15 facts.
T=has(_993,_994) 47 facts.
T=end_of_file 0 facts.
117 facts total
Generic 8.35 sec.
Keep 0.62 sec.
Unload 1.45 sec.
drlout.rul consulted 948 bytes 0.31667 sec.
T=signal(_3071) 7 facts.
T=pin(_3071) 10 facts.
T=comp_name(_3071) 4 facts.
T=comp_type(_3071) 3 facts.
T=connect(_3071,_3072,_3073) 15 facts.
T=has(_3071,_3072) 4 facts.
T=end_of_file 0 facts.
43 facts total.
Phase 2 5.33 sec.
Output Keep 3.88 sec.

Total time is 23.13 sec.

yes
| ?- halt.

[ Prolog execution halted ]

```

Figure 4.18. System Log of the Prototype Execution.

Step 3 involves the generation of generic data. The generic predicates (Figure 4.12) are read in one at a time (from "gencon.rul"), and all possible values for any variables contained in the generic predicate are generated. As each generic predicate is read in, it is echoed out into the sys-

tem log, and the number of facts generated for that predicate is printed also (e.g., "T=node(_993) 12 facts." ^(4.2) ). At the end of step 3, all generic data has been generated (see Figure 4.19). The system log shown that 117 generic facts were generated in total. The log also indicates that 8.35 seconds of CPU ^(4.3) time were expended in producing the generic data.

Step 4 involves creating kept facts which would be lost in translation between DR2 and the generic form. These are written into "drIK.dat". Additional kept facts are added in step 8. This step marked "Keep" took 0.62 CPU seconds.

Step 5 resets the Prolog database by removing any DR2 data and kept facts. The system log identifies this step as "Unload" and reports that it took 1.45 seconds.

Step 6 reads in the generic data created in step 3 and also the target output translation rules (see Figure 4.20), "drIout.rul". This event is reported in the system log, "drIout.rul.consulted ...".

Step 7 generates the target DBIF. As each DR1 predicate is read, it is echoed out and the number of facts generated is reported. The target output rules read in from step 6 are used to map from the generic data (Figure 4.19) into DR1 predicates. As the system log shows, 7 signals were

---

(4.2) The numbered Prolog variables (e.g., _993) beginning with an underscore are the internal Prolog names for variables.

(4.3) The CPU is a VAX 11/750 running LCC LOCUS, a distributed UNIX operating system.

```

node(node_1). node(node_2). node(node_3).
node(node_4). node(node_5). node(node_6).
node(node_7). node(node_8). node(node_9).
node(node_10). node(node_11). node(node_12).
net(s1). net(s2). net(s3). net(s4).
net(s5). net(s6). net(s7).
box(a). box(b). box(c). box(c1).
box_type(t1). box_type(t2). box_type(conn).
net_type(nt_1). net_type(nt_2).
net_type(nt_3). net_type(nt_4).
net_type(nt_5). net_type(nt_6).
net_type(nt_7).
node_type(i1). node_type(i2). node_type(i3).
node_type(y). node_type(o1). node_type(o2).
node_type(r). node_type(s). node_type(q).
node_type(qn).
node_dir(ndr_1). node_dir(ndr_2).
node_dir(ndr_3). node_dir(ndr_4).
node_dir(ndr_5). node_dir(ndr_6).
node_dir(ndr_7). node_dir(ndr_8).
node_dir(ndr_9). node_dir(ndr_10).
node_dir(ndr_11). node_dir(ndr_12).
connected(c1,s1,node_2). connected(a,s1,node_9).
connected(c1,s2,node_4). connected(a,s2,node_10).
connected(c1,s3,node_5). connected(b,s3,node_3).
connected(a,s4,node_11). connected(c,s4,node_1).
connected(a,s5,node_12). connected(c,s5,node_3).
connected(b,s5,node_1). connected(c,s6,node_6).
connected(c1,s6,node_7). connected(b,s7,node_6).
connected(c1,s7,node_8).
has_(t1,node_12) has_(node_12,qn).
has_(t1,node_11). has_(node_11,q).
has_(t1,node_10). has_(node_10,s).
has_(t1,node_9). has_(node_9,r).
has_(conn,node_8). has_(node_8,o2).
has_(conn,node_7). has_(node_7,o1).
has_(t2,node_6). has_(node_6,y).
has_(conn,node_5). has_(node_5,i3).
has_(conn,node_4). has_(node_4,i2).
has_(t2,node_3). has_(node_3,i2).
has_(conn,node_2). has_(node_2,i1).
has_(t2,node_1) has_(node_1,i1).
has_(a,t1). has_(b,t2). has_(c,t2).
has_(c1,conn). has_(s1,nt_1). has_(s2,nt_2).
has_(s3,nt_3). has_(s4,nt_4). has_(s5,nt_5).
has_(s6,nt_6). has_(s7,nt_7). has_(node_2,ndr_1).
has_(node_1,ndr_2). has_(node_4,ndr_3).
has_(node_3,ndr_4). has_(node_5,ndr_5).
has_(node_6,ndr_6). has_(node_7,ndr_7).
has_(node_8,ndr_8). has_(node_9,ndr_9).
has_(node_10,ndr_10). has_(node_11,ndr_11).
has_(node_12,ndr_12).

```

Figure 4.19. Generic Data Generated by the Prototype Translate Engine.

generated. After all target facts have been generated, their number is reported (43 in this case) and the end of step 7 is reported Phase 2 5.33 sec.". The 43 target DBIF facts are shown in Figure 4.21

```

signal(X):-net(X).
pin(X):-node_type(X).
comp_name(X):-box(X).
comp_type(X):-box_type(X).
connect(S,C,P):-connected(C,S,N),has_(C,CT),has_(CT,N),has_(N,P),
                node_type(P).
has(X,Y):-box(X),has_(X,Y),box_type(Y).
keep(node_dir(X)):-node_dir(X).
keep(has_(X,Y)):-node(X),has_(X,Y),node_type(Y).
keep(has_(X,Y)):-node(X),has_(X,Y),node_dir(Y).
keep(has_(X,Y)):-net(X),has_(X,Y),net_type(Y).
keep(has_(X,Y)):-box_type(X),has_(X,Y),node(Y).

```

Figure 4.20. DR1 Target Output Rules.

```

dbid(exschT,dr1,'10.0','1/10/84:14:53').
content([ signal(s7),signal(s6),signal(s5),
signal(s4),signal(s3),signal(s2),signal(s1),
pin(qn),pin(q), pin(s), pin(r), pin(o2),pin(o1),
pin(y),pin(i3),pin(i2),pin(i1),
comp_name(c1),comp_name(c),comp_name(b),
comp_name(a),
comp_type(conn),comp_type(t2),comp_type(t1),
connect(s7,c1,o2),connect(s7,b,y), connect(s6,c1,o1),
connect(s6,c,y), connect(s5,b,i1), connect(s5,c,i2),
connect(s5,a,qn), connect(s4,c,i1), connect(s4,a,q),
connect(s3,b,i2), connect(s3,c1,i3),connect(s2,a,s),
connect(s2,c1,i2),connect(s1,a,r), connect(s1,c1,i1),
has(c1,conn),has(c,t2),has(b,t2),has(a,t1),dummy]).

```

Figure 4.21. Target DR1 DBIF.

Step 8 adds kept facts to "dr1K.dat". These new kept facts are the result of translating generic predicates into target predicates. Figure 4.22 shows the set of kept facts. Here, it is shown that the "vertex" predicates were lost in transporting DR2 into the generic format and "node", "net_type", "node_type", "node_dir", and their corresponding "has_"

predicates were lost in transporting from generic into the DR1 format.

The system log indicates that step 8 ("Output Keep") took 3.88 seconds.

```

fromdb(exsch1,dr2,'3.0').
toddb(schemex,dr1,'1.0').
content([keep(vertex(a,r,[10,90],1)),
keep(vertex(a,s,[10,80],1)),
keep(vertex(a,qn,[20,80],1)),
keep(vertex(b,i2,[30,20],1)),
keep(vertex(c,i1,[50,80],1)),
keep(vertex(c,y,[60,70],1)),
keep(vertex(c1,i1,[0,90],1)),
keep(vertex(c1,i2,[0,80],1)),
keep(vertex(c1,i3,[10,20],1)),
keep(vertex(c1,o1,[70,70],1)),
keep(vertex(c1,o2,[60,30],1)),
keep(net_type(nt_7)),keep(net_type(nt_6)),
keep(net_type(nt_5)),keep(net_type(nt_4)),
keep(net_type(nt_3)),keep(net_type(nt_2)),keep(net_type(nt_1)),
keep(node_type(qn)),keep(node_type(q)),keep(node_type(s)),keep(node_type(r)),
keep(node_type(o2)),keep(node_type(o1)),keep(node_type(y)),
keep(node_type(i3)),keep(node_type(i2)),keep(node_type(i1)),
keep(node_dir(ndr_12)),keep(node_dir(ndr_11)),
keep(node_dir(ndr_10)),keep(node_dir(ndr_9)),
keep(node_dir(ndr_8)),keep(node_dir(ndr_7)),
keep(node_dir(ndr_6)),keep(node_dir(ndr_5)),
keep(node_dir(ndr_4)),keep(node_dir(ndr_3)),
keep(node_dir(ndr_2)),keep(node_dir(ndr_1)),
keep(has_(node_12,qn)),keep(has_(node_11,q)),
keep(has_(node_10,s)),keep(has_(node_9,r)),
keep(has_(node_8,o2)),keep(has_(node_7,o1)),
keep(has_(node_6,y)),keep(has_(node_5,i3)),
keep(has_(node_4,i2)),keep(has_(node_3,i2)),
keep(has_(node_2,i1)),keep(has_(node_1,i1)),
keep(has_(node_12,ndr_12)),keep(has_(node_11,ndr_11)),
keep(has_(node_10,ndr_10)),keep(has_(node_9,ndr_9)),
keep(has_(node_8,ndr_8)),keep(has_(node_7,ndr_7)),
keep(has_(node_6,ndr_6)),keep(has_(node_5,ndr_5)),
keep(has_(node_4,ndr_3)),keep(has_(node_3,ndr_4)),
keep(has_(node_2,ndr_1)),keep(has_(node_1,ndr_2)),
keep(has_(s7,nt_7)),keep(has_(s6,nt_6)),
keep(has_(s5,nt_5)),keep(has_(s4,nt_4)),
keep(has_(s3,nt_3)),keep(has_(s2,nt_2)),
keep(has_(s1,nt_1)),keep(has_(conn,node_2)),
keep(has_(conn,node_4)),keep(has_(conn,node_5)),
keep(has_(conn,node_7)),keep(has_(conn,node_8)),
keep(has_(t2,node_1)),keep(has_(t2,node_3)),
keep(has_(t2,node_6)),keep(has_(t1,node_9)),
keep(has_(t1,node_10)),keep(has_(t1,node_11)),
keep(has_(t1,node_12)),dummy]).

```

Figure 4.22. Kept Facts Generated in Transporting Data from DR2 to DR1.

Step 9 is the final clean-up, the Prolog database is reset to eliminate any facts from the last translation. The total time for the translation from DR2 to DR1 was 23.13 seconds.

This simple example illustrates several things which are common in transporting data between any two systems. First, the number of facts in each system is likely to be different. Second, the data model each system uses is likely to be different. This will necessitate that some facts be generated and others be stored as kept facts. Finally, the kept data needs to contain enough information to allow the original data format to be reconstructed. In the next chapter, the reverse translation from DR1 back to DR2 using the kept facts is presented. Then, the DR1-DR2 example complete, two additional test cases are analyzed.

## CHAPTER 5

### TEST CASES

In the preceding chapter, a knowledge-based prototype was described for transporting data between CAE/CAD/CAM systems. A simple example using hypothetical data base formats, DR1 and DR2, illustrated how the prototype operates. In this chapter, the DR1-DR2 example is completed by demonstrating the transport of the data back from DR1 into DR2, utilizing *kept data* to reconstruct the original DR2 data base exactly as it was to start. This example shows how the prototype addresses the *delta problem* described in Chapter 3.

To avoid over-generalizing the significance of the DR1-DR2 example, two additional test cases are analyzed. The first is another example of data belonging to the *logical class* of data described in Chapter 2. The TEGAS Design Language (TDL), a widely recognized schematic network description language, is translated into a Hughes PCB CAD schematic data base and then back again. The second test case transports "physical class" data (i.e., representing an IC layout) between the CALMA GDS II Stream Format and the Cal Tech Intermediate Form (CIF). As in the previous example, the data models between the two systems are not one-to-one. The methodology of using *kept data* is utilized to make the bi-directional transport possible.

## 5.1 Hypothetical Cases: DR1 and DR2

In the previous chapter, DR2 data was translated into DR1 format using a prototype transport system. In the process of creating the DR1 format, *kept data* (Figure 4.22) was created. Since DR1 does not contain all of the data entities that are a part of DR2 (e.g., vertex), it would be impossible to provide these entities without the *kept data*. In Chapter 4, a translate engine process flow was presented which used the *kept data* (see Figure 4.7). Using the steps in this flow diagram, the DR1 data can be translated back into DR2 format. Both the forward and reverse data transport are shown in Figure 5.1.

The Prolog system log for the processing steps of Figure 4.7, is shown in Figure 5.2. As Figure 5.1 indicates, several input files are required. First, the source DBIF and source input rules are read-in in Step 2 (Figure 4.7). ("dr1.dat consulted ..." and "dr1in.rul consulted ..." as noted in Figure 5.2) Figure 5.3 shows the output DR1 DBIF which was produced in Chapter 4. This file ("dr1.dat") will now serve as the input or source for the reverse translation (see Figure 5.1). Figure 5.4 lists the source input rules (dr1in.rul).

Next, Step 2A (Figure 4.7) reads-in the previously kept facts (noted as "dr1K.dat consulted ..." in (Figure 5.2). As shown in Figure 5.1, the *kept data* for this reverse transport example is the same kept data created in Chapter 4. (see Figure 4.22)

Step 3 (Figure 4.7) creates the generic facts, using the source DBIF, kept data, and source input rules. As each predicate in "gencon.rul" (Fig-



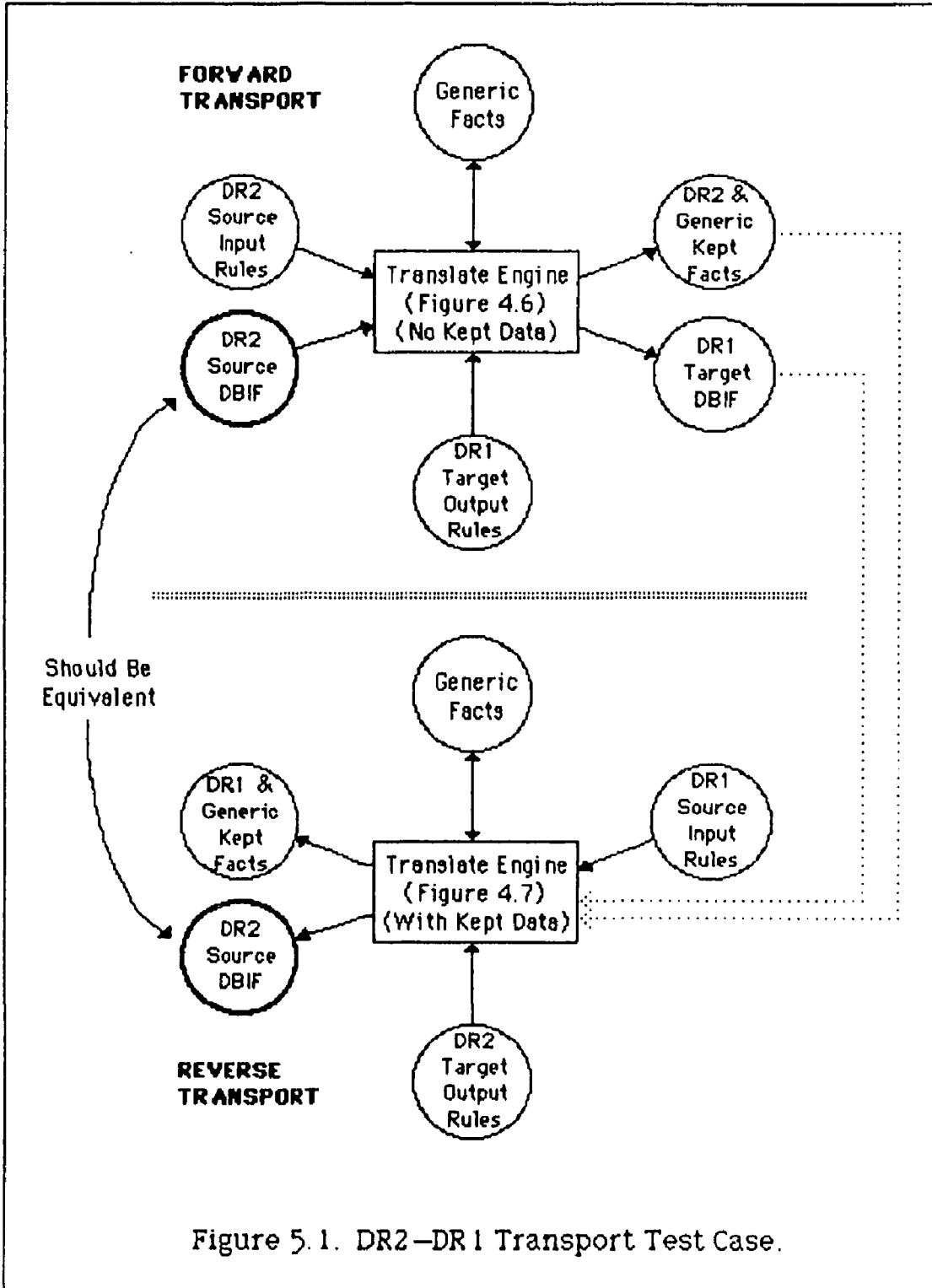


Figure 5.1. DR2-DR1 Transport Test Case.

```

CProlog version 1.4d.edai
| Restoring file tran11.env |

yes
| ?- translate('dr1.dat','dr1K.dat','dr2.out','dr2K.out',exschRT,dr2,'3.0',
               '1/11/84:11:55').
dr1.dat consulted 2248 bytes 0.65 sec.

>> Translate: V1.0 <<

dbid(schemex1,dr1,1.0.1/11/84:17:06)
dr1in.rul consulted 2184 bytes 0.7 sec.
dr1K.dat consulted 3648 bytes 0.96667 sec.
fromdb(exsch1,dr2,3.0)
todb(schemex,dr1,1.0)
Start Up 14.03 sec.
  T=node(_1781) 12 facts.
  T=net(_1781) 7 facts.
  T=box(_1781) 4 facts.
  T=box_type(_1781) 3 facts.
  T=net_type(_1781) 7 facts.
  T=node_type(_1781) 10 facts.
  T=node_dir(_1781) 12 facts.
  T=macro_call(_1781) 0 facts.
  T=macro_def(_1781) 0 facts.
  T=connected(_1781,_1782,_1783) 15 facts.
  T=has(_1781,_1782) 47 facts.
  T=end_of_file 0 facts.
117 facts total.
Generic 7.87 sec.
Keep 9.88 sec.
Unload 1.35 sec.
dr2out.rul consulted 1748 bytes 0.61668 sec.
  T=signal(_4599) 7 facts.
  T=ndt(_4599) 10 facts.
  T=bt(_4599) 3 facts.
  T=body(_4599) 4 facts.
  T=vertex(_4599,_4600,_4601,_4602) 15 facts.
  T=vertex(_4599,_4600,_4601) 15 facts.
  T=has(_4599,_4600) 16 facts.
  T=end_of_file 0 facts.
70 facts total.
dbid(exschRT,dr2,3.0,1/11/84:11:55)
Phase 2 18.83 sec.
Output Keep 14.6 sec.

Total time is 67.87 sec.

yes
| ?- halt.
| Prolog execution halted |

```

Figure 5.2. Prolog Output for DR2 to DR1 Translation.

ure 4.12) is read-in, it is echoed out as before with the number of facts which were generated. The rules which are contained in "dr1in.rul" (Figure 5.4) are used to derive the generic facts from the source DBIF and the kept facts.

```

dbid(exschT,dr1,'10.0','1/10/84:14:53').
content([signal(s7),
signal(s6),          connect(s7,c1,o2),
signal(s5),          connect(s7,b,y),
signal(s4),          connect(s6,c1,o1),
signal(s3),          connect(s6,c,y),
signal(s2),          connect(s5,b,i1),
signal(s1),          connect(s5,c,i2),
pin(qn),             connect(s5,a,qn),
pin(q),              connect(s4,c,i1),
pin(s),              connect(s4,a,q),
pin(r),              connect(s3,b,i2),
pin(o2),             connect(s3,c1,i3),
pin(o1),             connect(s2,a,s),
pin(y),              connect(s2,c1,i2),
pin(i3),             connect(s1,a,r),
pin(i2),             connect(s1,c1,i1),
pin(i1),             has(c1,conn),
comp_name(c1),       has(c,t2),
comp_name(c),        has(b,t2),
comp_name(b),        has(a,t1),
comp_name(a),        dummy]),
comp_type(conn),
comp_type(t2),
comp_type(t1),

```

Figure 5.3. Source DR1 DBIF (dr1.dat).

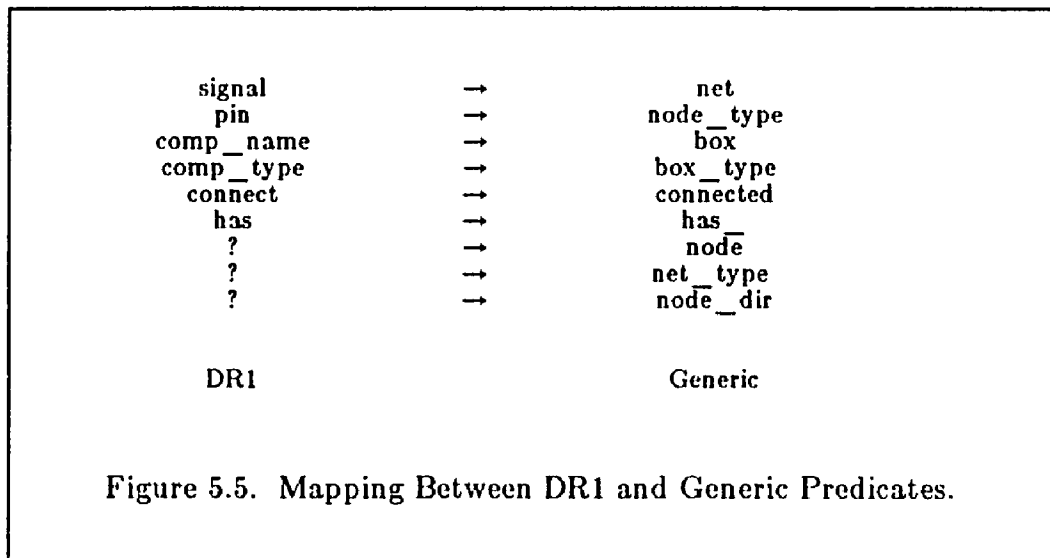
```

net(X):-signal(X).
node_type(X):-pin(X).
box(X):-comp_name(X).
connected(C,S,N):-keep(has_(N,P)),connect(S,C,P),pin(P),has(C,T),
keep(has_(T,N)).
connected(C,S,N):-!,connect(S,C,P),pin(P),\+keep(has_(N,P)),
has(C,B),comp_type(B),has_(B,N),node(N),has_(N,P).
node(X):-var(X),pin(Y),keep(has_(X,Y)).
node(X):-var(X),pin(P),\+keep(has_(X,P)),connect(S,C,P),has(C,B),
comp_type(B),asst_uniq(nd(B,P)),fail.
node(X):-var(X),retract(nd(B,P)),gensym(node_ X),asserta(node(X)),
asserta(has_(X,P)),asserta(has_(B,X)).
box_type(X):-comp_type(X).
net_type(X):-var(X),signal(S),keep(has_(S,X)).
net_type(X):-var(X),signal(S),\+keep(has_(S,X)),
gensym(net_ X),asserta(has_(S,X)).
node_dir(X):-var(X),keep(node_dir(X)).
node_dir(X):-var(X),pin(P),has_(Y,P),node(Y),gensym(ndr_ X),
assert(has_(Y,X)).
has_(X,Y):-var(X),var(Y),comp_name(X),has(X,Y),comp_type(Y),
\+keep(has_(X,Y)).
has_(X,Y):-var(X),var(Y),keep(has_(X,Y)).

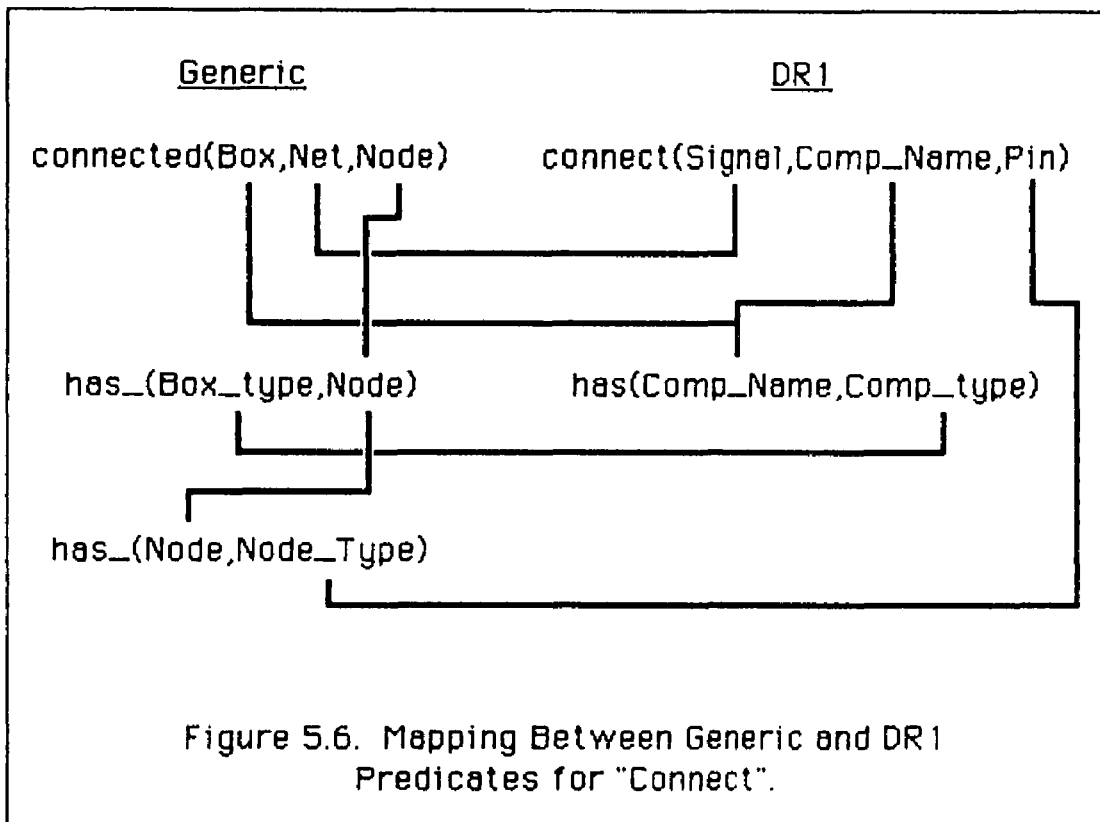
```

Figure 5.4. Source Input Rules for DR1 (dr1in.rul).

A closer look at these source input rules (Figure 5.4) indicate how DR1 predicates are mapped onto generic predicates. DR1 consists of the following predicates: signal, pin, comp_name, comp_type, connect, and has. The generic predicates consist of net, node, box, net_type, node_type, box_type, node_dir, connected, and has_. Figure 5.5 shows the mapping between these sets of predicates. For predicates which have a mapping between the two representations, the translate rule is usually simple. (See the rule which translates "signal" to "net" in Figure 5.5)



In some cases, there is a slight difference in meaning between the DR1 and generic predicates which map onto one another, and this is reflected in the rule for translation. Consider the second rule for "connected" in Figure 5.4. Figure 5.6 illustrates this rule and shows the relationships and the mapping between DR1 and generic predicates. In the case of the DR1 predicate "connect", a "signal" is connected to the "pin" of a "comp_name". However, the generic counterpart, "connected", connects



a "net" (S in Figure 5.4) to the "node" (N) of a "box_type" (B) which is the type of "box" (C). In this case, there is no "node" in DR1, but rather a "pin" instead, which corresponds to a generic "node_type". The mapping between "node" and pin is established by the translation rule for "node" (explained below), before the "connected" predicate is processed.

Another type of rule which is somewhat complicated, is that for the generic predicates in Figure 5.5 which do not have counterparts in DR1. Consider the rule for "node". Figure 4.17 showed that in the generic model, for every "box_type" there are one or more "nodes", each of which has a single "node_type". In the DR1 model (refer to Figure 5.7), there is no counterpart for "node". The closest thing to a node is a "pin", which really corresponds to a "node_type". The first rule for "node",

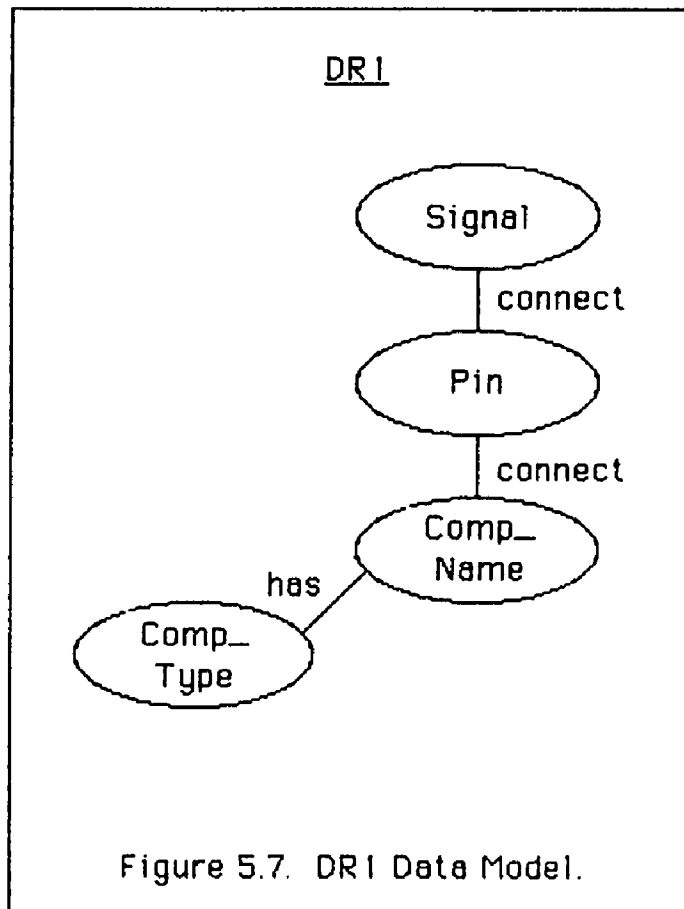
```
"node(X):-var(X),pin(Y),keep(has_(X,Y))."
```

says that

if "Y" is a "pin" (i.e., generic "node_type") and there is a kept fact that "has_(X,Y)", then "X" must be a "node".

This is because there is only one entity in the generic model which "has" a "node_type" (see Figure 4.17).

The second and third rules for "node" (Figure 5.4) work together in the event that no kept facts exist for "pins". The second rule searches the Prolog database, looking for all "pin" facts. Since the same pin name could be used in different component types and there also may be more than one instance of a given component type, the second rule finds all unique component type-pin pairs. As each unique pair is found, the fact is entered into the Prolog database as "nd(Body,Pin)". The "fail" clause at the end of the second rule causes backtracking to occur until there are no more



pins. At this point, the third rule begins. For each occurrence of "nd(B,P)", a unique node name is generated. The unique name is asserted as a node name, it is asserted that the node name has node type "P" (the pin name from DR1), and it is asserted that component type "B" has node name "X" (the newly generated unique name).

The same technique is applied to generate other data required in the generic model, but not present in DR1: net__type and node__dir. Once each of the generic predicates has been translated from DR1, Step 3 is complete and the file of generic facts is available for further processing.

Step 4 (Figure 4.7) creates any kept facts that would be lost in translating from the source DBIF to generic facts. In this example, all data represented in DR1 can be represented in terms of generic predicates. So, there are no rules in "dr1in.rul", the source translate rules, for the predicate "keep". And, no kept facts are written out into the new kept data file (dr2K.out).

Step 5 consists of resetting the fact/rule data base. In the system log of Figure 5.2, this event is marked "Unload ... sec." Next, Step 6 reads in the generic facts that were created in Step 3 and also reads in the target translate rules (dr2out.rul, Figure 5.8).

In Step 7 we generate target DBIF from the generic facts. To do this, the target predicates are read-in, one at a time, and rules for each predicate are applied to perform the translation. As was shown in Figure 4.17, all of DR2 has a counterpart in the generic model, except for the notion of a vertex. However, there are some generic predicates which have no



```

signal(X):-net(X).
body(X):-box(X).
vertex(X,V1,I):-connected(Z,X,N),has_(N,Nt),vertex(Z,Nt,V1,I),genvno(X,I).
genvno(Z,I):-maxsval(Z,J),I is J+1,retract(maxsval(Z,J)),
             asserta(maxsval(Z,I)).
genvno(Z,1):-asserta(maxsval(Z,1)).
vertex(W,X,Y,Z):-keep(vertex(W,X,Y,Z)).
vertex(B,Nt,[X,Y],I):-\+keep(vertex(B,Nt,V,I)),node_type(Nt),
                    box(B),has_(B,Bt),has_(Bt,N),has_(N,Nt),
                    gensym(coord,X),gensym(coord,Y),asserta(vertex(B,Nt,[X,Y],I)).

ndt(X):-node_type(X).
bt(X):-box_type(X).

has(X,Y):-box_type(X),has_(X,Z),node(Z),has_(Z,Y),node_type(Y).
has(X,Y):-box(X),has_(X,Y),box_type(Y).

keep(node(X)):-node(X).
keep(has_(X,Y)):-node(X),has_(X,Y),node_type(Y).
keep(has_(X,Y)):-node(X),has_(X,Y),node_dir(Y).
keep(has_(X,Y)):-net(X),has_(X,Y),net_type(Y).
keep(has_(X,Y)):-box_type(X),has_(X,Y),node(Y).

```

Figure 5.8. Target Output Rules for DR2 (dr2out.rul).

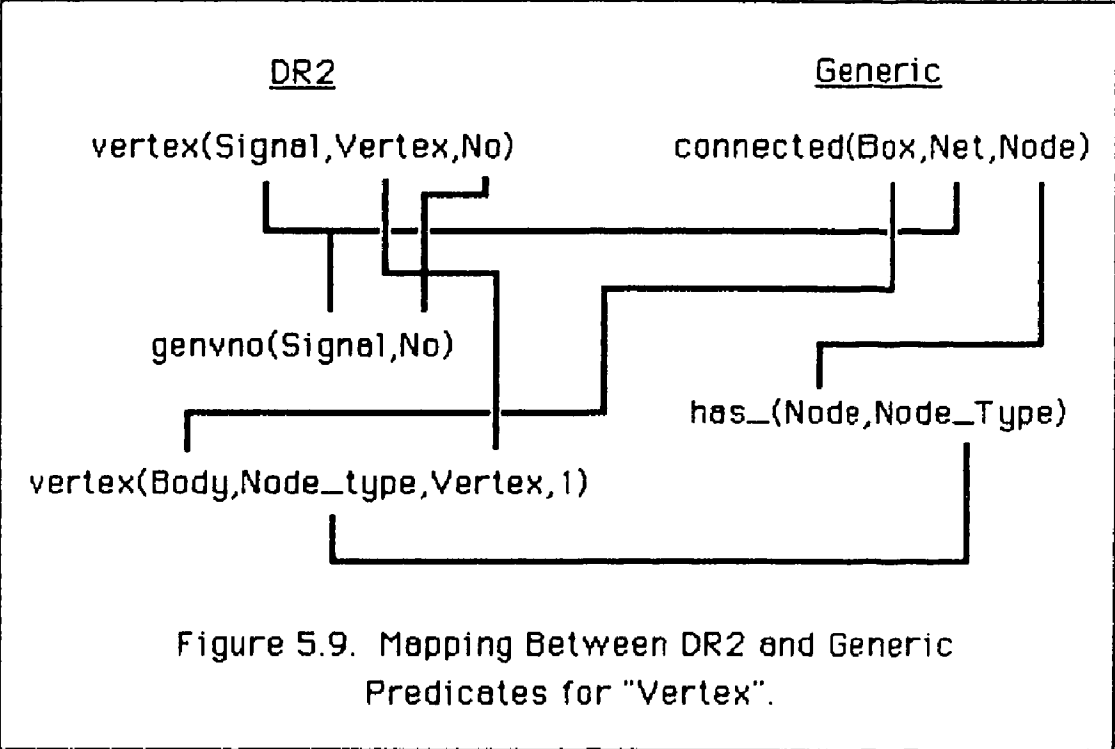
counterpart in DR2. This latter data will be stored away in Step 8. As Figure 4.17 shows, the predicates "box", "box_type", net, and "node_type" are easily translated into DR2. This is reflected in the simple target translate rules for "body", "bt", "signal", and "ndt" in Figure 5.8.

The DR2 predicate "has" corresponds to a subset of the generic "has_" instances. The first "has" rule in Figure 5.8 corresponds to the relationship between "bt" and "ndt". This relationship exists for every occurrence of "has_(Box_type,Node),has_(Node,Node_type)" among the generic facts. The second "has" rule in Figure 5.8 says that for every "box" which has "box_type" then the corresponding "body" has the corresponding "bt". The remaining generic "has_" facts are not translated into the target (DR2) DBIF.

The most complicated target output rules for DR2 are those for "vertex". There are actually two "vertex" predicates: one for signals which has 3 arguments and one for node__types which has 4 arguments. The first vertex rules processed are those with 4 arguments. These are used to create vertices for nodes. After the vertices for nodes are processed, then the vertex rules with 3 arguments are processed for signals.

The first vertex rule with 4 arguments in Figure 5.8 checks to see if the vertices for the node were saved from a previous translation. This assumes that we are performing a reverse translation, which is the case in this example. The second vertex rule with 4 arguments assumes that there is no previous information stored about vertices, which would be the case if this were the original translation from DR1 to DR2. In this case, the vertex for a body-node__type combination is generated. Two symbolic coordinates (one for X and one for Y) are generated and assigned to the node. There is only one vertex for a node, so by definition the fourth argument is 1. After the target DBIF is created, the symbolic coordinates will need to be assigned actual values in order to draw the schematic diagram represented by the DR2 database.

After the 4 argument vertex rules, the 3 argument vertex rules are processed for signals. The only "vertex" rule in Figure 5.8 used to generate a vertex for a signal assumes that the signal is connected to a node having a known vertex. This is illustrated in Figure 5.9. The last predicate in this rule, "getvno", is used to generate a series of vertex numbers for signals, since there are at least two vertices for most legitimate signals.



After all of the DR2 predicates are processed in Step 7 (Figure 4.7), the target DBIF is written out (see Figure 5.10). This database is equivalent to the original source DBIF that was shown in Figure 4.14 (except for order), indicating that no data was lost in the translation from DR2 to DR1 and back again.

```

dbid(exschRT,dr2,'3.0','1/11/84:11:55').
content({signal(s1),      signal(s2),
signal(s3),      signal(s4),      signal(s5),
signal(s6),      signal(s7),
ndt(i1), ndt(r), ndt(i2), ndt(s),
ndt(i3), ndt(q), ndt(qn),ndt(y),
ndt(o1),ndt(o2),
bt(conn),      bt(t1), bt(t2),
body(c1),      body(a),      body(b),      body(c),
vertex(a,r,[10,90],1), vertex(a,s,[10,80],1),
vertex(a,q,[20,90],1), vertex(a,qn,[20,80],1),
vertex(b,i1,[30,40],1), vertex(b,i2,[30,20],1),
vertex(b,y,[40,30],1), vertex(c,i1,[50,80],1),
vertex(c,i2,[50,60],1), vertex(c,y,[60,70],1),
vertex(c1,i1,[0,90],1), vertex(c1,i2,[0,80],1),
vertex(c1,i3,[10,20],1), vertex(c1,o1,[70,70],1),
vertex(c1,o2,[60,30],1), vertex(s4,[50,80],1),
vertex(s5,[30,40],1), vertex(s1,[0,90],1),
vertex(s3,[30,20],1), vertex(s5,[50,60],2),
vertex(s2,[0,80],1), vertex(s3,[10,20],2),
vertex(s6,[60,70],1), vertex(s7,[40,30],1),
vertex(s6,[70,70],2), vertex(s7,[60,30],2),
vertex(s1,[10,90],2), vertex(s2,[10,80],2),
vertex(s4,[20,90],2), vertex(s5,[20,80],3),
has(conn,o2), has(conn,o1), has(conn,i3),
has(conn,i2), has(conn,i1), has(t1,qn),
has(t1,q), has(t1,s), has(t1,r), has(t2,y),
has(t2,i2), has(t2,i1), has(c1,conn),
has(a,t1), has(b,t2), has(c,t2),
dummy}).

```

Figure 5.10 Target DBIF for DR2.

Next any rules for kept data are processed in Step 8 (Figure 4.7). There are several "keep" rules in Figure 5.8, which indicate that there is generic data which cannot be expressed in DR2. Any "node" facts are kept along with 4 types of "has_" facts: node-node_type, node-node_dir,

net-net_type, and box_type-node. These facts are stored in "dr2K.out", the fourth argument of the "translate" call in the system log (Figure 5.2). These facts are shown in Figure 5.11.

```

fromdb(schemex1,dr1,'1.0').toddb(exschRT,dr2,'3.0').
content((keep(has_(t2,node_1)),      keep(has_(t2,node_3)),
keep(has_(t2,node_6)),      keep(has_(t1,node_9)),
keep(has_(t1,node_10)),     keep(has_(t1,node_11)),
keep(has_(t1,node_12)),     keep(has_(conn,node_2)),
keep(has_(conn,node_4)),    keep(has_(conn,node_5)),
keep(has_(conn,node_7)),    keep(has_(conn,node_8)),
keep(has_(s7,nt_7)),      keep(has_(s6,nt_6)),
keep(has_(s5,nt_5)),      keep(has_(s4,nt_4)),
keep(has_(s3,nt_3)),      keep(has_(s2,nt_2)),
keep(has_(s1,nt_1)),      keep(has_(node_8,ndr_8)),
keep(has_(node_7,ndr_7)),  keep(has_(node_6,ndr_6)),
keep(has_(node_12,ndr_12)), keep(has_(node_11,ndr_11)),
keep(has_(node_5,ndr_5)),  keep(has_(node_10,ndr_10)),
keep(has_(node_4,ndr_3)),  keep(has_(node_3,ndr_4)),
keep(has_(node_9,ndr_9)),  keep(has_(node_2,ndr_1)),
keep(has_(node_1,ndr_2)),  keep(has_(node_8,o2)),
keep(has_(node_7,o1)),     keep(has_(node_6,y)),
keep(has_(node_12,qn)),   keep(has_(node_11,q)),
keep(has_(node_5,i3)),    keep(has_(node_10,s)),
keep(has_(node_4,i2)),    keep(has_(node_3,i2)),
keep(has_(node_9,r)),     keep(has_(node_2,i1)),
keep(has_(node_1,i1)),    keep(node(node_8)),
keep(node(node_7)),       keep(node(node_6)),
keep(node(node_12)),     keep(node(node_11)),
keep(node(node_5)),      keep(node(node_10)),
keep(node(node_4)),      keep(node(node_3)),
keep(node(node_9)),      keep(node(node_2)),
keep(node(node_1)),
dummy}).

```

Figure 5.11. Kept Data from the Generic to DR2 Translation.

The final step just resets the Prolog database and prints out the total CPU time taken for the translation.

This example is now complete. Several of the techniques required for handling the mis-matches between data representations have been presented. The remainder of the chapter describes additional test cases which are more complex and show additional methods for defining transla-

tion rules. The knowledge-based prototype steps are the same as used in this example and shown in Figures 4.6 and 4.7.

## 5.2 TDL to PCB CAD Data Base

This second example concerns the transport of a TDL database to the Hughes PCB CAD (HPC) system. The TDL language, described in section 4.2.1, is used to describe schematic networks for logic simulation. The HPC system data base is used to describe schematic networks for PC board routing. While there is data in common, not all TDL entities are described in the HPC system and vice versa. Figure 5.12 shows the source TDL file in native format. The portion of the file of interest is in the "DEFINE" section, which describes the network. Figure 5.13 shows the schematic diagram corresponding to this network. Using the TDL compiler described in section 4.2.1, a DBIF representation of the TDL was produced as shown in Figure 5.14. A few syntactical changes were made to the native data such as converting TDL names (e.g., NAND-A) into legitimate Prolog constants ("nand__a"). At the expense of additional coding, these changes could be eliminated. Figure 5.15 shows the TDL data model pictorially.

The HPC native data format is a relational data table quite similar to DR1. Figure 2.7 shows the list of entities in the HPC data base. Since this data base has the combined function of representing logical and physical classes of data, only a portion of the entities were used in this example. These correspond to the DR1 data entities, and Figure 5.16 illustrates this HPC data model.

```

COMPILE;
OPTIONS CATALOG, XREF;
DIRECTORY RPH;
MODULE JKFF/GATE/1/RPH;
INPUTS CLOCK, J, K, PS, PC;
OUTPUTS OQ, OQB;
DESCRIPTION THE MODULE IS A MASTER/SLAVE JK FLIP-FLOP
          WITH PRESET AND PRECLEAR LINES.  ;
          "(SEE TDL REF. MANUAL P.71)"
DELAYS NANDEL/3,2,4/, NOT/3,2,4/;

"THE FOLLOWING TWO LINES CREATE TWO DIFFERENT TYPES
BASED ON THE PRIMITIVE ELEMENT NAND.
3-NAND IS THE SAME AS NAND.
2-NAND IS DECLARED TO BE A 2 INPUT NAND. "

USE 3-NAND = NAND(3,1) /NANDEL/,
    2-NAND = NAND(2,1) /NANDEL/;

DEFINE
DEV1(NAND-A) = 3-NAND(J,QB,CLOCK);
DEV2(NAND-B) = 3-NAND(K,Q,CLOCK);
DEV3(NAND-C) = NAND(PS,NAND-A,NAND-D);
DEV4(NAND-D) = NAND(PC,NAND-B,NAND-C);
DEV5(I) = NOT(CLOCK);
DEV6(NAND-E) = 2-NAND(NAND-C,I);
DEV7(NAND-F) = 2-NAND(NAND-D,I);
G-NAND(Q) = NAND(NAND-E,QB);
H-NAND(QB) = NAND(NAND-F,Q);
DEV8(OQ) = NOT(Q);
DEV9(OQB /1/) = NOT(QB);
END MODULE;
END COMPILE;

```

Figure 5.12. Source TDL File in Native Format.

### 5.2.1 Data Mapping

The first step in transporting data between two formats is defining the mapping between their entities. From the mapping comes the rules which are used to transport the databases. An example was shown in Figure 4.17 for the mapping between DR2 and the generic form. In this example, mappings are needed between TDL and the generic format (Figure 5.17) and between the generic format and the HPC (DR1) format (Figure

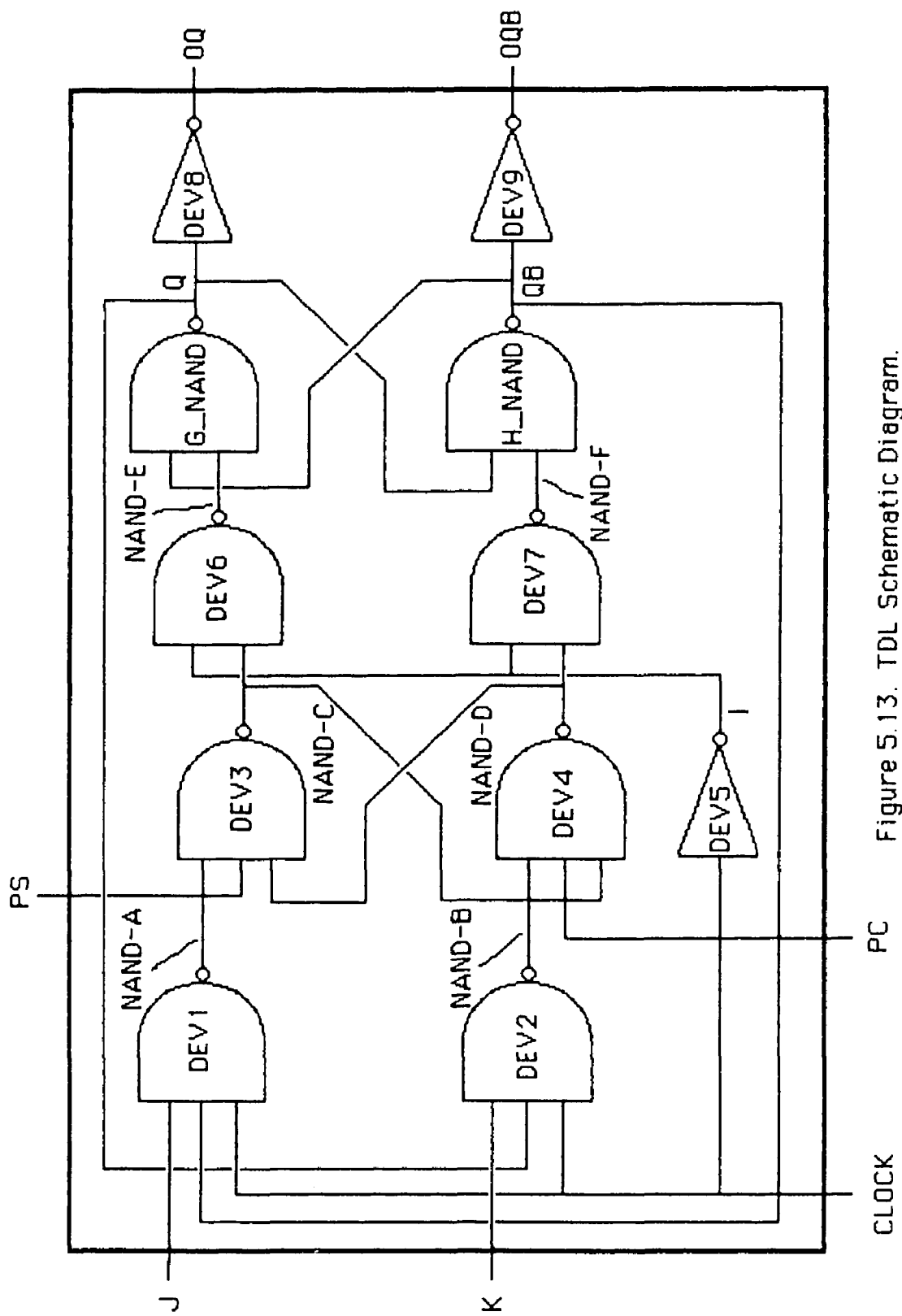


Figure 5.13. TDL Schematic Diagram.



```

dbid('jkkf',tdl,'1','2/21/84:10.05').
content([descript_class(gate),
dir_name('rph'),
pin_dir(in),
has('clock',in),
has('k',in),
has('pc',in),
has('oq',out),
desc('THE MODULE IS A MASTER / SLAVE JK FLIP-FLOP WITH PRESET
AND PRECLEAR LINES. ; '),
delay('nandel',3,2,4,'/'),
use('dig__3__nand',='nand',3,1,none,'nandel'),
use('dig__2__nand',='nand',2,1,none,'nandel'),
occ_name('dev1'),
has('dev1',dig__3__nand'),
connect('dev1','qb',in,'2'),
occ_name('dev2'),
has('dev2',dig__3__nand'),
connect('dev2','q',in,'2'),
occ_name('dev3'),
has('dev3','nand'),
connect('dev3','nand__a',in,'2'),
occ_name('dev4'),
has('dev4','nand'),
connect('dev4','nand__b',in,'2'),
occ_name('dev5'),
has('dev5','not'),
occ_name('dev6'),
has('dev6',dig__2__nand'),
connect('dev6','i',in,'2'),
occ_name('dev7'),
has('dev7',dig__2__nand'),
connect('dev7','i',in,'2'),
occ_name('g__nand'),
has('g__nand','nand'),
connect('g__nand','qb',in,'2'),
occ_name('h__nand'),
has('h__nand','nand'),
connect('h__nand','q',in,'2'),
occ_name('dev8'),
has('dev8','not'),
occ_name('dev9'),
connect('dev9','oqb',out,'1'),
connect('dev9','qb',in,'1'),
pin('oqb'),
pin('pc'),
pin('k'),
pin('clock'),
signal('qb'),
signal('nand__f'),
signal('i'),
signal('nand__c'),
signal('nand__a'),
device('not'),
device('dig__2__nand'),
ext_out_pin('oqb'),
ext_in_pin('pc'),
ext_in_pin('k'),
ext_in_pin('clock'),
has('j',in),
has('ps',in),
pin_dir(out),
has('oqb',out),
delay('not',3,2,4,'/'),
connect('dev1','nand__a',out,'1'),
connect('dev1','j',in,'1'),
connect('dev1','clock',in,'3'),
connect('dev2','nand__b',out,'1'),
connect('dev2','k',in,'1'),
connect('dev2','clock',in,'3'),
connect('dev3','nand__c',out,'1'),
connect('dev3','ps',in,'1'),
connect('dev3','nand__d',in,'3'),
connect('dev4','nand__d',out,'1'),
connect('dev4','pc',in,'1'),
connect('dev4','nand__c',in,'3'),
connect('dev5','i',out,'1'),
connect('dev5','clock',in,'1'),
connect('dev6','nand__e',out,'1'),
connect('dev6','nand__c',in,'1'),
connect('dev7','nand__f',out,'1'),
connect('dev7','nand__d',in,'1'),
connect('g__nand','q',out,'1'),
connect('g__nand','nand__e',in,'1'),
connect('h__nand','qb',out,'1'),
connect('h__nand','nand__f',in,'1'),
connect('dev8','oq',out,'1'),
connect('dev8','q',in,'1'),
delay('dev9',1,1,'/'),
has('dev9','not'),
pin('oq'),
pin('ps'),
pin('j'),
signal('q'),
signal('nand__c'),
signal('nand__d'),
signal('nand__b'),
device('nand'),
device('dig__3__nand'),
ext_out_pin('oq'),
ext_in_pin('ps'),
ext_in_pin('j'),
dummy]].

```

Figure 5.14. Source TDL DBIF.

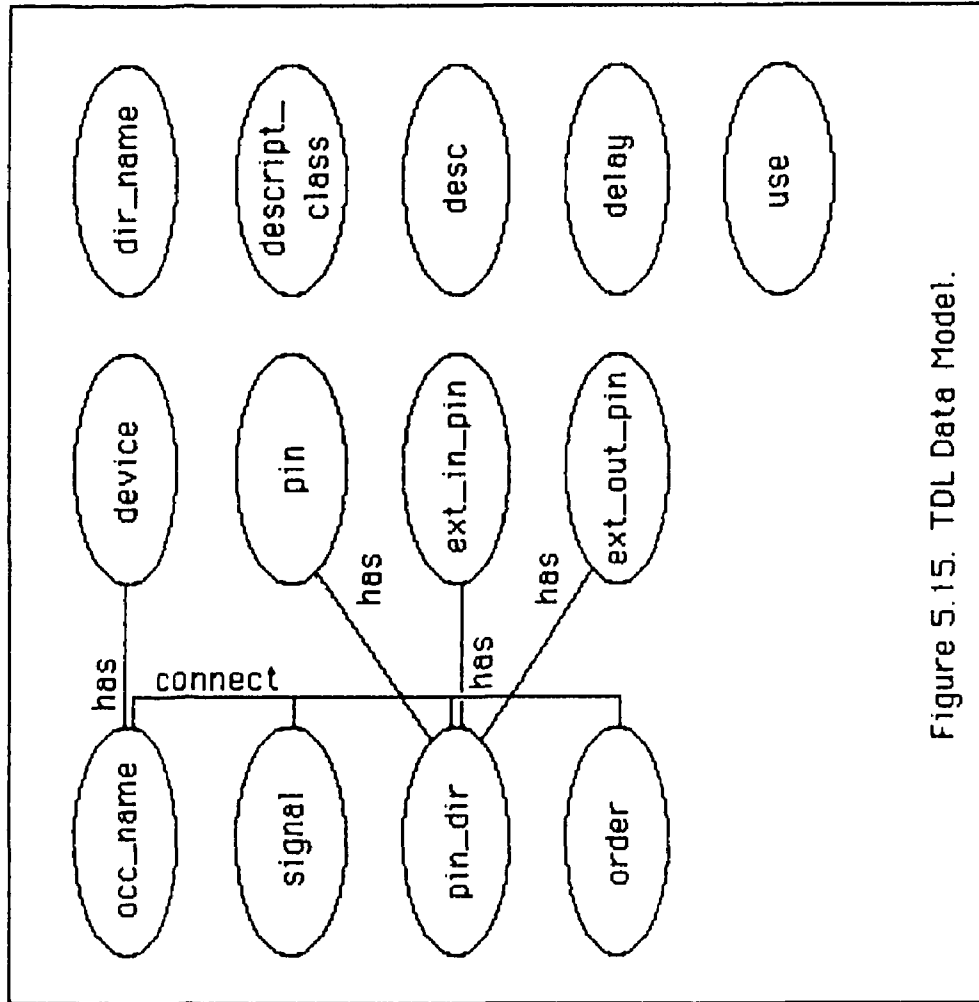
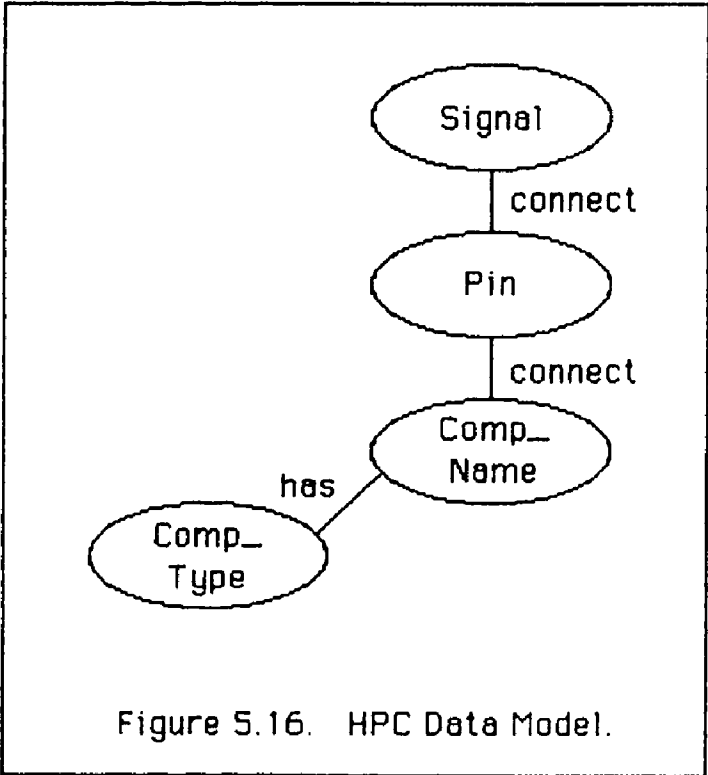


Figure 5.15. TDL Data Model.



5.18). From the TDL to generic mapping it is clear that some TDL information must be kept, since there is no generic counterpart (e.g., "order"). Likewise, there are generic predicates without TDL counterparts. These must be generated during the translation (e.g., "node").

### 5.2.2 Forward Rules

With an understanding of the data models and mappings, rules can be written for translation. The rules for transporting TDL into the translate engine are shown in Figure 5.19. The TDL predicates with generic counterparts are relatively straight-forward (i.e., "box", "box_type", "net", "node_type", "node_dir", "has", and "connected", see Figure 5.17). One complication to these rules is required to handle TDL external pins, since there is no name given to the signals which connect to these pins, and there is no device to which these external pins are assigned in the TDL model. This is shown pictorally in the schematic diagram of Figure 5.13. Consequently, the translation of "box", "box_type", "net", "node_type", and "node_dir" is more than just a series of one-to-one mappings. The remainder of this section describes the rules for translating TDL into the generic form.

Rules 1-9 (Figure 5.19) generate node names. Rules 2 and 3 create nodes for external pins. A unique node name is generated and it is asserted that device "exterior" "has" the node and that the node has the pin, which becomes the node_type. Rule 4 looks for unique occurrences of each device type. Each pin connected to a signal is associated with its appropriate device. Since devices may be used more than once, the pin name, device pair must be accounted for once. Since pin names are not always

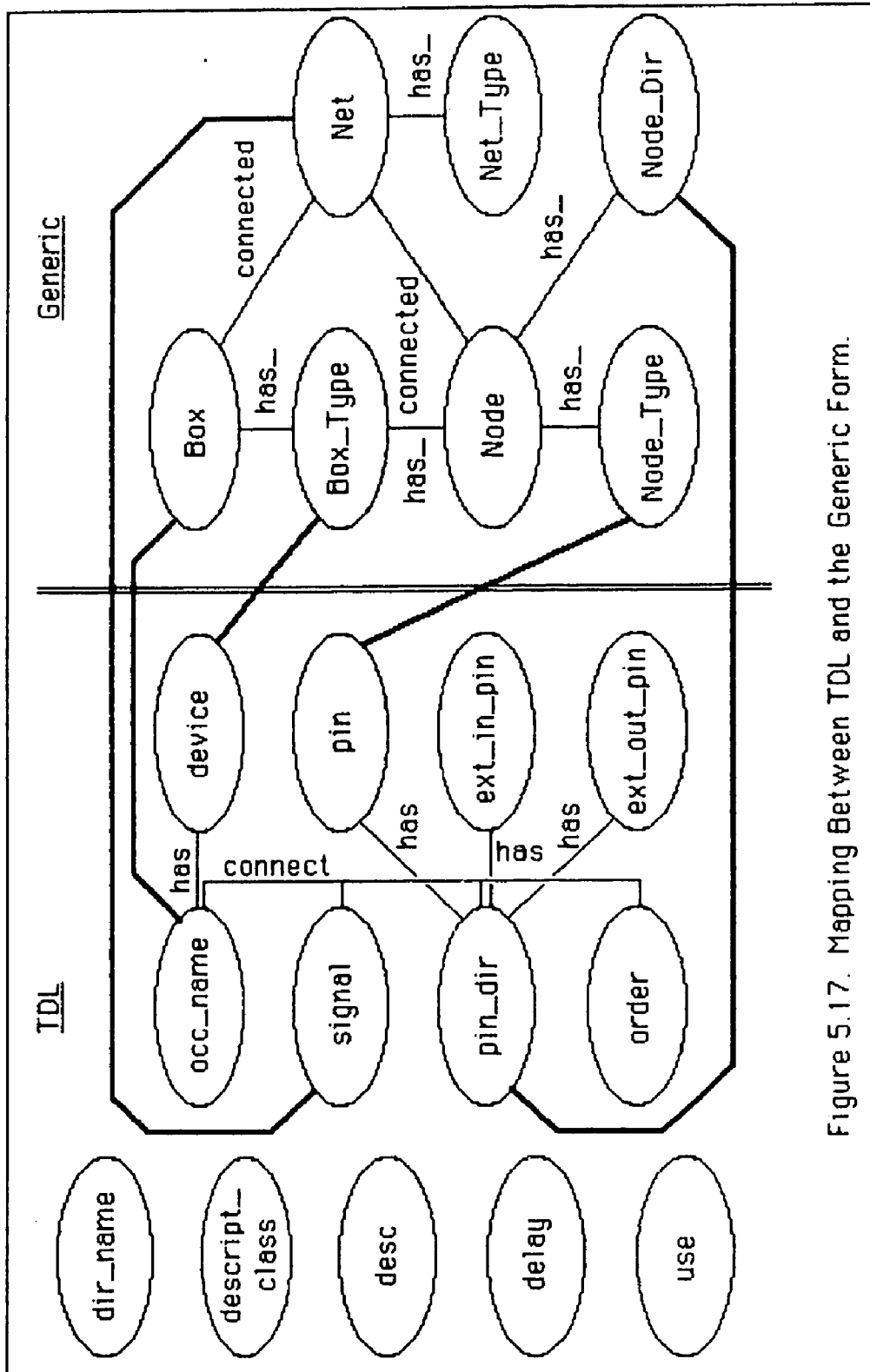


Figure 5.17. Mapping Between TDL and the Generic Form.

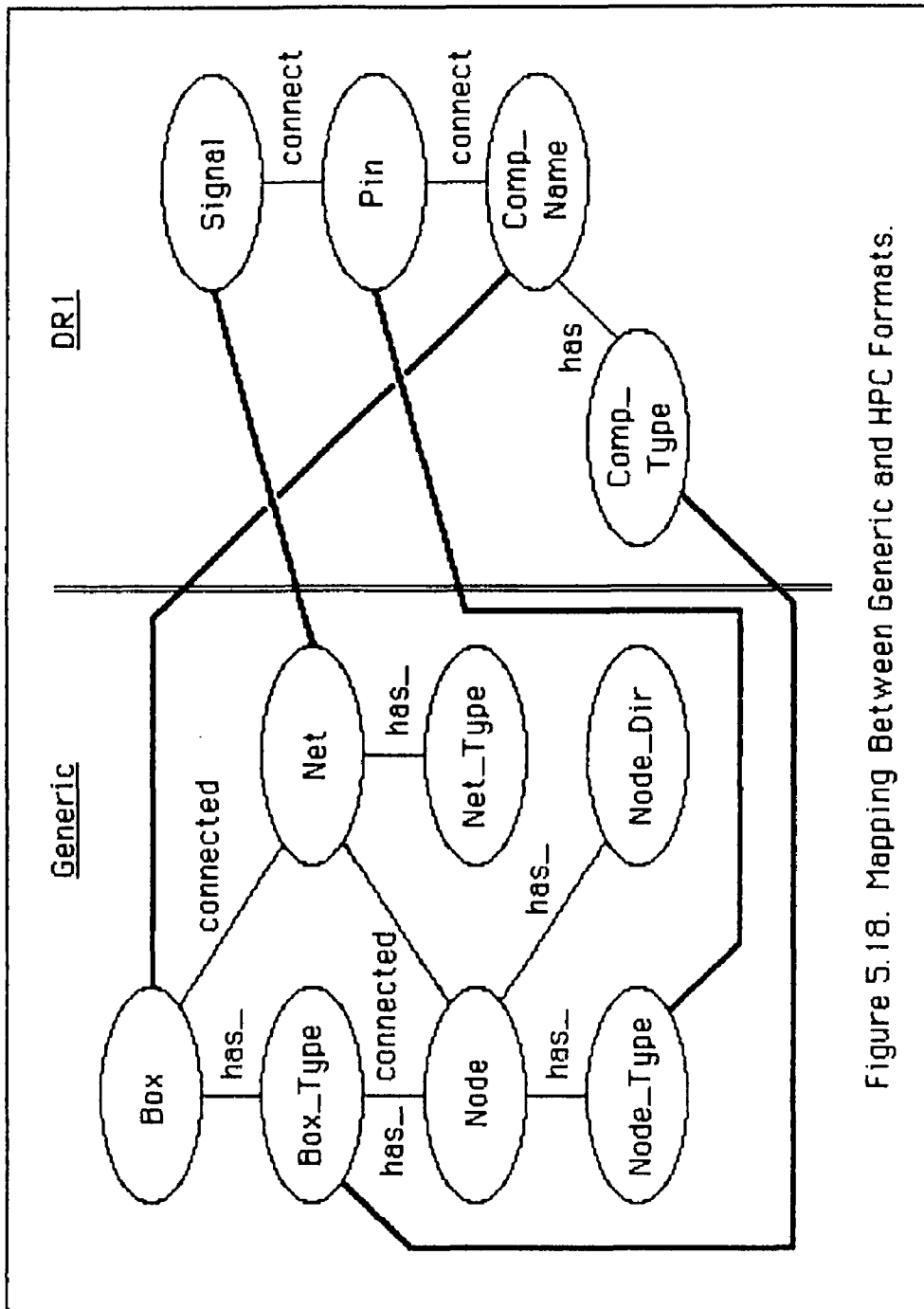


Figure 5.18. Mapping Between Generic and HPC Formats.

```

/* 1 */ node(X):-var(X),nd2(X).
/* 2 */ nd2(X):-ext_in_pin(Nt),nd3(X,exterior,Nt).
/* 3 */ nd2(X):-ext_out_pin(Nt),nd3(X,exterior,Nt).
/* 4 */ node(X):-var(X),device(T),has(B,T),connect(B,S,D,N),nd4(D,N,R),
      asst_uniq(ndsv(T,R)),fail.
/* 5 */ node(X):-var(X),retract(ndsv(T,N)),nd3(X,T,N).
/* 6 */ nd3(X,T,Nt):-gensym(node_,X),asserta(node(X)),asserta(has_(X,Nt)),
      asserta(has_(T,X)).
/* 7 */ nd4(D,N,R):-nd5(N),concat(D,N,R).
/* 8 */ nd4(D,N,N):-\+nd5(N).
/* 9 */ nd5(N):-name(N,[H|T]),H>48,H<58,T==|.

/* 10 */ net(X):-signal(X).
/* 11 */ net(X):-var(X),ext_out_pin(S),net2(S,B,X,D,N),asserta(connect(ext,X,out,S)).
/* 12 */ net(X):-var(X),ext_in_pin(S),net2(S,B,X,D,N),asserta(connect(ext,X,in,S)).
/* 13 */ net2(S,B,X,D,N):-connect(B,S,D,N),gensym(net_,X),asserta(signal(X)),
      asserta(connect(B,X,D,N)).

/* 14 */ box(X):-occ_name(X).
/* 15 */ box(ext):-ext_pin(_,!).

/* 16 */ box_type(X):-device(X).
/* 17 */ box_type(exterior):-ext_pin(_,!).

/* 18 */ net_type(X):-var(X),keep(net_type(X)).
/* 19 */ net_type(X):-var(X),signal(Y),gensym(nt_,X),asserta(has_(Y,X)).

/* 20 */ node_type(N):-var(N),pin(N),asst_uniq(n_t(N)),fail.
/* 21 */ node_type(N):-var(N),connect(B,S,D,R),nd4(D,R,N),asst_uniq(n_t(N)),fail.
/* 22 */ node_type(N):-var(N),retract(n_t(N)),asserta(node_type(N)).

/* 23 */ node_dir(X):-pin_dir(X),asst_uniq(n_d(X)),fail.
/* 24 */ node_dir(X):-connect(B,S,X,N),\+pin_dir(X),asst_uniq(n_d(X)),fail.
/* 25 */ node_dir(X):-retract(n_d(X)).

/* 26 */ connected(B,S,X):-connect(B,S,D,N),signal(S),nd4(D,N,R),node_type(R),
      has(B,T),device(T),has_(T,X),node(X),has_(X,R).
/* 27 */ connected(ext,S,X):-connect(ext,S,D,N),signal(S),node_type(N),has_(X,N),
      node(X),has_(exterior,X).

/* 28 */ has_(X,Y):-has(X,Y),occ_name(X),device(Y).
/* 29 */ has_(N,ND):-var(ND),var(N),has(NT,ND),pin_dir(ND),has_(N,NT),
      asst_uniq(hx(N,ND)),fail.
/* 30 */ has_(N,ND):-var(ND),var(N),connect(B,S,ND,NT),nd4(ND,NT,R),has_(N,R),node(N),
      asst_uniq(hx(N,ND)),fail.
/* 31 */ has_(N,ND):-retract(hx(N,ND)).
/* 32 */ has_(ext,exterior):-ext_pin(_).

/* 33 */ ext_pin(X):-ext_in_pin(X),!.
/* 34 */ ext_pin(X):-ext_out_pin(X),!.

/* 35 */ asst_uniq(X):-retr(X),asserta(X).

/* 36 */ keep(dir_name(X)):-dir_name(X).
/* 37 */ keep(descript_class(X)):-descript_class(X).
/* 38 */ keep(desc(X)):-desc(X).
/* 39 */ keep(delay(A,B,C)):-delay(A,B,C).
/* 40 */ keep(delay(A,B,C,D)):-delay(A,B,C,D).
/* 41 */ keep(delay(A,B,C,D,E)):-delay(A,B,C,D,E).
/* 42 */ keep(delay(A,B,C,D,E,F,G,H)):-delay(A,B,C,D,E,F,G,H).
/* 43 */ keep(use(A,B,C,D,E,F,G)):-use(A,B,C,D,E,F,G).

/* 44 */ keep(has(X,Y)):-has(X,Y),ext_in_pin(X),pin_dir(Y).
/* 45 */ keep(has(X,Y)):-has(X,Y),ext_out_pin(X),pin_dir(Y).
/* 46 */ keep(ext_in_pin(X)):-ext_in_pin(X).
/* 47 */ keep(ext_out_pin(X)):-ext_out_pin(X).

```

Figure 5.19. TDL Source Input Rules.

provided, rule 4 invokes "nd4" to generate a pin name from the pin direction and order. Rule 5 takes each unique device/pin pair and generates a unique node name which is linked to the device and to the node type with a "has_" clause (using rule 6). Figure 5.20, the generic equivalent to the TDL schematic, shows the unique node names which were assigned the various nodes in the generic form of the TDL schematic.

For "net", rule 10 is a one-to-one mapping. In the event of external pins, rules 11-13 create net names for each external pin. Rule 11 and 12 are identical, except that 11 covers external *output* pins and 12 covers *input* pins. In each case, a "connect" clause which references an external pin as its "signal" name (second argument) is converted to a connect clause which references a unique (generated) signal name which is connected to the external pin. A new connect clause is added which identifies the newly created signal name as being connected to the external pin on device "ext" with the appropriate direction (rules 11 and 12).

Rule 14 is the simple, basic rule for transforming an "occ_name" into a "box" as shown in Figure 5.17. Rule 15 is also needed to create a "box" name "ext" in the event that there are any external pins in the incoming TDL. (The "!" at the end of rule 15 will keep the translate engine from generating multiple occurrences of "box(ext)" due to more than one external pin).

Rules 16 and 17 for "box_type" are very similar to those for "box". Rule 16 is a one-to-one mapping with "device". Rule 17 creates a new "box_type", "exterior", if there are any external pins.



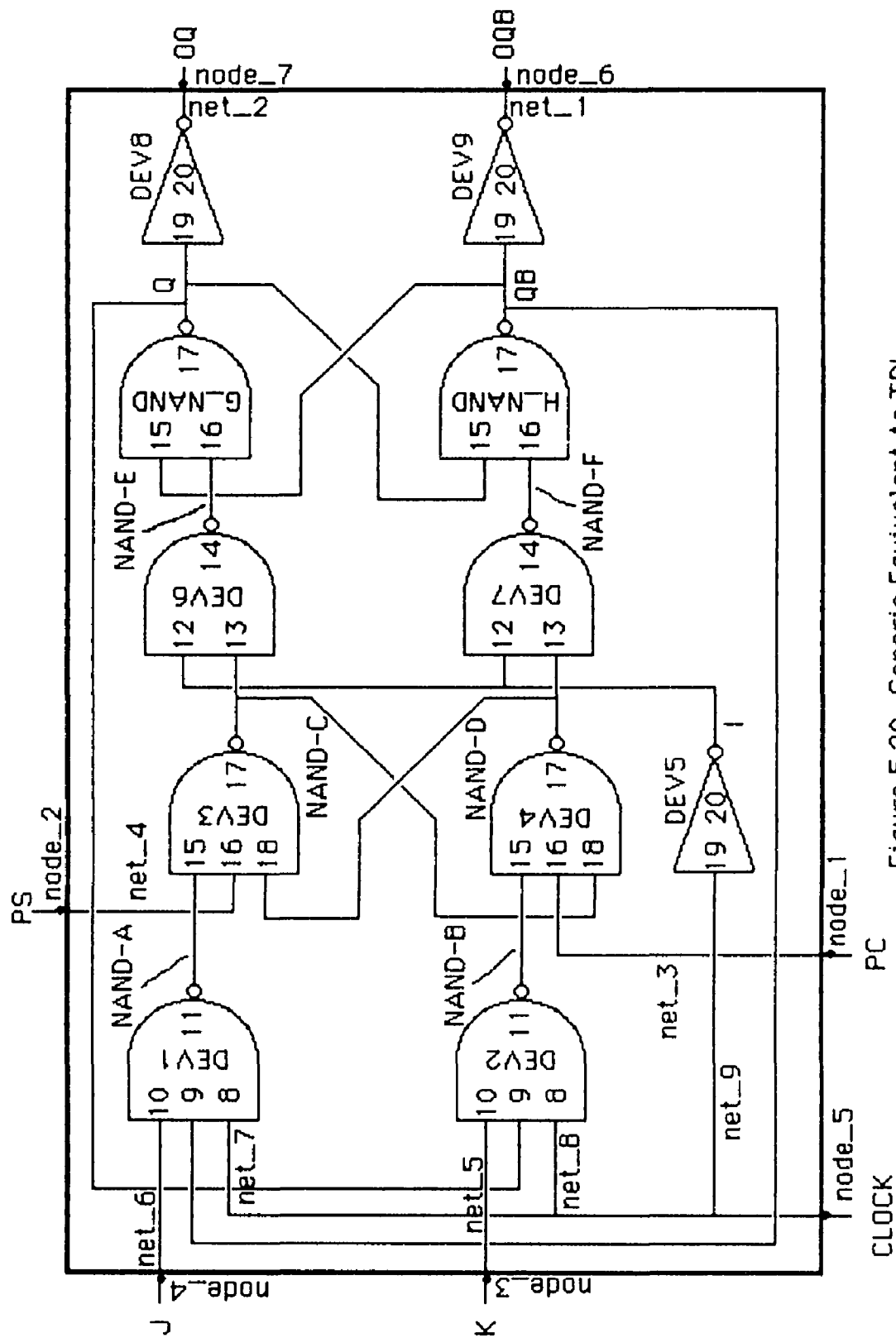


Figure 5.20. Generic Equivalent to TDL Schematic Diagram.

Rule 19 creates net types, one for each signal. Rule 18 provides a way of retrieving kept net types in the event that this is a reverse translation.

The rules for "node_type" look at every pin (rule 20) and every connect clause (rule 21) for unique "node_type" names. In this particular example, only the external pins are explicitly declared to be pins in the DBIF (Figure 5.14). No other pins are identified by name. Instead, the pins are implicitly declared by their position in the Define statement (Figure 5.13). For example, the first clause in the Define statement identifies three input signals, "J", "QB", and "CLOCK". These are numbered 1-3, respectively. This numbering is reflected in the source DBIF in the "connect" clauses. The first argument of the connect clause identifies the occurrence name, the second argument identifies the signal, the third identifies the direction of signal flow, and the fourth identifies the input source. This can either be an explicit pin or implicit pin position in the define clause ^(5.1). In order to create pin names (node types) for each input and output of a device type, the pin direction and order are concatenated (rules 21 and 7-9). Having identified all unique "node_types", rule 22 adds these to the data base.

The rules for "node_dir" (rules 23-25) look at every pin_dir clause and every connect clause for unique direction values. In general, the only two directions will be "in" and "out". Sometimes the direction "inout" will

---

^(5.1) Refer to Appendix H which describes the TDL syntax, especially, the "<source-pin match>" vs. the "<input source>" constructs.

occur for bi-directional signals. These rules work in the same way as the rules for "node_type".

The rules for "connected" are illustrated in Figure 5.21. Rule 26 corresponds to pins which are not external. Rule 27 handles external pins. Note that the execution of these rules must occur after the execution of the "node" rules which create many of the "has_" facts.

Rules 28-32 create the "has_" facts, except those which relate to "nodes" which were created in rule 6. Rule 28 is apparent from Figure 5.17; the "has" relationship between "occ_names" and "devices" is analogous to the "has_" relationship between "boxes" and "box_types". Rule 29 translates the TDL "pin has pin_dir" into the generic "node has node_dir". Note that since TDL "pins" are not equivalent to generic "nodes", the mapping must be performed via generic "node_types" (see Figure 5.17). Rule 30 also translates the "pin has pin_dir" like rule 29, but for those pins which are not explicitly declared to have a pin direction. This is the case when a pin is implied to have a pin direction in a connect clause.

The remaining rules in Figure 5.19 deal with data which must be kept or must be created in translating from TDL into generic form. Rules 36-43 and 46-47 store as kept facts those with predicates "dir_name", "descript_class", "desc", "delay", "use", "ext_in_pin", and "ext_out_pin". Rules 44-45 store "has_" facts which deal with external pins.

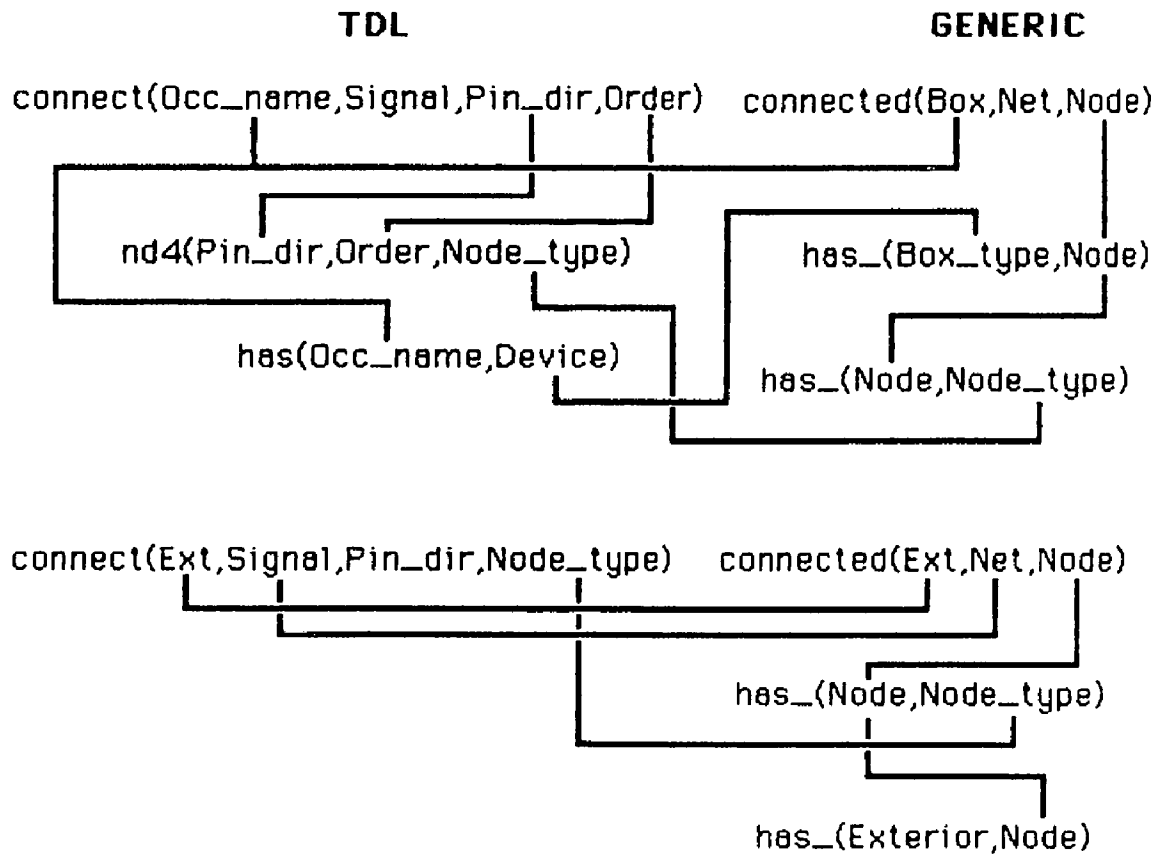


Figure 5.21. Rules for "conntected" in Translating TDL into Generic Form.

The rules for translating generic facts into HPC DBIF are the same as the DR1 target output rules shown in Figure 4.20. These rules are simpler than those for TDL and this is reflected in the data mapping shown in Figure 5.17. These rules have already been explained in section 4.3 at the end of Chapter 4. It is noteworthy that these rules were developed independently, without taking into consideration which system was used to create the generic facts to begin with.

### 5.2.3 Reverse Rules

In order to demonstrate that no data is lost using the translate engine, a set of reverse rules is also required. One set of rules converts HPC (DR1) into generic facts, and the other creates TDL DBIF from a set of generic facts. The source input rules for DR1 have already been used and explained in section 5.1. This same set is used in this DR1-TDL case as well. The target output rules for TDL DBIF are shown in Figure 5.22. The remainder of this section discusses these rules in detail.

Rule 1 maps generic "nets" onto TDL "signals" except for the case that the net was generated to connect an external pin to the interior of the module.

Rule 2 creates TDL "devices" from generic "box_types" unless the box type is the generated type "exterior".

Rule 3 and 4 handle external input pins. Rule 3 recreates an external input pin, if one exists in the file of kept facts. Rule 4 will create an external input pin if no kept fact exists and if the convention of naming the generic box type "exterior" is followed.

```

/* 1 */ signal(S):-net(S),\+connected(ext,S,Nt).
/* 2 */ device(D):-box_type(D),D\==exterior.
/* 3 */ ext_in_pin(EIP):-var(EIP),keep(ext_in_pin(EIP)).
/* 4 */ ext_in_pin(EIP):-var(EIP),\+keep(ext_in_pin(EIP)),has_(exterior,N),
      has_(N,in),has_(N,EIP),node_type(EIP).
/* 5 */ ext_out_pin(EOP):-var(EOP),keep(ext_out_pin(EOP)).
/* 6 */ ext_out_pin(EOP):-var(EOP),\+keep(ext_out_pin(EOP)),
      has_(exterior,N),has_(N,out),has_(N,EOP),node_type(EOP).
/* 7 */ pin(P):-node_type(P),\+ntgen(P,_,_).
/* 8 */ pin_dir(Pd):-node_dir(Pd).
/* 9 */ dir_name(X):-keep(dir_name(X)).
/* 10 */ descript_class(X):-keep(descript_class(X)).
/* 11 */ desc(X):-keep(desc(X)).
/* 12 */ delay(A,B,C):-keep(delay(A,B,C)).
/* 13 */ delay(A,B,C,D):-keep(delay(A,B,C,D)).
/* 14 */ delay(A,B,C,D,E):-keep(delay(A,B,C,D,E)).
/* 15 */ delay(A,B,C,D,E,F,G,H):-keep(delay(A,B,C,D,E,F,G,H)).
/* 16 */ use(A,B,C,D,E,F,G):-keep(use(A,B,C,D,E,F,G)).
/* 17 */ occ_name(X):-box(X),X\==ext.
/* 18 */ connect(W,X,Y,Z):-connected(W,X,N),W\==ext,signal(X),has_(N,Nt),
      node_type(Nt),ntgen(Nt,Y,Z).
/* 19 */ connect(D,EP,Dr,P):-connected(D,S,N),D\==ext,connected(ext,S,Fp),
      has_(Fp,EP),node_type(EP),has_(N,Nt),node_type(Nt),
      ntgen(Nt,Dr,P).
/* 20 */ has(X,Y):-has_(exterior,N),has2(N,X,Y).
/* 21 */ has2(N,X,in):-has_(N,in),has_(N,X),node_type(X).
/* 22 */ has2(N,X,out):-has_(N,out),has_(N,X),node_type(X).
/* 23 */ has(On,D):-occ_name(On),has_(On,D),device(D).
/* 24 */ ntgen(Nt,in,Z):-name(Nt,Ntl),append("in",Zl,Ntl),name(Z,Zl),integer(Z).
/* 25 */ ntgen(Nt,out,Z):-name(Nt,Ntl),append("out",Zl,Ntl),name(Z,Zl),integer(Z).

```

Figure 5.22. Target Output Rules for TDL.

Rules 6 and 7 are analogous to rules 3 and 4, but for external output pins.

Rule 7 creates TDL "pins" from generic "node_types" unless the pin is a previously generated pin of the form {"in" | "out"} "integer" (e.g.,

"out2").

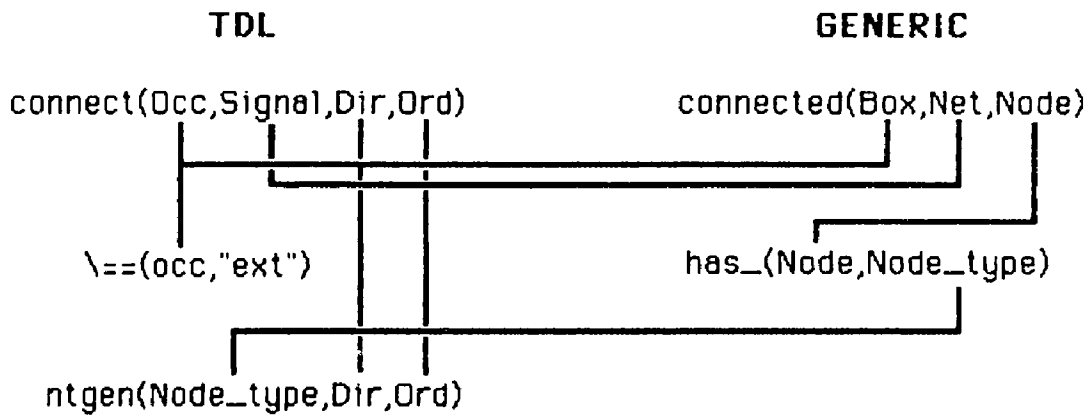
Rule 8 is a one-to-one mapping from generic "node__dirs" to TDL "pin__dirs".

Rules 9-16 recreate kept TDL facts if they were previously saved and stored in the file of kept facts.

Rule 17 creates a TDL "occ__name" from a generic "box" if it isn't the generated "ext" box.

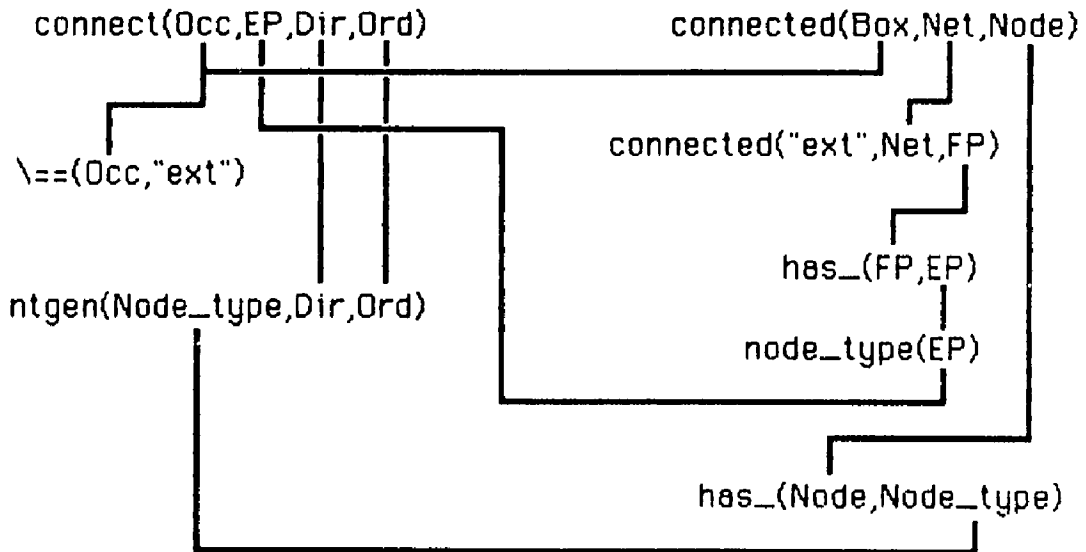
Rules 18 and 19 translate between the TDL "connect" predicate and the generic predicate "connected". These rules are shown pictorially in Figure 5.23. Rule 18 deals with net-node pairs which are not on the exterior of the module, and where the node__type was implied in the original TDL (i.e., the node__type was generated from the pin direction and its order). Rule 19 reduces two generic "connected" clauses which tie an external pin with an internal pin via a generated net into a single TDL "connect" clause which references an external pin name instead of a signal name as its second argument.

Rules 20-23 translate "has__" clauses back into TDL. Rules 20-22 create facts of the form "external pin has pin__dir". Rule 23 translates the generic "box has box__type" into the TDL "occ__name has device". There is no rule to create the TDL "pin has pin__dir" for non-external pins. However this relationship is implied in the "connect" facts where a pin is explicitly given instead of an implied order.



**RULE 18**

---



**RULE 19**

---

Figure 5.23. Connect Rules from the TDL Target Output Rules.



Rules 24 and 25 are used by the connect rules to determine if a node type was generated by concatenating the pin direction with the order.

#### 5.2.4 Forward Translation Results

Utilizing the forward rules, the translate engine first produced the set of generic facts shown in Appendix L from the source TDL DBIF (Figure 5.14). There are facts for every generic predicate including those for "node" and "net_type", which have no TDL counterpart. Also, nets were generated for external pins, which do not have signal counterparts in TDL. Thus the generic model is complete and ready for translation into another CAE/CAD/CAM data format. The prototype system log is shown in Figure 5.24. The specific number of facts of each predicate type is shown and it is reported that there are a total of 219 generic facts .

In order to be able to return back to TDL at a later time with this database, several TDL facts were kept aside since they couldn't be translated into the generic form. These are shown in Figure 5.25. Included are facts regarding "external pins", "has" relationships, "use", "delay", "desc", and "descript_class".

The second phase of this transport problem is to translate the set of generic facts into HPC (DR1). Using the forward rules (Figure 4.20) applied to the set of generic facts yields the target DBIF shown in Figure 5.26. All of the DR1 predicates in Figure 5.16 are represented. All of the TDL symbolic names in the original TDL database are apparent in this DR1 representation. Also, names generated in the translation from TDL to the generic form have also carried through (e.g., "signal(net_8)"). The

```

CProlog version 1.4d.edai
[ Restoring file /u/ua/hooper/tran11.env ]

yes

[ ?- translate('jk.dat','jk.out','jkK.out',jkff,drl,'2.0','4/11/84:13:57').
jk.dat consulted 3148 bytes 0.85 sec.

>> Translate: V1.0 <<

dbid(jkff,tdl,1,2/21/84:10:05)
tdlin rul consulted 5112 bytes 1.9 sec.
Start Up 5.05 sec.
  T=node(_1021) 20 facts.
  T=net(_1021) 18 facts.
  T=box(_1021) 12 facts.
  T=box_type(_1021) 5 facts.
  T=net_type(_1021) 18 facts.
  T=node_type(_1021) 11 facts.
  T=node_dir(_1021) 2 facts.
  T=macro_call(_1021) 0 facts.
  T=macro_def(_1021) 0 facts.
  T=connected(_1021,_1022,_1023) 43 facts.
  T=has(_1021,_1022) 90 facts.
  T=end_of_file 0 facts.
  219 facts total.
Generic 31.15 sec.
Keep 1.72 sec.
Unload 3.72 sec.
drlout.rul consulted 920 bytes 0.33336 sec.
  T=signal(_4276) 18 facts.
  T=pin(_4276) 11 facts.
  T=comp_name(_4276) 12 facts.
  T=comp_type(_4276) 5 facts.
  T=connect(_4276,_4277,_4278) 43 facts.
  T=has(_4276,_4277) 12 facts.
  T=end_of_file 0 facts.
  101 facts total.
Phase 2 17.03 sec.
Output Keep 8.25 sec.

Total time is 68.33 sec.

yes
[ ?- halt

[ Prolog execution halted ]

```

Figure 5.24. Prototype System Log for Forward Translation (TDL to DR1).

system log (Figure 5.24) shows that 101 DR1 facts were created from the set of generic facts. Utilizing the DR1 formatter presented in section 4.2.1, a file of records can be produced which is readily loaded into a native HPC format (relational DBMS).

The data mapping of Figure 5.17 shows that there are generic facts which do not translate into DR1. These are appended to the previous set

```

fromdb(jkff,tcl,'1').
toddb(jkff,drl,'2.0').
content({keep(ext_out_pin(oq)), keep(ext_out_pin(oqb)),
keep(ext_in_pin(clock)), keep(ext_in_pin(j)),
keep(ext_in_pin(k)), keep(ext_in_pin(ps)),
keep(ext_in_pin(pc)), keep(connect(ext_net_1,out,oqb)),
keep(has(oqb,out)), keep(has(oq,out)),
keep(has(pc,in)), keep(has(ps,in)),
keep(has(k,in)), keep(has(j,in)),
keep(has(clock,in)),
keep(use(dig_2_nand,=,nand,2,1,none,nandel)),
keep(use(dig_3_nand,=,nand,3,1,none,nandel)),
keep(delay(not,3,2,4,/)), keep(delay(nandel,3,2,4,/)),
keep(delay(dev9,1,1,/)),
keep(desc('THE MODULE IS A MASTER / SLAVE JK FLIP-FLOP WITH
PRESET AND PRECLEAR LINES. : ')),
keep(descript_class(gate)),
keep(dir_name(rph)),
keep(has(not,node_20)), keep(has(not,node_19)),
keep(has(nand,node_18)), keep(has(nand,node_17)),
keep(has(nand,node_16)), keep(has(nand,node_15)),
keep(has(dig_2_nand,node_14)),
keep(has(dig_2_nand,node_13)),
keep(has(dig_2_nand,node_12)),
keep(has(dig_3_nand,node_11)),
keep(has(dig_3_nand,node_10)),
keep(has(dig_3_nand,node_9)),
keep(has(dig_3_nand,node_8)),
keep(has(exterior,node_7)), keep(has(exterior,node_6)),
keep(has(exterior,node_5)), keep(has(exterior,node_4)),
keep(has(exterior,node_3)), keep(has(exterior,node_2)),
keep(has(exterior,node_1)), keep(has(qb,nt_10)),
keep(has(q,nt_11)), keep(has(nand_f,nt_12)),
keep(has(nand_e,nt_13)), keep(has(i,nt_14)),
keep(has(nand_d,nt_15)), keep(has(nand_c,nt_16)),
keep(has(nand_b,nt_17)), keep(has(nand_a,nt_18)),
keep(has(net_1,nt_9)), keep(has(net_2,nt_8)),
keep(has(net_3,nt_7)), keep(has(net_4,nt_6)),
keep(has(net_5,nt_5)), keep(has(net_6,nt_4)),
keep(has(net_7,nt_3)), keep(has(net_8,nt_2)),
keep(has(net_9,nt_1)), keep(has(node_1,in)),
keep(has(node_2,in)), keep(has(node_3,in)),
keep(has(node_4,in)), keep(has(node_5,in)),
keep(has(node_6,out)), keep(has(node_7,out)),
keep(has(node_8,in)), keep(has(node_9,in)),
keep(has(node_10,in)), keep(has(node_11,out)),
keep(has(node_12,in)), keep(has(node_13,in)),
keep(has(node_14,out)), keep(has(node_15,in)),
keep(has(node_16,in)), keep(has(node_17,out)),
keep(has(node_18,in)), keep(has(node_19,in)),
keep(has(node_20,out)), keep(has(node_1,pc)),
keep(has(node_2,ps)), keep(has(node_3,k)),
keep(has(node_4,j)), keep(has(node_5,clock)),
keep(has(node_6,oqb)), keep(has(node_7,oq)),
keep(has(node_8,in3)), keep(has(node_9,in2)),
keep(has(node_10,in1)), keep(has(node_11,out1)),
keep(has(node_12,in2)), keep(has(node_13,in1)),
keep(has(node_14,out1)), keep(has(node_15,in2)),
keep(has(node_16,in1)), keep(has(node_17,out1)),
keep(has(node_18,in3)), keep(has(node_19,in1)),
keep(has(node_20,out1)), keep(node_dir(out)),
keep(node_dir(in)),
dummy}).

```

Figure 5.25. TDL and Generic Kept Facts.

```

dbid(1jkkf,dr1,'2.0','4/11/84:13:57').
content(signal(net_9), signal(net_8), signal(net_7),
signal(net_6), signal(net_5), signal(net_4),
signal(net_3), signal(net_2), signal(net_1),
signal(nand__a), signal(nand__b), signal(nand__c),
signal(nand__d), signal(i), signal(nand__e),
signal(nand__f), signal(q), signal(qb),
pin(clock), pin(j), pin(k),
pin(ps), pin(pc), pin(oq),
pin(oqb), pin(in3), pin(in2),
pin(out1), pin(in1), comp_name(ext),
comp_name(dev9), comp_name(dev8), comp_name(h__nand),
comp_name(g__nand), comp_name(dev7), comp_name(dev6),
comp_name(dev5), comp_name(dev4), comp_name(dev3),
comp_name(dev2), comp_name(dev1), comp_type(exterior),
comp_type(dig__3__nand), comp_type(dig__2__nand), comp_type(nand),
comp_type(not), connect(net_1,ext,oqb), connect(net_2,ext,oq),
connect(net_3,ext,pc), connect(net_4,ext,ps), connect(net_5,ext,k),
connect(net_6,ext,j), connect(net_7,ext,clock), connect(net_8,ext,clock),
connect(net_9,ext,clock), connect(qb,dev9,in1), connect(q,dev8,in1),
connect(q,h__nand,in2), connect(nand__f,h__nand,in1), connect(qb,h__nand,out1),
connect(qb,g__nand,in2), connect(nand__e,g__nand,in1), connect(q,g__nand,out1),
connect(i,dev7,in2), connect(nand__d,dev7,in1), connect(nand__f,dev7,out1),
connect(i,dev6,in2), connect(nand__c,dev6,in1), connect(nand__e,dev6,out1),
connect(i,dev5,out1), connect(nand__c,dev4,in3), connect(nand__b,dev4,in2),
connect(nand__d,dev4,out1), connect(nand__d,dev3,in3), connect(nand__a,dev3,in2),
connect(nand__c,dev3,out1), connect(q,dev2,in2), connect(nand__b,dev2,out1),
connect(qb,dev1,in2), connect(nand__a,dev1,out1), connect(net_1,dev9,out1),
connect(net_2,dev8,out1), connect(net_3,dev4,in1), connect(net_4,dev3,in1),
connect(net_5,dev2,in1), connect(net_6,dev1,in1), connect(net_7,dev1,in3),
connect(net_8,dev2,in3), connect(net_9,dev5,in1), has(ext,exterior),
has(dev9,not), has(dev8,not), has(h__nand,nand),
has(g__nand,nand), has(dev7,dig__2__nand), has(dev6,dig__2__nand),
has(dev5,not), has(dev4,nand), has(dev3,nand),
has(dev2,dig__3__nand), has(dev1,dig__3__nand), dummy}).

```

Figure 5.26. Target HPC (DR1) DBIF.

of TDL kept facts in Figure 5.25. Specifically, there are "has_" facts and "node_dir" facts which do not translate, so they are kept for later reverse translation.

Once the HPC database is available, additions, changes, or deletions may occur before the database is transported back to TDL. In this example no changes were made in order to verify that all data was transported forward and backward without loss.

### 5.2.5 Reverse Translation Results

Using the same HPC results and the file of kept facts, the translate engine is invoked again. This time, generic data is created from the HPC (DR1) database. The logical content of the generic database created this time is identical to that created during the forward translation. However, the specific order of the facts is changed due to the order in which the rules are processed. The system log for this reverse translation is shown in Figure 5.27. Again, 219 generic facts were generated which matches the number of generic facts created during the forward translation.

At the end of "Phase 2", it is reported that there were 100 TDL facts generated, which again points to the differences between TDL, DR1, and the generic form. The resulting TDL facts are shown in Figure 5.28. While the order of the facts is different than that of Figure 5.14, the source TDL DBIF and the target TDL DBIF are logically equivalent. This proves that the translate engine and the knowledge base (rules) work together to transport the data without loss. In this reverse translation, no kept facts were generated, but rules could have been added to preserve the original generic nodes and `net_` types created during the forward translation.

This completes the second example of using the knowledge-based prototype transport system. Next, the techniques shown for keeping data and creating missing data are used on a more complex example involving layout data.

```

CProlog version 1.4d.edai
| Restoring file /u/ua/hooper/tran11.env |

yes
| ?- translate('jk.out','jkK.out','jkt dl.dat','jkt dlK.dat','jkff.tdl','2.1','4/11/84:15:47').
jk.out consulted 2680 bytes 0.43333 sec.

>> Translate: V1.0 <<

dbid(jkff.dr1,2.0,4/11/84:13:57)
dr1in.rul consulted 2184 bytes 0.43333 sec.
jkK.out consulted 4088 bytes 0.63333 sec.
fromdb(jkff.tdl.1)
toddb(jkff.dr1,2.0)
Start Up 8.32 sec.
  T=node(_1893) 20 facts.
  T=net(_1893) 18 facts.
  T=box(_1893) 12 facts.
  T=box_type(_1893) 5 facts.
  T=net_type(_1893) 18 facts.
  T=node_type(_1893) 11 facts.
  T=node_dir(_1893) 2 facts.
  T=macro_call(_1893) 0 facts.
  T=macro_def(_1893) 0 facts.
  T=connected(_1893,_1894,_1895) 43 facts.
  T=has(_1893,_1894) 90 facts.
  T=end_of_file 0 facts.
  219 facts total.
Generic 8.55 sec.
Keep 5.42 sec.
Unload 0.75 sec.
tdlout.rul consulted 2836 bytes 0.55001 sec.
  T=signal(_5650) 9 facts.
  T=device(_5650) 4 facts.
  T=ext_in_pin(_5650) 5 facts.
  T=ext_out_pin(_5650) 2 facts.
  T=pin(_5650) 7 facts.
  T=pin_dir(_5650) 2 facts.
  T=dir_name(_5650) 1 facts.
  T=descript_class(_5650) 1 facts.
  T=desc(_5650) 1 facts.
  T=delay(_5650,_5651,_5652) 0 facts.
  T=delay(_5650,_5651,_5652,_5653) 1 facts.
  T=delay(_5650,_5651,_5652,_5653,_5654) 2 facts.
  T=delay(_5650,_5651,_5652,_5653,_5654,_5655,_5656,_5657) 0 facts.
  T=use(_5650,_5651,_5652,_5653,_5654,_5655,_5656) 2 facts.
  T=occ_name(_5650) 11 facts.
  T=connect(_5650,_5651,_5652,_5653) 34 facts.
  T=has(_5650,_5651) 18 facts.
  T=end_of_file 0 facts.
  100 facts total.
dbid(jkff.tdl,2.1,4/11/84:15:47)
Phase 2 14.92 sec.
Output Keep 5.18 sec.

Total time is 44.23 sec.

yes
| ?- halt.

| Prolog execution halted |

```

Figure 5.27. Prototype System Log for Reverse Translation (DR1 to TDL).

```

dbid(jkff,tcl,'2.1','4/11/84:15:47').
content((signal(qb),      signal(q),      signal(nand__f),
signal(nand__e),      signal(i),      signal(nand__d),
signal(nand__c),      signal(nand__b),      signal(nand__a),
device(not),      device(nand),      device(dig__2__nand),
device(dig__3__nand),      ext_in_pin(clock),      ext_in_pin(j),
ext_in_pin(k),ext_in_pin(ps),      ext_in_pin(pc),
ext_out_pin(oq),      ext_out_pin(oqb),      pin(oqb),
pin(oq), pin(pc), pin(ps),
pin(k), pin(j), pin(clock),
pin_dir(in),      pin_dir(out),      dir_name(rph),
descript_class(gate),
desc(THE MODULE IS A MASTER / SLAVE JK FLIP-FLOP WITH PRESET
AND PRECLEAR LINES. ; ),
delay(dev9,1,1,/),      delay(not,3,2,4,/),      delay(nandel,3,2,4,/),
use(dig__2__nand,=,nand,2,1,none,nandel),
use(dig__3__nand,=,nand,3,1,none,nandel),
occ_name(dev1),      occ_name(dev2),      occ_name(dev3),
occ_name(dev4),      occ_name(dev5),      occ_name(dev6),
occ_name(dev7),      occ_name(g__nand),      occ_name(h__nand),
occ_name(dev8),      occ_name(dev9),
connect(dev1,nand__a,out,1),      connect(dev1,qb,in,2),
connect(dev2,nand__b,out,1),      connect(dev2,q,in,2),
connect(dev3,nand__c,out,1),      connect(dev3,nand__a,in,2),
connect(dev3,nand__d,in,3),      connect(dev4,nand__d,out,1),
connect(dev4,nand__b,in,2),      connect(dev4,nand__c,in,3),
connect(dev5,i,out,1),      connect(dev6,nand__e,out,1),
connect(dev6,nand__c,in,1),      connect(dev6,i,in,2),
connect(dev7,nand__f,out,1),      connect(dev7,nand__d,in,1),
connect(dev7,i,in,2),      connect(g__nand,q,out,1),
connect(g__nand,nand__e,in,1),      connect(g__nand,qb,in,2),
connect(h__nand,qb,out,1),      connect(h__nand,nand__f,in,1),
connect(h__nand,q,in,2),      connect(dev8,q,in,1),
connect(dev9,qb,in,1),      connect(dev5,clock,in,1),
connect(dev2,clock,in,3),      connect(dev1,clock,in,3),
connect(dev1,j,in,1),      connect(dev2,k,in,1),
connect(dev3,ps,in,1),      connect(dev4,pc,in,1),
connect(dev8,oq,out,1),      connect(dev9,oqb,out,1),
has(pc,in),      has(ps,in),      has(k,in),
has(j,in),      has(clock,in),      has(oqb,out),
has(oq,out),      has(dev1,dig__3__nand),      has(dev2,dig__3__nand),
has(dev3,nand),      has(dev4,nand),      has(dev5,not),
has(dev6,dig__2__nand),      has(dev7,dig__2__nand),
has(g__nand,nand),      has(h__nand,nand),
has(dev8,not).      has(dev9,not),
dummy)).

```

Figure 5.28. Target TDL DBIF Produced by the Reverse Translation.

### 5.3 CALMA to CIF

The third and final example involves transporting a CALMA GDS II Stream Format database into a CalTech Intermediate Format (CIF) database. Both of these database formats were described in section 4.2.1. The CALMA Stream Format is a widely recognized database format used to transport VLSI layouts, consisting of polygons and other geometric shapes, describing the various mask layers which are used to fabricate IC's. CIF is used for the same purpose as the CALMA Stream Format, but the entities are different. Appendices C, D, F, and G provide examples of this data and the corresponding DBIF's which were processed by the compilers and formatters described in Chapter 4.

#### 5.3.1 Data Mapping

As in the previous two test cases, the rules for translation can be defined, data models for the CALMA, CIF, and generic formats are necessary along with mappings between the data schemas. Figure 4.2 showed the BNF-like description of the *CALMA GDS II Stream Format*. A graphic equivalent of the Stream Format syntax is provided in the data model diagram of Figure 5.29. In general, a CALMA Stream Format (hereafter referred to as CALMA) file consists of one or more structures which are like macros. Each structure contains one or more elements of the type: boundary (polygon), path, node, box, text, structure reference, and array reference. Other CALMA data entities are modifiers which further define these basic entities. Each entity describes some feature of the layout.



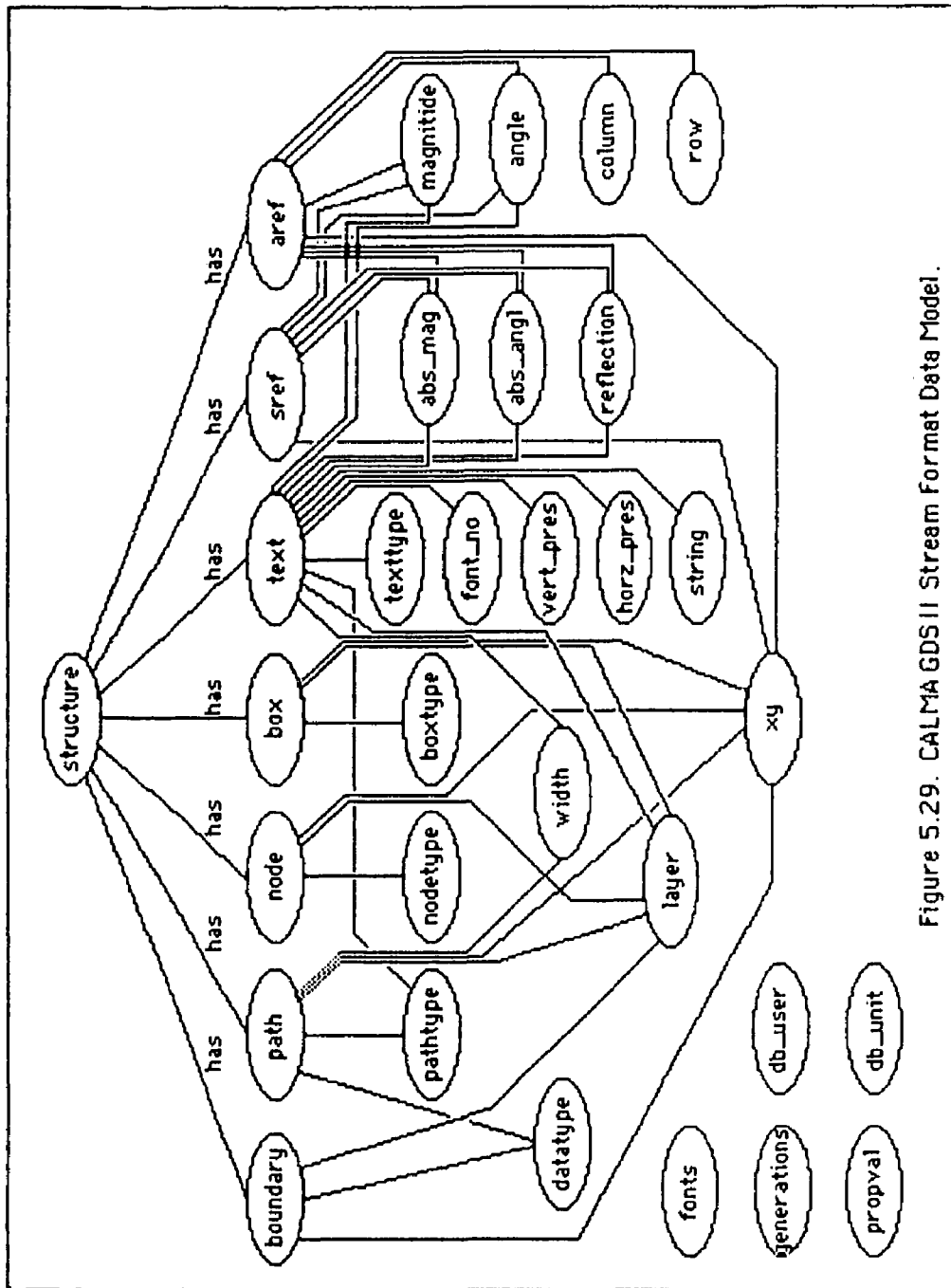
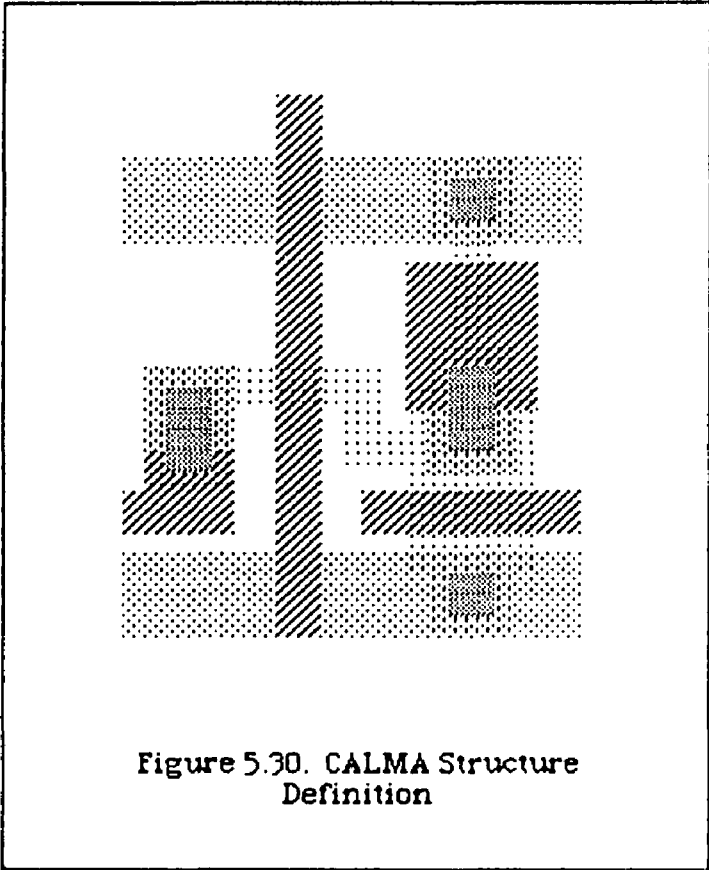


Figure 5.29. CALMA GDS II Stream Format Data Model.

One distinguishing feature of CALMA in comparison with some other IC layout data models is the *array reference*. This provides a way of creating a regular lattice with a structure reference (i.e., macro call) at each lattice node. To illustrate this, we consider the example of a CALMA structure shown in Figure 5.30. A 2 x 3 array reference using this structure would create the lattice shown in Figure 5.31. In CALMA, an array reference is defined by specifying the number of rows and columns and by indicating the reference point and X-axis and Y-axis extrema, which contain the lattice node origins, as shown in the illustration.

Another feature of CALMA which is not universally available in this type of CAD system is the ability to apply magnification factors to structures either for a structure reference (sref) or an array reference (aref). Since a CALMA data base is hierarchical, "sref"s (or "aref"s) may be nested several levels deep. When a magnification factor is applied for an "sref" or "aref", the factor is, by default, relative to the next higher level in the hierarchy. It is also possible to specify that a magnification factor be absolute. In this case, any previous scaling factors in effect from higher levels in the hierarchy are disregarded.

The CIF data model has a different set of geometric entities (see Figure 5.32). As with the CALMA, each CIF data entity describes some facet of the VLSI layout. As with the CALMA data model, CIF has polygons, boxes, wires (paths), and a symbol (macro) definition capability. However, CIF does not have an equivalent to the CALMA array reference. On the other hand, one feature of the CIF model not present in the CALMA model is the "flash" entity. This feature is common with data formats



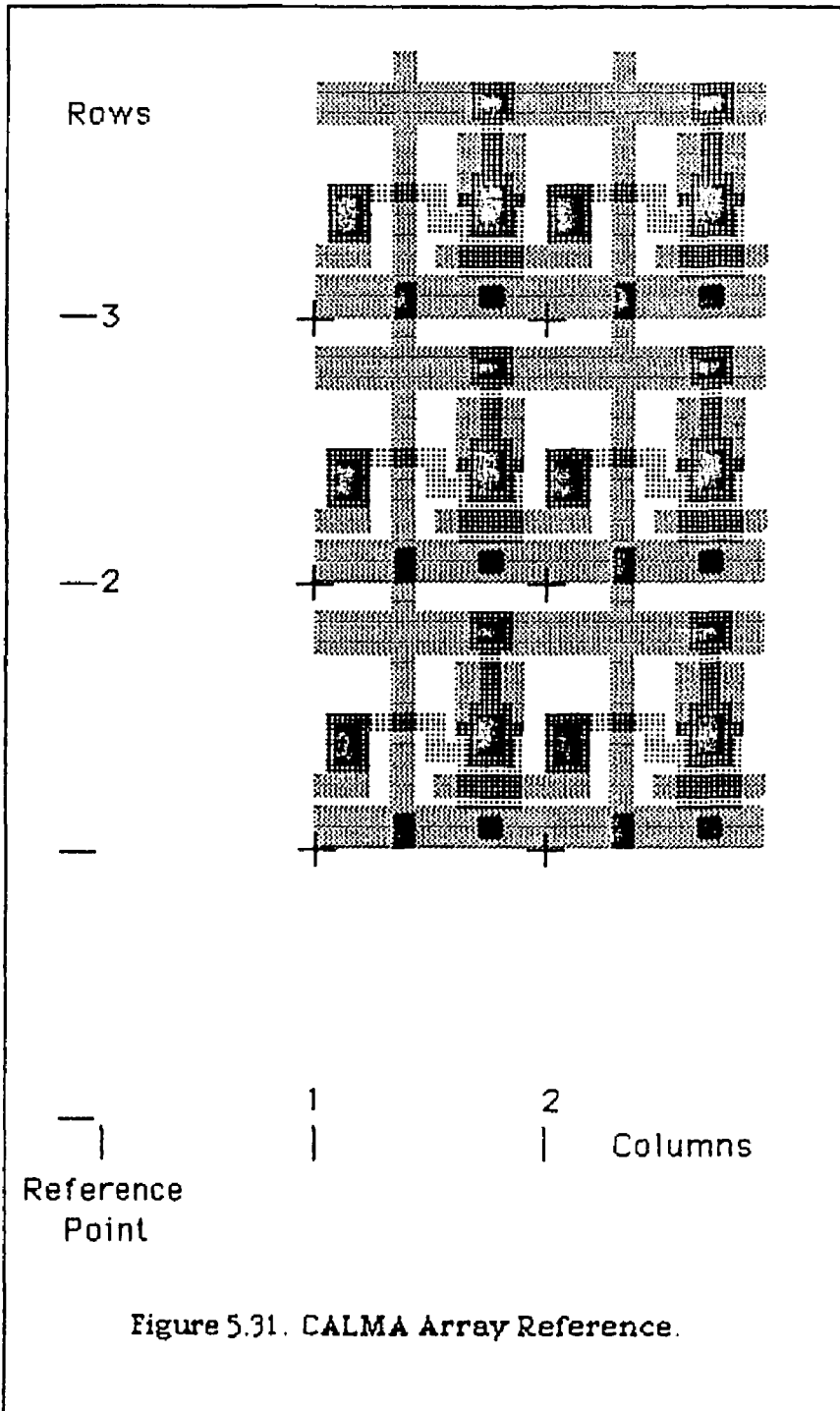


Figure 5.31. CALMA Array Reference.

which drive photo-plotters, having a circular aperture capability. The CIF flash is merely a circle of given diameter.

Also absent in CIF is the ability to scale a symbol (macro) when it is referenced. The CIF scale feature is only used during symbol definition to avoid having to write decimal points and/or zeros in the coordinates given in the symbol definition. For example, if the coordinates 43000, 52000, 160000, 200000 were needed in the body of a symbol definition, then by applying a scaling factor of 1/1000, they could be written as 43, 52, 160, and 200. This makes for more compact data bases and saves typing in the data if done manually. However, if the same basic structure is to be repeated in different sizes, then in CIF, two separate structures must be defined with differently scaled coordinates. This is in contrast with CALMA which allows two instances of the same macro to have different sizes by applying scaling factors.

In Chapter 3, a set of *generic predicates* for physical data was presented (see Figure 3.18). This set is a combination of predicates from several different geometric data models. Alternative generic sets could be defined, but this set is sufficient to demonstrate the knowledge-based methodology for CAE/CAD/CAM data transport. In general, the generic set should be as all-inclusive as possible. However, there are some flaws in the data models of existing CAD systems which should be avoided to facilitate data transfer between systems.

One anomaly in both the CALMA and CIF data models is the presence of both "polygons" and "boxes". The problem with having both of these entities is that the box is a special case of the polygon. A manifesta-

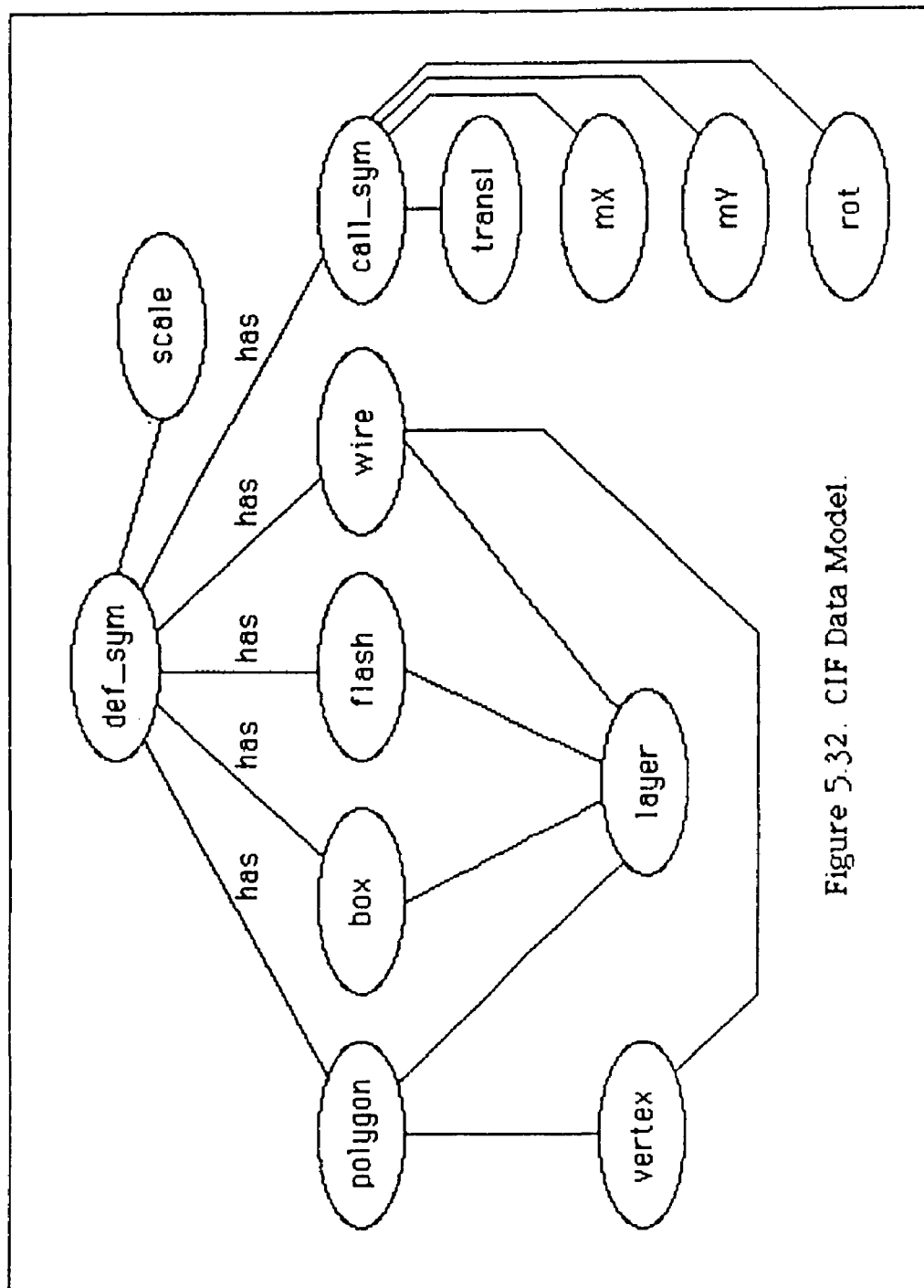


Figure 5.32. CIF Data Model.

tion of the problem would be a rectangle represented as a polygon instead of a box. There is no way to determine how to represent the data if it is transported to another system which also has both polygons and boxes. By looking at the vertices, the rectangle would meet the criteria for a box. This translation would be flawed since the original data representation wouldn't be preserved. The problem is compounded as a source data base is translated into a generic data base and then into a target data base. At each step in the transport of the data, there is the possibility of misrepresenting the data contents. The reverse translation is more disconcerting, since it is possible that the same data being represented in terms of alternative predicates. Consequently, two equivalent data bases wouldn't compare equal. To avoid this problem, the set of generic predicates defined in Chapter 3 includes only the polygon entity, and not the box entity. In this way two sets of generic data can be compared since there is only one way of representing each data entity.

The next step in preparing for the transport of a CALMA database to a CIF database is defining the mapping from CALMA to the generic format. Figure 5.33 shows this mapping. Many CALMA predicates map directly onto generic predicates. Some CALMA predicates can be mapped onto generic predicates, but a mapping other than one-to-one is required. These predicates map onto generic predicates indicated in "()". Still another set of CALMA predicates have no generic counterpart.

Those predicates having a mapping which is not one-to-one, require a special approach. The CALMA predicates "aref", "columns", and "rows" map onto the generic "macro_call", since the generic data model has no

CALMA Stream		Generic
boundary(B)	→	polygon_(B)
box(B)	→	polygon_(B)
path(P)	→	wire_(P)
layer(S,L)	→	layer_(S,L)
xy(S,X,Y,I)	→	vertex_(S,X,Y,I)
width(S,W)	→	width_(S,W)
text(T)	→	text_(T)
structure(S)	→	macro_def(S)
string(T,S)	→	textval_(T,S)
vert_pres(T,V)	→	v_just(T,V)
horz_pres(T,H)	→	h_just(T,H)
font_no(T,N)	→	tfont_(T,N)
db_unit_meters(S)	→	scale_(S)
sref(S,Name)	→	macro_call(S,Name)
magnitude(S,M)	→	magnif_(S,M)
abs_mag(S)	→	relative_magnif(S)
abs_angl(S)	→	relative_orient(S)
datatype(S,D)	→	?
pathype(S,P)	→	?
node	→	?
nodetype	→	?
boxtype	→	?
textype	→	?
reflection	→	(orient)
angle	→	(orient)
aref	→	(macro_call)
column	→	(macro_call)
row	→	(macro_call)
fonts	→	?
generations	→	?
propval	→	?
db_user_units	→	?

Figure 5.33. Mapping from CALMA to Generic Predicates.

array reference. To do the mapping, it is necessary that CALMA "aref"s be converted to a set of "sref"s. While the generic data is correct initially, the potential for error exists since each individual "sref" can be changed independently, but not in the original "aref". To perform the translation, each node in the "aref" lattice is determined and an "sref" placed at that origin.

When specifying an "aref", it is possible to apply X-axis reflection and/or an angle of rotation. This presents a further complication in



translating "aref"s, since the origin for each individual "sref" must be computed, according to the angular transformation and the row/column position (see Figure 5.34). Also, the angle of rotation and reflection must be applied to the coordinates of each "sref". The relative orientation of each "sref" with respect to the array lattice remains fixed.

The CALMA predicates "reflection" and "angle" map onto the single generic predicate, "orient_". The reflection predicate mirrors the data about the "X" axis. "Angle" rotates the data about the "Z" axis (within the X-Y plane) with a positive angle in the counterclockwise direction. The single generic predicate, orient_, has 3 angles of rotation: about the X-axis, Y-axis, and Z-axis. Therefore, the CALMA predicates, "reflection" and "angle", map onto the single generic predicate "orient_".

Those CALMA data items which simply do not map onto generic predicates are stored as "kept facts". These are available for processing during a return translation.

Having defined a mapping from CALMA to the generic form, the second half of the translation involves mapping generic predicates onto CIF. Figure 5.35 shows this mapping. As with the CALMA-Generic mapping, there are also mismatches between the generic and CIF formats. Two CIF entities have no generic counterpart: "flash" and "box". A flash can be approximated as an octagon (a special case of polygon, see Figure 5.36). A CIF box is just a rectangular polygon. In the reverse translation from generic data into CIF, all polygons are inspected for the special cases octagon and rectangle. In these cases, a polygon is translated as either a flash or a box.

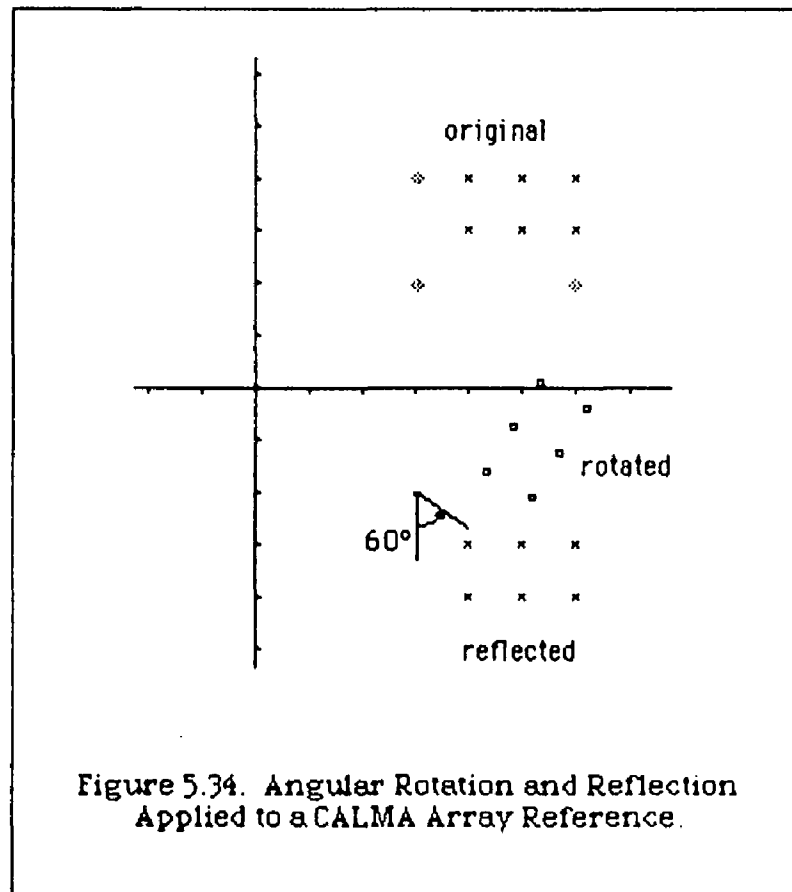


Figure 5.34. Angular Rotation and Reflection Applied to a CALMA Array Reference.

CIF		Generic
polygon(P)	←	poly gon _ (P)
box(B,L,W,CX,CY)	←	poly gon _ (B)
flash(F)	←	poly gon _ (F)
wire(W,Wid)	←	wire _ (W)
wireW,Wid)	←	width _ (W,Wid)
def _ sym(N)	←	macro _ def(N)
scale(N,A,B)	←	scale(S)
layer(S,L)	←	layer _ (S,L)
transl(P,X,Y,I)	←	vertex _ (P,X,Y,I)
vertex(P,X,Y,I)	←	vertex _ (P,X,Y,I)
call _ sym(S,Name)	←	macro _ call(S,Name)
?	←	magnif _ (S,M)
?	←	relative _ magnif(S)
?	←	relative _ orient(S)
mirrorx(P,I)	←	orient(X,AX,AY,AZ)
mirrory(P,I)	←	orient(X,AX,AY,AZ)
rotate(P,A,B,I)	←	orient(X,AX,AY,AZ)
has(N,P)	←	has _ (N,P)
?	←	text _ (T).
?	←	textval _ (T,Str).
?	←	h _ just(T,N).
?	←	v _ just(T,N).
?	←	tfont _ (T,F).

Figure 5.35. Mapping from Generic to CIF Predicates.

There are also two sets of generic predicates without CIF counterparts. The first stores data about text strings. CIF has no text facility. The second set of generic predicates without CIF counterparts are "magnitude" and "abs_mag". In the case of text predicates, the generic data is stored as "kept facts". For generic magnitudes which are applied at the time of a macro call, the lack of CIF counterpart requires that separate macros be defined for each reference at a different magnitude. Note that the CIF scaling feature is not equivalent to the generic magnitude feature. CIF scaling is merely a way of reducing the number of digits typed for defining macro coordinates. Scaling is not applied when the macro is "called". Consequently, all references to a macro are at the same size.

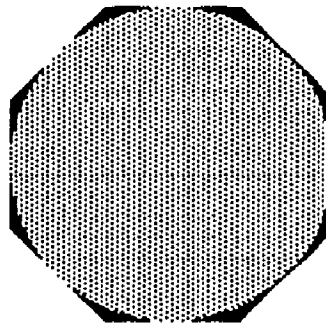


Figure 5.36. Octagonal Approximation of a CIF Flash.

To test out the solution to the CALMA-CIF data transport problem, the sample data introduced in Chapter 4 was used. Figure 4.3 shows a plot of this CALMA GDS II data. The data was modified slightly to make the test case more difficult and to exhibit more of the features of CALMA. For example, an array reference was added to the original database. Appendix I shows the actual CALMA data which was used as the source DBIF for this test case. This data consists of two CALMA structure definitions: "dev" and "t18", containing several boundaries and some paths. Inside of the structure "dev" is a structure reference to "t18" and also two array references to "t18", one at half-size reduction and 90° rotation. Figure 5.37 shows the resulting hierarchy which represents this CALMA database. When translated into CIF, the database must represent the array references as individual structure references with the appropriate orientation and translation. Figure 5.38 shows the hierarchy as it should be represented in CIF.

### 5.3.2 Forward Rules

To accomplish the translation from CALMA to CIF, the first step is to define rules for transforming the CALMA source data into generic format. Appendix J contains the Prolog rules for transforming CALMA into generic form. In order to minimize typing, several of the CALMA entities are abbreviated (e.g., "bdry" for boundary, "strct" for structure). The first 16 rules are trivial. Rules 1-9 and 13-16 are one-to-one mappings as shown in Figure 5.33 (with the exception of abbreviations used). Rule 10, dealing with vertices, is a one-to-one mapping except for the case where the entity possessing the vertex is an "aref". In this case, an x-y coordinate is not

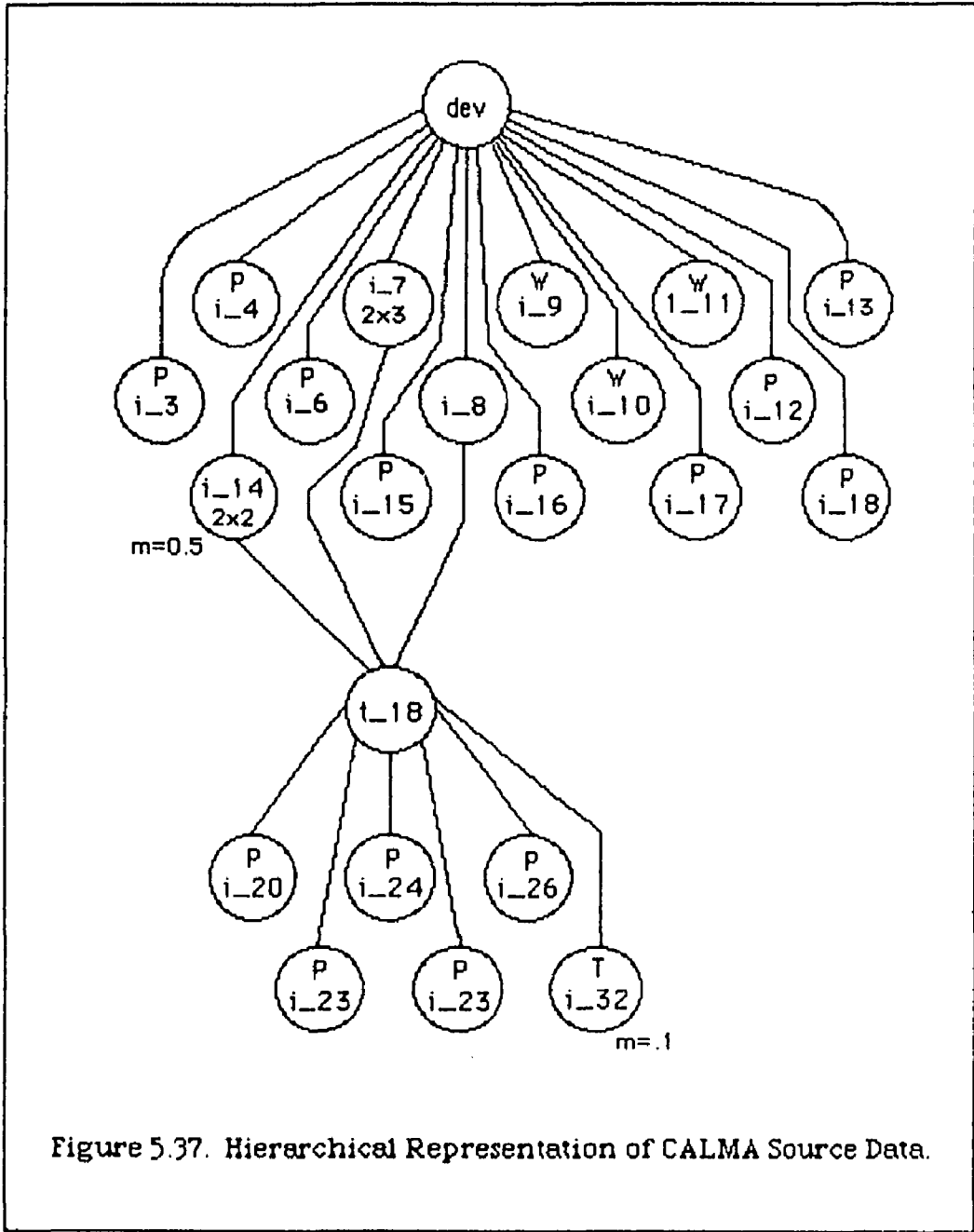


Figure 5.37. Hierarchical Representation of CALMA Source Data.

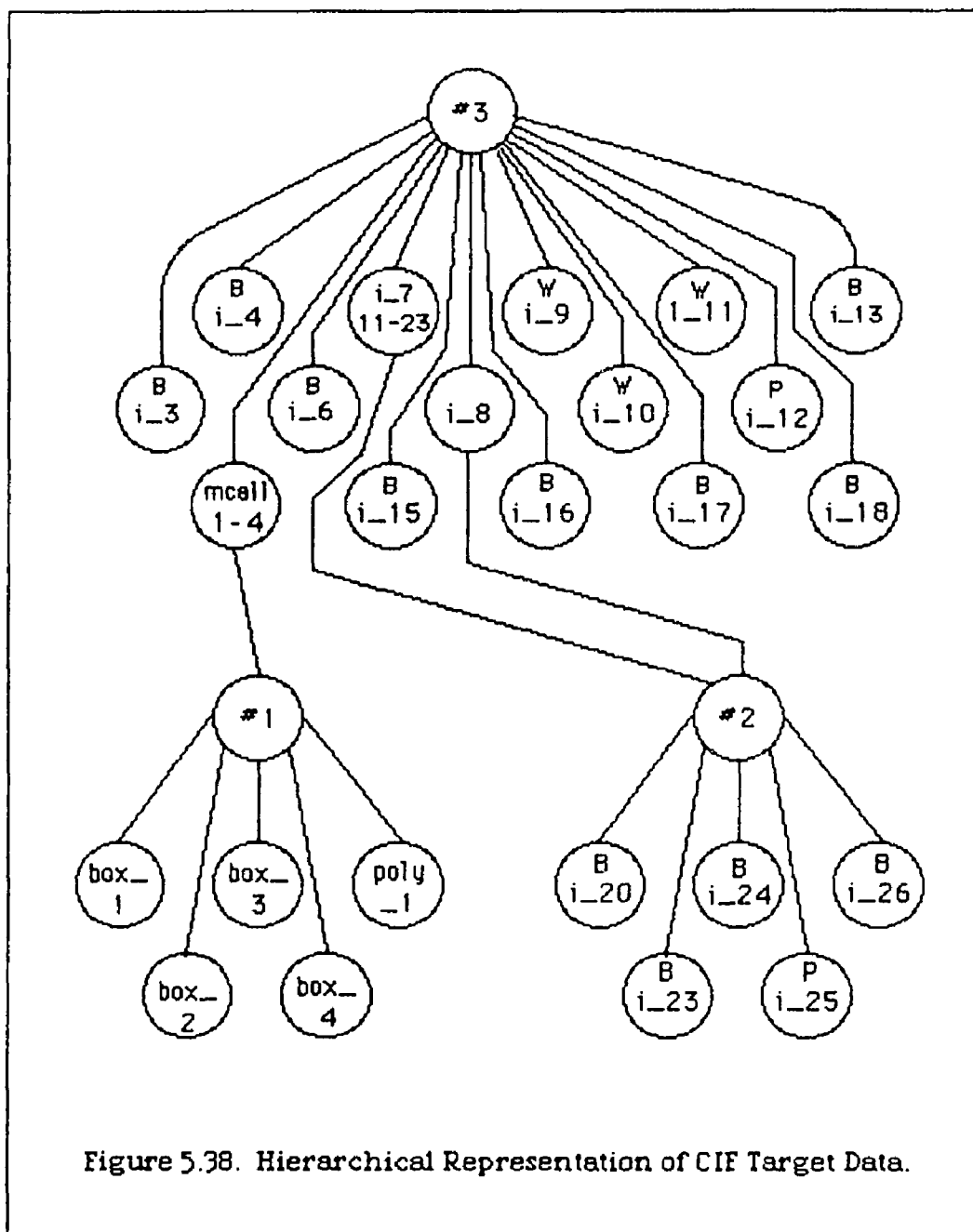


Figure 5.38. Hierarchical Representation of CIF Target Data.

translated. These coordinates are special cases handled by other rules. Rules 11 and 12 merely map CALMA text font numbers onto generic text font names.

Rules 17 and 18 transform CALMA reflection and angle entities onto the single generic orientation entity as described previously in section 5.3.1.

Rules 19-25 deal with "sref"s and "aref"s. Rule 19 maps all CALMA "sref"s onto generic macro `_calls`. Rule 20 begins the translation of "aref"s by determining the number of rows and columns in the array lattice. "Aref(N,Name)" obtains from the input CALMA DBIF the identifier of an array reference (N) and the structure name (Name) which is being placed at each lattice node. Next, "row_gen" (rules 22 and 23) and "rc_gen" (rules 24 and 25) are invoked to assert a series of facts ("`current_rc(ArrayID, Row, Column)`"), one for each row-column node in the lattice.

Rule 21 begins the generation of macro `_call`'s for each lattice node. One at a time, each row-column node is retrieved from the Prolog database. The structure referenced by the "aref" is determined next by mapping "n" to "Name" using "aref(N,Name)". The next several Prolog clauses through "name(M,NM)" create a new "sref" name for the current "aref" node. The "sref" name is the concatenation of the original "aref" identifier with the row and column number. For example, the first node of "aref(abc,t18)" would be identified as "abc__1__1", followed by "abc__1__2" and so on. The remainder of rule 21 generates a new set of x-y coordinates for the origins of the new "sref"s and transfers any "has" or "magnitude" attributes from the original "aref" to each new "sref". This is accomplished by invoking rules 44-56, defining "xy_asst", "has_asst",



and "mag_asst".

To summarize, for each node in the array lattice, a new "sref" is created. The "xy_asst" determines the origin of the structure reference. This process was discussed in the previous section (5.3.1) and is shown pictorially in Figure 5.34. Reflection and/or angular rotation applied to the "aref" is used in calculating the new "sref" origin. Rules 45-49 implement this. The same equations were used to produce Figure 5.34.

Any structure which contains (i.e., "has") the original "aref" must now be shown to contain each newly created "sref". This is accomplished by rules 50 and 51. Similarly, any magnitude which was applied to the "aref", must be applied to each new "sref". Rules 52 and 53 perform this.

Rule 26 translates all CALMA "has" facts onto generic "has_" facts, except those for CALMA "aref"s and nodes, which are not directly translated into generic facts.

Rules 27-43 keep all CALMA data which cannot be represented in terms of generic facts. This data will later be used for reverse translation to fill in the gaps in a generic database and to provide a complete CALMA database.

Finally, rules 54 and 55 are used to map CALMA absolute angle and magnitude indicators onto generic relative indicators. Rules 56 and 57 are used to define a superset (union) called "st_ref", consisting of all "sref"s and "aref"s.

These rules constitute the means by which CALMA data is transformed into a set of generic facts. The other set of rules necessary to complete the transport of data from CALMA to CIF is the set which transforms generic facts into CIF. These are also contained in Appendix J. This set of rules is ordered alphabetically by clause. The first rule in this set defines "asst_uniq" which stands for "assert unique". This is a miscellaneous utility which guarantees that a single fact is only contained once in the Prolog database.

Rule 1 defines a CIF "box" which in turn invokes "box_in" (rule 3) which in turn invokes "box_chk" (rule 2). In order to illustrate the use of these rules, consider the CALMA design hierarchy of Figure 5.37. Item "i_20" will translate into a CIF box. As the diagram shows, there are 3 distinct (indirect) references to i_20 by i_7, i_8, and i_14. The references by i_7 and i_8 are at the original scaling. However, the reference by i_14 has a reduction factor of one-half. So, a new i_20 will need to be generated with all coordinates scaled appropriately since CIF doesn't have a scaling feature. The same scaling will apply to all constituents of "t18", but for purposes of illustrating the rules, we concentrate on i_20.

Starting with box_chk (rule 2), generic polygons are inspected to verify whether the geometric constraints of a box (rectangle) are met. Specifically, there must be 5 vertices with the first and last being equal. In addition, the vertices must form an orthogonal polygon and this is checked for as well. Finally, the vertices of the polygon are converted to an alternate CIF box notation using a center coordinate (CX,CY), a length, a width, and an orientation within the X-Y plane. This orientation is

defined as a CIF direction vector (formed by the line intersecting points (0,0) and (DX,DY)) rather than by an angle. `Box_chk(B ,L ,W ,CX ,CY ,DX ,DY)` will return all boxes with their identifiers and defining parameters.

In the example, `i_20` meets the requirements for a box, so it is one of those returned by `"box_chk"`. `Box_in` (rule 3) retrieves each box from `box_chk` and then removes its vertices from the Prolog database. Since all the information represented by the vertices is contained in the CIF box parameters, there is no need to translate the vertices themselves. The last clause (rule 3) `"retr(vertex_(B,_,_,_))"` deletes all vertices of Box "B" from the Prolog database. For `i_20`, the vertices are converted to center, length, width, and orientation. Once `"box_in"` retrieves a box and its parameters, processing returns back to `"box"` (rule 1).

The remainder of rule 1 handles the case when the box is called by a higher level macro and a magnification factor is applied. `"Mac_link(N,Ref)"` finds all macro definitions which refer to the current box "N". Then, `"box_scale"` (rules 4-6) is invoked to generate a scaled version of the box being processed. For `i_20`, the processing continues according to the hierarchy of Figure 5.37. As indicated earlier, since `i_7` and `i_8` reference `i_20` (via `t_18`) without any scaling factor, no new scaled instance of `i_20` is required. However, since `i_14` references `i_20` (via `t_18`) and applies a scale factor, a complete duplicate of `t_18` must be generated (including `i_20`). This is reflected in Figure 5.38, which has both `i_20` (as part of symbol #2) and also `box_1` (as part of symbol #1). `Box_1` is a scaled down version of `i_20`. Rules 4-6 (`box_scale`) along with

rules 26-27 (`get_mag`), rules 23-25 (`g_m`), rules 561 (`new_call`), rules 62-64 (`new_inst`), and 53-58 (`mhas`, `mhas2`) accomplish this.

The rules for `"get_mag"` and `"g_m"` determine which magnitude (scaling) to apply considering whether it is absolute or relative, multiplying all nested magnitude factors. The rules `"new_call"` and `"new_inst"` generate new scaled instances and alter the original reference (`call`) to point to the newly created scaled instance. The rules `"mhas"` and `"mhas2"` establish all of the appropriate `"has"` links in the new macro definition (the scaled `t_18`) and its constituent elements (scaled versions of `i_20`, `i_23`, `i_24`, `i_25`, and `i_26`).

Continuing on to describe the remainder of the rules (Appendix J) which transform generic data to CIF data, the next rule not yet described is rule 7 (`call_sym`). For every generic `"macro_call"`, including those generated due to scaling, a CIF `"call_sym"` is produced. Since symbols in CIF are identified by integers rather than by name as in the generic form, a mapping is produced when a generic macro definition is encountered. This mapping is stored in the Prolog data base and is then included as part of the kept data by rule 45.

Since CIF symbol calls can have rotations applied as in CALMA, the generic orientation values are evaluated (rules 8-16) in order to produce the necessary CIF transformations: mirror about the X or Y axis and/or rotate in the X-Y plane (refer to Figure 4.8 for the specific syntax). Finally the origin of the generic macro call is used as the translation coordinate for the CIF symbol call.

Rule 17 (`def__sym`) is used to translate a generic macro definition into a CIF symbol definition. This rule merely maps generic macro names onto unique CIF symbol numbers and stores the mapping as a fact into the Prolog data base as indicated previously.

Rules 19-22 create CIF flashes from generic polygons in the manner that CIF boxes were created. In this case, "flash__chk" (rule 22) determines whether the generic polygon is an octagon. If so, then a CIF flash is created. Also, since a flash may be a part of a generic macro which is referenced with a scaling factor, we also include rules 20-21 (`fl__scale`) which is analogous to `box__scale` discussed in detail previously.

Rule 28 (`has`) is straight-forward. Any generic "has__" data is mapped onto a CIF "has" clause, except for text. Since CIF doesn't provide for text, it is stored with the kept facts (rule 33). Also, in translating from generic "has__" data to CIF "has" data, all generic macro names must be converted to CIF symbol numbers using previously generated "map__" facts.

Rules 29-45 store untranslatable generic facts into kept facts. Also any artificially created CIF symbol calls or CIF symbol definitions (due to lack of a magnification factor) are stored along with the original generic macro name. For example, in Figure 5.38, CIF symbols #1 and #2 are equivalent to separate calls to the same generic macro `t__18` of Figure 5.37. These relationships are stored as kept facts. The majority of the "keep" rules store text values and their related characteristics (e.g., font, magnitude, orientation, etc.). This will be more evident as the results of translation are presented further on in section 5.3.4.

Rules 46-49 (layer) map generic layer numbers onto CIF layer names. Layers indicate which physical medium will be used to fabricate the layout feature. For example, paths can be formed in metal or polysilicon using the NMOS fabrication process. These layer rules merely guarantee a consistent mapping. Note that rule 49 is a default rule used if a generic layer number is used which has not been assigned to a CIF layer type (nd, np, or nm for NMOS diffusion, polysilicon, and metal, respectively). To create a unique CIF layer name out of an unrecognized generic layer number, the character "l" is prepended to the generic layer number. This may cause an error when the native CIF file is produced by a formatter from the CIF DBIF. At that time, the owner of the data would need to determine the correct fabrication layer to be used.

Rules 50-51 (layer__chk) are used to add layer information for new scaled versions of polygons, boxes, wires, etc. when they are created from a single generic item.

Rules 52-64 have been discussed previously in conjunction with generating scaled versions of various generic data items due to lack of a CIF scaling capability.

Rules 65-66 are used to transfer orientation information when a new symbol call is created as a result of scaling by rule 60.

Rules 67-71 are used to create CIF polygons from those generic polygons which do not qualify as boxes or flashes.

Rules 72-77 translate generic vertices into CIF vertices with the generic scaling factors applied.

Rules 78-81 create CIF wires from generic wires by applying any scaling factors in effect and creating new instances where necessary. This is analogous to the scaling scenario for boxes.

This completes the description of the two sets of rules required for forward translation from CALMA to CIS. The first set translated from CALMA to generic data; the second from generic to CIF data. The results of applying these rules to actual data are presented below in section 5.3.4, after first describing the reverse translation rules for CIF to CALMA transport.

### **5.3.3 Reverse Rules**

In order to produce CALMA data from CIF data, two sets of rules are used as in the forward translation case. The first set translates CIF to generic data; the second translates from generic to CALMA. These sets of rules are included in Appendix K.

The rules translating CIF into generic data are less numerous than were those for the generic to CIF translation. This is principally because CIF is a simpler data model than the generic model. Kept data is used to avoid propagating artificial CIF symbols, created for lack of CIF scaling, back into the generic data. Instead, these artificial CIF symbols are deleted by the rules for "macro_def". Similarly calls to artificial CIF symbols are removed by the rules for "macro_call".

CIF polygons, boxes, and flash are mapped back onto generic polygons by the rules for "polygon_" in Appendix K. Included in these rules is the programming necessary to convert from CIF box center, length, width, and direction vectors into 5 polygon vertices (the first equal to the last). Also, CIF flashes are converted into generic polygons (octagons) and the programming to do this is also embedded within the "polygon_" rules.

Finally, rules are included to compute a three-dimensional transformation matrix from CIF translation, X-axis and Y-axis mirroring, and X-Y plan rotation. This transformation matrix is used to complete all generic vertices.

The remainder of the CIF input rules are the reverse of the CIF output rules. Also, generic data is re-created from any kept data previously generated.

The second set of rules takes generic data and produces CALMA output. These rules are also presented in Appendix K and are for the most part one-to-one mappings. One exception is the set of rules which re-created CALMA "aref"s from individual generic "macro_call"s. This is accomplished by looking for "macro_call"s which have a row-column designator appended to a name which was stored in the kept data as an "aref". These rules were applied to the output data from the forward translation, and the results are presented in section 5.3.5.



### 5.3.4 Forward Translation Results

With the CALMA to CIF rules defined, the prototype translation engine was used to generate CIF DBIF from CALMA DBIF. The original data for this test case was described earlier in section 5.3.1. The actual source DBIF is contained in Appendix I and the hierarchical nature of the data was illustrated in Figure 5.37.

The system log of the translate engine is shown in Figure 5.39. The first phase of the translation involves translating CALMA data into generic data. This phase is indicated in the log by the lines beginning "T= ..." in Figure 5.39 which are between the lines "Start Up ..." and "Generic ...". Note that 201 generic facts were generated. It is important that this number be compared with the number of generic facts generated during the reverse translation.

The 201 generic facts generated are shown in Appendix M. There are facts for every generic predicate except for "tfont_" which means that the input CALMA data specified no text font for any input text string (which is true). The generic polygons, wires, and layers were just simple translations of their CALMA counterparts. However, the source CALMA data contained 2 "aref"s (i_7 and i_14) and 1 "sref" (i_8) as shown in Figure 5.37. Since the generic data model doesn't contain an "aref" equivalent, the "aref"s were translated into individual generic "macro_call"s. This is reflected in Figure 5.39. Since i_14 is a 2x2 "aref", the "macro_call"s for i_14__1__1 through i_14__2__2 are contained in the set of generic data. Similarly, there are 6 individual "macro_call"s in the set of generic data replacing the single 2x3 "aref"

```

CProlog version 1.4d.edai
[ Restoring file tran11.env ]

yes
| ?- translate('gds.dat','3cif.out','3cifK.out','test.db',cif.'1.0','7/26/84:21:55').
gds.dat consulted 6324 bytes 1.0167 sec.

>> Translate. V1.0 <<

dbid(test.db,gds,1.4,7/24/84:14:50)
gdsin.rul consulted 7188 bytes 1.3667 sec.
Start Up 7.5 sec.
T=polygon(_1638) 14 facts.
T=wire(_1638) 3 facts.
T=macro_def(_1638) 2 facts.
T=macro_call(_1638,_1639) 11 facts.
T=scale(_1638) 1 facts.
T=layer(_1638,_1639) 18 facts.
T=vertex(_1638,_1639,_1640,_1641) 105 facts.
T=width(_1638,_1639) 3 facts.
T=orient(_1638,_1639,_1640,_1641) 2 facts.
T=has(_1638,_1639) 29 facts.
T=magnif(_1638,_1639) 6 facts.
T=relative_orient(_1638) 2 facts.
T=relative_magnif(_1638) 1 facts.
T=text(_1638) 1 facts.
T=textval(_1638,_1639) 1 facts.
T=h_just(_1638,_1639) 1 facts.
T=v_just(_1638,_1639) 1 facts.
T=tfoot(_1638,_1639) 0 facts.
T=end_of_file 0 facts.
201 facts total.
Generic 6.15 sec.
Keep 0.92 sec.
Unload 2.22 sec.
cifout.rul consulted 12572 bytes 2.8333 sec.
T=polygon(_4942) 3 facts.
T=box(_4942,_4943,_4944,_4945,_4946) 16 facts.
T=flash(_4942,_4943,_4944,_4945) 0 facts.
T=wire(_4942,_4943) 3 facts.
T=def_sym(_4942) 3 facts.
T=has(_4942,_4943) 33 facts.
T=scale(_4942,_4943,_4944) 3 facts.
T=call_sym(_4942,_4943) 11 facts.
T=transl(_4942,_4943,_4944,_4945) 11 facts.
T=mirrorx(_4942,_4943) 0 facts.
T=mirrory(_4942,_4943) 0 facts.
T=rotate(_4942,_4943,_4944,_4945) 16 facts.
T=vertex(_4942,_4943,_4944,_4945) 44 facts.
T=layer(_4942,_4943) 22 facts.
T=end_of_file 0 facts.
165 facts total.
Phase 2 18.63 sec.
Output Keep 1.13 sec.

Total time is 40.05 sec.

yes
| ?- halt.

[ Prolog execution halted ]

```

Figure 5.39. System Log for the Forward Translation (CALMA to CIF).

i_7. Vertices were created for each of these newly generated "macro_call"s from the x-y coordinates of the "aref" as previously described in section 5.3.1. The CALMA "angle" fact for i_14 in the source data (Appendix I) was used to determine the individual vertices for each "macro_call" generated and the rules for performing this are shown in Appendix J (rules 44-49). Having used this angle fact for i_14, it is not included in the generic set, since no rotation should apply to the individual "macro_call"s. However, the i_14 angle is stored as part of the kept facts generated during this first part of the forward translation.

The kept facts generated in translating CALMA into generic data are shown in Figure 5.40. This is the CALMA data which cannot be represented in terms of generic predicates.

The next part of the forward translation converts generic data into CIF data. As shown in the system log (Figure 5.39), this part of translation generated 165 facts. The reduction in the number of facts is a clue that some generic data couldn't be represented in CIF. This is not strictly true since the output DBIF predicates could contain more information (parameters) than the generic predicates. The actual CIF produced is presented in Appendix N. As Figure 5.35 shows, all CIF data has a generic counterpart (although not always one-to-one). The converse is not true; there are generic predicates which do not map onto CIF counterparts. One example is the generic predicate "magnif_". As indicated previously in section 5.3.1, CIF doesn't allow scaling to be applied to a macro when it is referenced. Consequently, there are complex CIF output rules (get_mag, g_m,new_call, and new_inst) which create scaled instances of generic

```

fromdb('test.db',gds,'1.4').
toddb('test.db',cif,'1.0').
content({keep(angle(i_14,90)),
keep(xy(i_14,46000,8000,3)),
keep(xy(i_14,53250,3000,2)),
keep(xy(i_14,46000,3000,1)),
keep(xy(i_7,35000,3000,3)),
keep(xy(i_7,46000,0,2)),
keep(xy(i_7,35000,0,1)),
keep(columns(i_14,2)),
keep(columns(i_7,3)),
keep(rows(i_14,2)),
keep(rows(i_7,2)),
keep(aref(i_14,t18)),
keep(aref(i_7,t18)),
keep(db_user_unit(1e-05)),
keep(generations(3)),
keep(fonts('gdsii:font.tx',2)),
keep(fonts('gdsii:font.tx',1)),
keep(dtatyp(i_26,0)),
keep(dtatyp(i_25,0)),
keep(dtatyp(i_24,0)),
keep(dtatyp(i_23,0)),
keep(dtatyp(i_20,0)),
keep(dtatyp(i_18,0)),
keep(dtatyp(i_17,0)),
keep(dtatyp(i_16,0)),
keep(dtatyp(i_15,0)),
keep(dtatyp(i_13,0)),
keep(dtatyp(i_12,0)),
keep(dtatyp(i_11,0)),
keep(dtatyp(i_10,0)),
keep(dtatyp(i_9,0)),
keep(dtatyp(i_6,0)),
keep(dtatyp(i_4,0)),
keep(dtatyp(i_3,0)),

```

Figure 5.40. Kept Facts Generated in Translating CALMA into Generic Data.

macro definitions. Referring to Figures 5.37 and 5.38, it can be seen how the original CALMA array reference to "t_18" at magnitude 0.5 gets converted into a call to CIF symbol #1 which is a completely replicated version of CIF symbol #2, reduced by half. This data is shown in the output CIF DBIF of Appendix N. One additional set of generic data not translated into CIF are the predicates which define text. These are not seen in the CIF output DBIF.

Appendix R shows the entire set of kept facts generated in this forward translation from CALMA to CIF. In addition to those kept facts shown previously in Figure 5.40, new generic data which couldn't be translated into CIF has been added. As indicated before, all text data has been "kept". Also, those predicates such as "magnif_", "ncall", "item_inst", "macro_inst", and "macro_call" which are the result of not having a CIF magnification feature have been "kept" as well. Note that the "map_" facts have also been kept to correlate CIF symbol numbers with generic macro definition names.

This completes the forward translation of the sample CALMA data into CIF. To add further validity to the experiment, the resulting CIF data was transported back into CALMA.

### 5.3.5 Reverse Translation Results

The rules needed to perform the reverse translation (CIF to CALMA) have been described in section 5.3.3. Applying these rules along with the CIF DBIF, created by the forward translation, the prototype translation engine yields the system output log shown in Appendix Q. The results of the first half of the translation were 200 generic facts. Comparing these with the 201 generic facts from the forward translation (Appendix M), shows that one "orient_" fact was lost. Inspection of the "orient_" facts shows that the fact lost was "orient_(i_8,0,0,0)". This is due to the "vtxfin" rule (marked /* V */) in the CIF input rules (Appendix K). This rule precludes a "zero" rotation (i.e., orient_(X,0,0,0)) from being written out into the generic set of data, since zero rotation has no effect on the end result. Consequently, the two sets of generic data are equivalent.

Note also that the generic data produced by the reverse translation (Appendix O) contains the magnification factors applied to the "macro_call"s for i_14. In fact, both i_7 and i_14 refer to t_18 in the generic data with i_14 having a magnification factor. However, in the CIF data, "mcall_1" through "mcall_4" (the CIF equivalent to i_14) call symbol #1 (the half-scale equivalent to t_18 in CIF). The kept data preserved the information necessary to know that the CIF data has a scaled-down version of t_18.

The second part of this reverse translation takes the generic data and produces the final CALMA output DBIF. The system log (Appendix Q) shows that 199 CALMA facts were generated. These are shown in Appendix P. The original CALMA DBIF contained 200 facts. The discrepancy can be explained.

First, the system log shows that 3 CALMA "mag" facts were generated from the generic data. The original CALMA source had only two "mag" facts. Inspection of the output CALMA data shows that two of the "mag" facts are the same. This anomaly was introduced during the forward translation when the single "aref" i_14 was converted to four generic "macro_call"s. The original magnification factor of 0.5 was attributed to each of the four "macro_call"s, and it was also stored along with i_14 as well (see Appendix M). This is a minor error in the original input rules for CALMA data. Specifically, rule 14 (CALMA to CIF rules Appendix J) is wrong. It says that all CALMA "mag(S,M)" facts should be translated into generic "magnif_(S,M)" facts. This rule should be modified to do so only if the magnification doesn't apply to an "aref". The error was left in the

rules to show the impact of such an oversight in the resulting output. Thus far, we have shown that the 199 output CALMA facts are actually only 198 facts (one was an extra "mag" fact).

The original set of CALMA data had 200 facts, therefore, there are two facts unaccounted for. Again, by inspection, there are two angle facts which were in the original CALMA data which have been lost during the translation. Two of the original angle facts (see Appendix I, `angle(i_8,0.0)` and `angle(i_14,90.0)`) are missing from the output; only `angle(i_32,0.0)` remains. The `i_32` is the identifier for the only CALMA text item. Since this was never translated into CIF, it was not removed from CIF by the input rule `/* V */` (Appendix K) which removes "zero" orientations. Instead, the zero orientation for `i_32` crept back into the generic data since it had been a part of the kept data. This explains why the zero orientation for `i_8` is missing: it was removed by the CIF input rule `/* V */`. Since it is a zero orientation, it makes little difference. However, the 90° rotation on `i_14` in the source CALMA data would seem to be an error. However, a closer look at the CALMA output vertices for `i_14` show three coordinates (see Appendix P): `xy(i_14,46000,3000,1)`, `xy(i_14,41000,3000,2)`, and `xy(i_14,46000,10250,3)`, where the second and third arguments to the `xy` predicate are the X and Y coordinates, respectively. These three vertices form a CALMA "aref", shown in Figure 5.41. The original source CALMA vertices for `i_14` were (see Appendix I): `xy(i_14,46000,3000,1)`, `xy(i_14,53250,3000,2)`, and `xy(i_14,46000,8000,3)`. Figure 5.41 shows that the output vertices are merely the result of a 90° rotation of the source CALMA vertices. Therefore, it is alright that `angle(i_14,90.0)` is missing from the output CALMA data, since the rotation was applied to the ver-

tices instead. Thus, the source and output CALMA data are equivalent.

This third example using the knowledge-based prototype system shows the feasibility of the methodology for the "physical" class of data discussed in Chapter 2. The methodology has proven feasible for two major classes of CAE/CAD/CAM data (logical and physical) and for several different source-target system combinations. It is clear from these examples that there are significant differences in the data models and data element representations which are used by the various CAE/CAD/CAM systems. These differences complicate the data base transport problem and sophisticated translation rules are required to accomplish the objective.



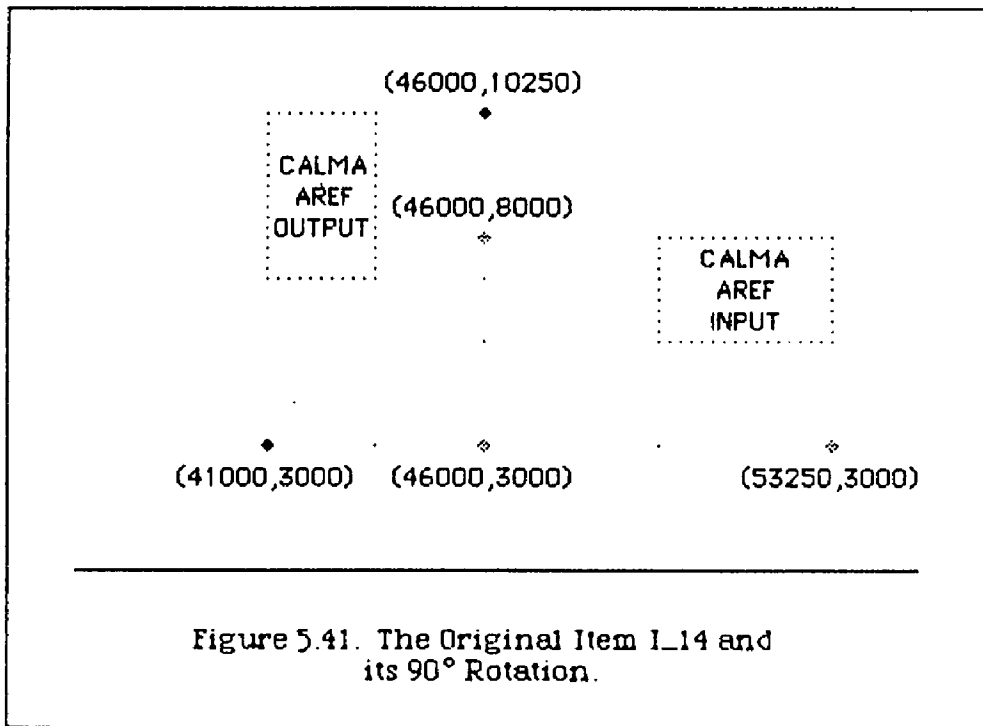


Figure 5.41. The Original Item L₁₄ and its 90° Rotation.

## CHAPTER 6

### CONCLUSION

The objective of this dissertation was to develop a method for transporting data bases between distinct types of electronic CAE/CAD/CAM systems. The need for this research is evident from

- 1) The numerous CAE/CAD/CAM systems in use today, both commercial and privately-owned;
- 2) The years spent by representatives of dozens of companies in trying to agree upon common data interface specifications (notably, IGES and EDIF); and
- 3) The lack of an industry-wide solution which has been accepted and put into production use.

Indeed, work continues at present toward standards for data base transport.

Toward this stated objective, this dissertation has provided new insight into the problem and has defined a methodology which has proven feasible. Specifically, the accomplishments are:

- 1) The definition of categories of CAE/CAD/CAM data,
- 2) An analysis of the difference in electronic CAE/CAD/CAM data

representation, content, and organization,

3) A presentation of examples of the difficulties in translating between data base types,

4) The definition of a data transport system architecture.

5) The construction of a prototype translation engine using a knowledge-based system approach,

6) The development of rules (knowledge-base) which contain the expertise on how to translate out of the master data schema and five native CAE/CAD/CAM data base types, and

7) The application of the prototype to three two-way transport test cases.

The system architecture developed consists of data base compilers, a data base intermediate format (DBIF), a master data schema defined upon generic predicates, a generic translation engine, a knowledge base of translation rules/expertise, and data base formatters. The system is adaptive and not limited to one class of CAE/CAD/CAM data. In order to introduce a new data base type into the system, new rules must be defined for the new data base only. Since a master data schema is utilized, the rules only need specify how to translate data between the master schema and the new data base. Consequently, rules are not needed for each source-destination data base pair.

One problem which has rendered previous data base transport approaches limited in their effectivity is the mismatch between the content of two distinct data base types. Previous translators have discarded any source data which couldn't be translated in the destination data representation. This is problematic when the data must be transported back to the originating system. The methodology defined herein addressed this problem (referred to as the *delta problem*), and provides a capability which stores this data for later use in reverse translation.

The system prototype was implemented using Prolog. This was a very natural application of the logic assertion style of Prolog. All of the rules necessary to translate between data base types were expressed as Prolog statements. The DBIF selected used the Prolog syntax which made the translation process straight-forward by utilizing the built-in Prolog inference engine to derive the destination data base. The test cases have shown that the original objective is feasible, and that artificial intelligence (knowledge-base/expert systems) is an integral part of the methodology.

In the course of developing the prototype and in performing the experiments with the test cases, several conclusions were drawn which may aid others in similar work.

- 1) Prolog is well-suited to this type of work, because rules can be developed incrementally with knowledge of how data should be represented in the various CAE/CAD/CAM systems.

- 2) C-Prolog (an interpreter) provides very helpful built-in debugging facilities, including traces of execution and break-points on logical expres-

sions encountered during execution.

3) The use of a set of generic predicates facilitates the transport of data by minimizing the amount of new rules which must be added when new data base formats are used. The proof of this was that no new rules were necessary for DR1 when translating into TDL instead of DR2.

4) The use of a neutral syntax such as the data base intermediate form (DBIF) described in this methodology facilitates data schema translation and verification.

5) The arithmetic/mathematic built-in functions of C-Prolog are quite adequate for most geometric transform needed.

One limitation to this work is the size of the data bases. The test cases were small enough to fit within the memory constraints of the C-Prolog interpreter. A full-scale Prolog compiler would no doubt provide faster processing and perhaps larger memory capacity.

There are a number of areas for future work in this area. One major concern is how to discern whether a data base has changed from the time it leaves its source system until it is transported back. For example, a CALMA data base could be translated into CIF, then modified, and then translated back into CALMA. It would be useful to develop rules which indicate how to detect the modification. This problem was investigated briefly in the midst of experimenting with the translation test cases. A set of change detection rules were written in Prolog, but additional effort was set aside as a future topic of investigation.

Another area for further work is the refinement of the generic predicate sets for the different classes of CAE/CAD/CAM data. Perhaps, a new set of generic predicates can be developed from the new EDIF standard once it is released. There is great promise that this standard will be very comprehensive as regards electronic CAE/CAD/CAM data entities. The LISP-like syntax of EDIF should be conducive to being expressed in Prolog.

Another concern is that there must be some criteria to determine whether a set of translation rules is complete with respect to all of the data elements of the native CAE/CAD/CAM data schema. Further work into this area needs to define desirable characteristics of native CAE/CAD/CAM data schema which will facilitate specifying translation rules.

The scope of this research was limited to the field of electronic CAE/CAD/CAM. The idea of rule-based data base translators may have application to other fields which have similar interchange requirements. As investigators encounter similar data interchange problems, it is hoped that this work will benefit them in their endeavors.

It is certain that this research provides further credibility as to the viability of artificial intelligence as a problem solving tool. Also, the ease of which CAE/CAD/CAM data is expressed in terms of logic constructs may broaden the application of artificial intelligence techniques to other CAE/CAD/CAM problem areas.

## APPENDIX A

### GLOSSARY

**ASCII:** American Standard Code for Information Interchange. This is a seven bit character code used to represent alphanumeric data. There are 128 unique characters which are represented with this code. It is also represented in an eight bit format which leaves the high-order bit either set or cleared by convention adopted by the computer manufacturer.

**ANSI:** American National Standards Institute. The agency responsible for setting standards pertaining to computer software (e.g., programming languages)

**CAD:** Computer-Aided Design. Refers to any automation support (i.e., tools) for any design activity. Originally, it pertained to all computer systems which were used by engineers. More recently, CAE (computer-aided engineering) refers to front-end simulation and analysis tools and logic capture. CAD has been reduced in scope to cover those tools which are used to assist in the physical mechanization of a design (e.g., layout tools: placement and route, design rule check, artwork generation, etc.)

**CAE:** Computer-Aided Engineering. Refers to front-end design tools such as logic (design) capture, logic simulation, and timing analysis. (Also,

see CAD.)

**CALMA:** A turn-key CAD system. A variety of models are available. A popular system for VLSI layout. No automatic placement or routing is available. All layout is performed manually, or by CALMA GPL programs. The main CAD system is GDS II. The CALMA Stream Format is an industry-recognized format for layout geometry description.

**CMOS:** Complementary Metal-Oxide Semiconductor. Several technologies exist for use in building integrated circuits, CMOS being one of these. Others include nMOS and bipolar technologies. Each technology has unique features that make it suitable for different operating environments and performance requirements.

**Computervision (CV):** A turn-key CAD system. This system supports a wide variety of CAD applications, including mechanical CAD, architectural design, piping, and electronics. CV is primarily a drafting tool; however, it does support some design automation tools. The local processing power restricts CV's usefulness to smaller-scale designs. New and more powerful CV hardware will become available in the near future.

**Database (DB):** A database can be anything from a structured collection of files which contain the data manipulated by a DBMS to a loose set of files which contains data. In CAD circles, the term has been used to cover the gamut. From a computer science perspective, a database is a collection of data which is organized into a physical struc-



ture which supports a logical view or model of the data. Traditionally, there are three recognized logical models for a database: hierarchical, network, and relational. In practice, many commercial CAE/CAD/CAM vendors refer to a conventional file as a database.

**Data Base Intermediate Format (DBIF)** - This is a Prolog-compatible representation of a CAE/CAD/CAM database, the form of which has been developed as part of this dissertation. All native syntax from the original system is removed and only the data content and schema is apparent in the DBIF.

**Data Base Management System (DBMS)** - A system which stores data in a way which facilitates the interaction with the data by users and programs according to a pre-specified logical view or data model.

**EBCDIC** - Extended Binary Coded Decimal International Code. This is an eight bit character code used to represent alphanumeric data. There are 256 unique characters which are represented with this code.

**ECL** - Emitter-Coupled Logic. A family of logic such as nMOS and CMOS.

**HDL** - Hardware Design Language. This is a language which is used to represent the logical and physical characteristics of a circuit. The language can be used to generate a CAE/CAD database and/or it can be used directly to drive various CAE/CAD tools, such as simulation and routing.

**IC** - Integrated Circuit. This is a circuit built entirely upon a single chip

(usually of silicon).

**IGES** - Initial Graphics Exchange Specification. This was a cooperative effort to define an industry standard for the exchange of graphics CAD data between distinct CAD system types.

**IPC** - Institute of Printed Circuits. This agency has defined standards for the description of physical characteristics which define printed circuit board.

**KBS** - Knowledge Based System. A system different from conventional, procedural systems which are driven by logic embedded within the program. A KBS has a generalized inference engine which can be used to derive facts based upon the content of a database of knowledge (knowledge base).

**NAND** - One of the Boolean functions of two variables. This function returns true if both inputs are false.

**NMOS** - N-channel Metal Oxide Semiconductor. A family of logic such as ECL and CMOS.

**PCB** - Printed Circuit Board. This is a means of building an electronic circuit whereby the interconnections between components are implemented as copper lines sandwiched between layers of non-conductive material. The copper lines are formed by etching into a copper sheet that has been laminated onto the non-conductive material.

**PROLOG** - A programming language which is used for logic programming and artificial intelligence applications.

PWB - Printed Wiring Board. This is another term for PCB.

VLSI - Very Large Scale Integration. Refers to the highest degree of density in electronic circuitry attainable at the present. The density is evident in the number of logic elements contained on a single chip and in the small size of interconnect lines.

## APPENDIX B

### ALTERNATIVE PROTOTYPE IMPLEMENTATIONS

In the course of developing this methodology for data base transport, a system block diagram was developed and analyzed long before Prolog was selected as a means for implementation. In fact, since the topic of research dealt with data, the first implementation approach attempted was a to use a relational DBMS with application programs built upon it to provide the necessary functionality.

For example, the first DBIF selected was a relation with domains "ID", "Predecessor", "Key", "Field", and "Value". The first encoding of DR2 is shown in Figure B.1. Note that there is already a problem since, it is not clear what data type should be given to the domain "Value". Each different value of "Field" will have a different data type associated with the domain "Value". For example, the data type for "Value" should be "text" if the value of "Field" is (Body) or (Node); whereas the data type should be "coordinate" if the value of "Field" is (Nodexy).

Another use of a relational data base for the prototype was a pair of relations "Key Table" and "Translate Table", used to store translation rules. The relation "Translate Table" was used to store logical relationships between data items which were stored in "Key Table". Actual tables were defined using the INGRES DBMS and rudimentary DR1-DR2 transla-

ID	Pred	Key	Field	Value	ID	Pred	Key	Field	Value
1	0	12	(Body)	A	24	23	15	(Signal)	S1
2	1	13	(Node)	R	25	24	16	(Sigxy)	(0,9)
3	2	14	(Nodexy)	(1,9)	26	25	16	(Sigxy)	(1,9)
4	3	13	(Node)	S	27	26	15	(Signal)	S2
5	4	14	(Nodexy)	(1,8)	28	27	16	(Sigxy)	(0,8)
6	5	13	(Node)	Q	29	28	16	(Sigxy)	(1,8)
7	6	14	(Nodexy)	(2,9)	30	29	15	(Signal)	S3
8	7	13	(Node)	QN	31	30	16	(Sigxy)	(1,2)
9	8	14	(Nodexy)	(2,8)	32	31	16	(Sigxy)	(3,2)
10	9	12	(Body)	B	33	32	15	(Signal)	S4
11	10	13	(Node)	I1	34	33	16	(Sigxy)	(2,9)
12	11	14	(Nodexy)	(3,4)	35	34	16	(Sigxy)	(5,8)
13	12	13	(Node)	I2	36	35	15	(Signal)	S5
14	13	14	(Nodexy)	(3,2)	37	36	16	(Sigxy)	(2,8)
15	14	13	(Node)	Y	38	37	16	(Sigxy)	(2,5,6)
16	15	14	(Nodexy)	(4,3)	39	38	16	(Sigxy)	(5,6)
17	16	12	(Body)	C	40	39	16	(Sigxy)	(3,4)
18	17	13	(Node)	I1	41	40	15	(Signal)	S6
19	18	14	(Nodexy)	(5,8)	42	41	16	(Sigxy)	(6,7)
20	19	13	(Node)	I2	43	42	16	(Sigxy)	(7,7)
21	20	14	(Nodexy)	(5,6)	44	43	15	(Signal)	S7
22	21	13	(Node)	Y	45	44	16	(Sigxy)	(4,3)
23	22	14	(Nodexy)	(6,7)	46	45	16	(Sigxy)	(6,3)

Figure B.1. Original DR2 Representation as a Relation.

tion rules were stored. The intent was to store facts such as "data time A from system 1 translates into data item B from system 2".

As the research continued into specific data relationships between systems, it became obvious that translation would not be one-to-one. In fact, some translations are conditional, and there were no facilities to store this kind of information. Stonebraker and Keller propose extensions to conventional DBMS's which make this more feasible [Ston80]. In short, it became obvious that to solve the CAE/CAD/CAM data transport problem with a relational DBMS was not a good match.

Not only was it difficult to express translation knowledge into a relation, but the queries to extract out translated data were difficult to formulate. Also, there was no built-in mechanism for reasoning (e.g. an inference engine). This would need to have been built. At that point, the process of developing the prototype was severely impacted and the approach was dropped in favor of using Prolog.

**APPENDIX C**  
**CALMA STREAM FORMAT DATA**

Dump of file _DUA0:[RPH.MLC]STREAM.3;1 on  
 3-OCT-1984 14:12:21.53  
 File ID (10533,14,0) End of file block 8 / Allocated 9

Virtual block number 1 (00000001), 512 (0200) bytes

```

1A000A00 53000201 1C000400 02000600 .....S.... 000000 H,BL
05000B00 1A000A00 53000000 00000000 .....S..... 000010
42442E4E 594C4952 414D0602 0E000000 .....MARILYN.DB 000020 L
542E544E 4F463A49 49534447 0620B400) .. .GDSII:FONT.T 000030 F
00000000 00000000 00000000 0000005B X..... 000040
00000000 00000000 00000000 00000000 ..... 000050
0000005B 542E544E 4F463A49 49534447 GDSII:FONT.TX... 000060
00000000 00000000 00000000 00000000 ..... 000070
49534447 00000000 00000000 00000000 .....GDSI 000080
00000000 0000005B 542E544E 4F463A49 I:FONT.TX..... 000090
00000000 00000000 00000000 00000000 ..... 0000A0
4F463A49 49534447 00000000 00000000 .....GDSII:FD 0000B0
00000000 00000000 0000005B 542E544E NT.TX..... 0000C0
00000000 00000000 00000000 00000000 ..... 0000D0
413E0503 14000300 02220600) 00000000 .....>A 0000E0 G,U
1C00105A 9BA02FB8 4439EFA7 C64B3789 .7K...9D./..Z... 0000F0 BS
53002100 04000B00 1A000A00 53000205 ...S.....!S 000100
414D0606 10002500 04000B00 1A000A00 .....%....MA 000110 S
06000008 04000056 45445F4E 594C4952 RILYN_DEV..... 000120 B
51000310 2C000000 020E0600) 2800020D ...(......Q 000130 LY,DT,XY
4700DB33 FEFFAB24 51004084 0400AB24 $...@*...3..G 000140
51004084 0400A059 4700DB33 FEFFA059 Y...3..GY....@.Q 000150
06000008 04000011 04004084 0400AB24 $...@..... 000160 EL,B
4F000310 2C000000 020E0600) 0100020D .....D 000170 LY,DT,XY
47000000 0000805B 4F004084 0400805B X...@.DX....G 000180
4F004084 0400A059 47000000 0000A059 Y.....GY....@.Q 000190
06000008 04000011 04004084 0400805B X...@..... 0001A0 EL,B
51000310 2C000000 020E0600) 0100020D .....Q 0001B0 LY,DT,XY
4700DB33 FEFFAB24 51004084 0400AB24 $...@*$...3..G 0001C0
51004084 0400A059 4700DB33 FEFFA059 Y...3..GY....@.Q 0001D0
06000008 04000011 04004084 0400AB24 $...@..... 0001E0 EL,B
4C000310 2C000000 020E0600) 0A00020D .....L 0001F0 LY,DT,XY

```

LEGEND

B = BOUNDARY	H = HEADER	SN = SNAME
BL = BGNLIB	L = LIBNAME	SR = SREF
BS = BGNSTR	LY = LAYER	ST = STRANS
DT = DATATYPE	MG = MAGNIFICATION	T = TEXT
EL = ENDEL	P = PATH	TT = TEXTTYPE
ES = ENDSTR	PR = PRESENTATION	U = UNITS
F = FONTS	S = STRNAME	W = WIDTH
G = GENERATIONS	SG = STRING	XY = XY COORD



Dump of file _DUA0:[RPH.MLC]STREAM.3;1 on  
 3-OCT-1984 14:12:21.53  
 File ID (10533,14,0) End of file block 8 / Allocated 9

Virtual block number 2 (00000002), 512 (0200) bytes

```

4E0040B4 040040BC 4E0040B4 040080E7 .....@.N.@...@.N 000000
4C00C045 040080E7 4C00C045 040040BC .@..E..L....E..L 000010
0600000B 04000011 040040B4 040080E7 .....@..... 000020 EL,B
4C000310 2C000000 020E0600 0A00020D .....L 000030 LY,DT,XY
4E00409C 000040BC 4E00409C 000080E7 .....@.N.@...@.N 000040
4C000071 020080E7 4C000071 020040BC .@..q..L....q..L 000050
1000000A 04000011 0400409C 000080E7 .....@..... 000060 EL,SR,SN
0600003B 31545F4E 594C4952 414D0612 ..MARILYN_T1B... 000070
(00000000 40694700 03100C00 0000011A .....Gi@.... 000080 ST,XY
06000A00 020D0600 00090400 00110400 ..... 000090 EL,P,LY
03101400 9B3A0000 030F0800 0000020E ..... 0000A0 DT,W,XY
(C0450400 C0A14800 40B40400 C0A14800 .H.....@.H....E. 0000B0
06000A00 020D0600 00090400 00110400 ..... 0000C0 EL,P,LY
03102C00 9B3A0000 030F0800 0000020E ..... 0000D0 DT,W,XY
B0A90300 E0D14D00 C0450400 E0D14D00 .M....E..M..... 0000E0
80380100 404B4C00 B0A90300 404B4C00 .LK@....LK@..B. 0000F0
(00090400 00110400 80380100 00AF4B00 .K....B..... 000100 EL,P
030F0800 0000020E 06000A00 020D0600 ..... 000110 LY,DT,W
409C0000 E0D14D00 03102400 9B3A0000 .....$...M.....@ 000120 XY
00000000 C0A14800 00000000 E0D14D00 .M.....H..... 000130
(00080400 00110400 409C0000 C0A14800 .H.....@..... 000140 EL,B
03105C00 0000020E 06000300 020D0600 .....\.. 000150 LY,DT,XY
04170300 AB245100 40B40400 AB245100 .Q$....@.Q$.... 000160
FCCA0100 3C344900 04170300 3C344900 .I4<....I4<.... 000170
04170300 44B04A00 FCCA0100 44B04A00 .J.D....J.D.... 000180
DB33FEFF AB245100 04170300 AB245100 .Q$....Q$...3. 000190
40B40400 A0594700 DB33FEFF A0594700 .GY...3..GY...@ 0001A0
(00080400 00110400 40B40400 AB245100 .Q$....@..... 0001B0 EL,B
03102C00 0000020E 06000B00 020D0600 ..... 0001C0 LY,DT,XY
FCCA0100 440F4800 FCCA0100 3C344900 .I4<....H.D.... 0001D0
04A60000 3C344900 04A60000 440F4800 .H.D....I4<.... 0001E0
(00080400 00110400 FCCA0100 3C344900 .I4<..... 0001F0 EL,B

```

LEGEND

B = BOUNDARY	H = HEADER	SN = SNAME
BL = BGNLIB	L = LIBNAME	SR = SREF
BS = BGNSTR	LY = LAYER	ST = STRANS
DT = DATATYPE	MG = MAGNIFICATION	T = TEXT
EL = ENDEL	P = PATH	TT = TEXTTYPE
ES = ENDSTR	PR = PRESENTATION	U = UNITS
F = FONTS	S = STRNAME	W = WIDTH
G = GENERATIONS	SG = STRING	XY = XY COORD

Dump of file _DUA0:[RPH.MLC]STREAM.3;1 on  
 3-OCT-1984 14:12:21.53  
 File ID (10533,14,0) End of file block 8 / Allocated 9

Virtual block number 3 (00000003), 512 (0200) bytes

```

03102C00 (0000020E 06000B00 020D0600) ..... 000000 LY,DT,XY
FCCA0100 44B04A00 FCCA0100 3CA54B00 .K.<.....J.D.... 000010
04A60000 3CA54B00 04A60000 44B04A00 .J.D.....K.<.... 000020
(000B0400) 00110400 FCCA0100 3CA54B00 .K.<..... 000030 EL,B
03102C00 (0000020E 06000B00 020D0600) ..... 000040 LY,DT,XY
3C670200 44F14C00 3C670200 7CB24E00 .N.!..g<.L.D..g< 000050
04A60000 7CB24E00 04A60000 44F14C00 .L.D.....N.!... 000060
(000B0400) 00110400 3C670200 7CB24E00 .N.!..g<..... 000070 EL,B
03102C00 (0000020E 06000B00 020D0600) ..... 000080 LY,DT,XY
B44F0400 7CB24E00 40B40400 7CB24E00 .N.!...@.N.!..D. 000090
40B40400 44F14C00 B44F0400 44F14C00 .L.D...D..L.D...@ 0000A0
(000B0400) 00110400 40B40400 7CB24E00 .N.!...@..... 0000B0 EL,B
03102C00 (0000020E 06000B00 020D0600) ..... 0000C0 LY,DT,SY
FC3B0400 440F4800 FC3B0400 3C344900 .I4<...;..H.D...; 0000D0
04170300 3C344900 04170300 440F4800 .H.D.....I4<.... 0000E0
(000B0400) 00110400 FC3B0400 3C344900 .I4<...;..... 0000F0 EL,B
03102C00 (0000020E 06000B00 020D0600) ..... 000100 LY,DT,XY
FC3B0400 44B04A00 FC3B0400 3CA54B00 .K.<...;..J.D...; 000110
04170300 3CA54B00 04170300 44B04A00 .J.D.....K.<.... 000120
(00070400) 00110400 FC3B0400 3CA54B00 .K.<...;..... 000130 EL,ES
26000200 0B001A00 0A005300 02051C00) .....S.....& 000140 BS
06061000 (1C000400 0B001A00 0A005300 .S..... 000150 S
(000B0400) 003B3154 5F4E594C 4952414D MARILYN_T1B.... 000160 B
03102C00 (0000020E 06000A00 020D0600) ..... 000170 LY,DT,XY
C0450400 C0450400 C0450400 400D0300 ...@..E...E...E. 000180
400D0300 400D0300 400D0300 C0450400 ..E....@...@...@ 000190
(000B0400) 00110400 C0450400 400D0300 ...@..E..... 0001A0 EL,B
03107C00 (0000020E 06000A00 020D0600) .....!.. 0001B0 LY,DT,XY
409C0000 C0D40100 409C0000 409C0000 ...@...@.....@ 0001C0
C0D40100 CC550100 C0D40100 C0D40100 .....U..... 0001D0
B4530200 1C190200 B4530200 CC550100 ..U...S.....S. 0001E0
5B0F0200 2C400200 5B0F0200 1C190200 .....X...@,...X 0001F0

```

LEGEND

B = BOUNDARY	H = HEADER	SN = SNAME
BL = BGNLIB	L = LIBNAME	SR = SREF
BS = BGNSTR	LY = LAYER	ST = STRANS
DT = DATATYPE	MG = MAGNIFICATION	T = TEXT
EL = ENDEL	P = PATH	TT = TEXTTYPE
ES = ENDSTR	PR = PRESENTATION	U = UNITS
F = FONTS	S = STRNAME	W = WIDTH
G = GENERATIONS	SG = STRING	XY = XY COORD

Dump of file _DUA0:[RPH.MLC]STREAM.3;1 on 3-OCT-1984 14:12:21.53  
 File ID (10533,14,0) End of file block B / Allocated 9

Virtual block number 4 (00000004), 512 (0200) bytes

```

ABD20200 1C190200 ABD20200 2C400200 ...@,..... 000000
4C8E0200 341B0100 4C8E0200 1C190200 .....L...4...L 000010
COD40100 409C0000 COD40100 341B0100 ...4.....@.... 000020
(00080400)(00110400)(409C0000 409C0000 ...@...@..... 000030 EL, B
03107C00 0000020E 06000A00 020D0600) .....i... 000040 LY, DT, XY
409C0000 C0450400 409C0000 400D0300 ...@...@..E...@ 000050
COD40100 CCC60300 COD40100 C0450400 ...E..... 000060
4C8E0200 E4C80200 4C8E0200 CCC60300 .....L.....L 000070
ABD20200 D4A10200 ABD20200 E4C80200 ..... 000080
580F0200 E4C80200 580F0200 D4A10200 .....X.....X 000090
B4530200 348C0300 B4530200 E4C80200 .....S....4..S. 0000A0
COD40100 400D0300 COD40100 348C0300 ...4.....@.... 0000B0
(00080400)(00110400)(409C0000 400D0300 ...@...@..... 0000C0 EL, B
03102C00 0000020E 06000200 020D0600) ..... 0000D0 LY, DT, XY
D0FB0100 785D0200 D0FB0100 94050200 .....lx.... 0000E0
30E60200 94050200 30E60200 785D0200 ...lx...0.....0 0000F0
(00080400)(00110400)(D0FB0100 94050200 ..... 000100 EL, B
03102C00 0000020E 06000200 020D0600) ..... 000110 LY, DT, XY
D0FB0100 6CDC0200 D0FB0100 88B40200 .....l.... 000120
30E60200 88B40200 30E60200 6CDC0200 ...l...0.....0 000130
(00080400)(00110400)(D0FB0100 88B40200 ..... 000140 EL, B
03105C00 0000020E 06000A00 020D0600) .....\. 000150 LY, DT, XY
C0450400 COD40100 C0450400 409C0000 ...@..E.....E. 000160
F8950300 4C8E0200 F8950300 COD40100 .....L.... 000170
38C10100 B4530200 38C10100 4C8E0200 ...L...B..S....B 000180
DB470300 COD40100 DB470300 B4530200 ...S...G.....G. 000190
400D0300 409C0000 400D0300 COD40100 .....@...@...@ 0001A0
(00080400)(00110400)(C0450400 409C0000 ...@..E..... 0001B0 EL, B
03102C00 0000020E 06000400 020D0600) ..... 0001C0 LY, DT, XY
94050200 F0490200 94050200 580F0200 ...X.....I..... 0001D0
6CDC0200 580F0200 6CDC0200 F0490200 ..I...l...X...l 0001E0
(00080400)(00110400)(94050200 580F0200 ...X..... 0001F0 EL, B

```

LEGEND

B = BOUNDARY	H = HEADER	SN = SNAME
BL = BGNLIB	L = LIBNAME	SR = SREF
BS = BGNSTR	LY = LAYER	ST = STRANS
DT = DATATYPE	MG = MAGNIFICATION	T = TEXT
EL = ENDEL	P = PATH	TT = TEXTTYPE
ES = ENDSTR	PR = PRESENTATION	U = UNITS
F = FONTS	S = STRNAME	W = WIDTH
G = GENERATIONS	SG = STRING	XY = XY COORD

Dump of file _DUA0:[RPH.MLC]STREAM.3;1 on  
 3-OCT-1984 14:12:21.53  
 File ID (10533,14,0) End of file block 8 / Allocated 9

Virtual block number 5 (00000005), 512 (0200) bytes

```

03102C00)0000020E 06000400 020D0600)..... 000000 LY,DT,XY
94050200 ABD20200 94050200 109B0200 ..... 000010
6CDC0200 109B0200 6CDC0200 ABD20200 .....1.....1 000020
(000B0400)00110400)94050200 109B0200 ..... 000030 EL,B
03102C00)0000020E 06000800 020D0600)..... 000040 LY,DT,XY
1C190200 68360200 1C190200 E0220200 ..".....6h.... 000050
E4CB0200 E0220200 E4CB0200 68360200 ..6h....."..... 000060
000B0400)00110400)1C190200 E0220200 .."..... 000070 EL,B
03102C00)0000020E 06000800 020D0600)..... 000080 LY,DT,XY
1C190200 20BF0200 1C190200 98AB0200 ..... 000090
E4CB0200 98AB0200 E4CB0200 20BF0200 ... .. 0000A0
(000B0400)00110400)1C190200 98AB0200 ..... 0000B0 EL,B
03102C00)0000020E 06000800 020D0600)..... 0000C0 LY,DT,XY
FCCA0100 04170300 FCCA0100 FCCA0100 ..... 0000D0
04170300 FCCA0100 04170300 04170300 ..... 0000E0
(000B0400)00110400)FCCA0100 FCCA0100 ..... 0000F0 EL,B
03102C00)00000216 06001000 020D0600)..... 000100 LY,DT,XY
04170300 FC3B0400 04170300 CB200300 .. ..;..... 000110
FC3B0400 CB200300 FC3B0400 FC3B0400 ..;...;... ..; 000120
000C0400)00110400)04170300 CB200300 .. .. 000130 EL,T
01170600)00000216 06001000 020D0600)..... 000140 LY,TT,PR
00004041 051B0C00)0000011A 06000500 .....A..... 000150 ST,MG
CCC60300 F0490200 03100C00)00000000 .....I..... 000160 XY
(000B0400)00110400)0D383154 06190800)....T18..... 000170 SG,EL,B
03102C00)0000020E 06000400 020D0600)..... 000180 LY,DT,XY
CB200300 38320400 CB200300 BC2A0300 ..*... ..2B.. 000190
38320400 BC2A0300 38320400 38320400 ..2B..2B..*...2B 0001A0
(00070400)00110400)CB200300 BC2A0300 ..*... .. 0001B0 EL,ES
00000000 00000000 00000000 (00040400)..... 0001C0 ENDLIB
00000000 00000000 00000000 00000000 ..... 0001D0
00000000 00000000 00000000 00000000 ..... 0001E0
00000000 00000000 00000000 00000000 ..... 0001F0

```

LEGEND

B = BOUNDARY	H = HEADER	SN = SNAME
BL = BGNLIB	L = LIBNAME	SR = SREF
BS = BGNSTR	LY = LAYER	ST = STRANS
DT = DATATYPE	MG = MAGNIFICATION	T = TEXT
EL = ENDEL	P = PATH	TT = TEXTTYPE
ES = ENDSTR	PR = PRESENTATION	U = UNITS
F = FONTS	S = STRNAME	W = WIDTH
G = GENERATIONS	SG = STRING	XY = XY COORD

**APPENDIX D**  
**DBIF REPRESENTING CALMA STREAM FORMAT**

```

IS_A (gdsII.library, 1%)
ATTR ( 1%, libname, MARILYN.DB)
ATTR ( 1%, font definition, GDSII:FONT.TX)
ATTR ( 1%, font definition, GDSII:FONT.TX)
ATTR ( 1%, font definition, GDSII:FONT.TX)
ATTR ( 1%, font definition, GDSII:FONT.TX)
ATTR ( 1%, generations, 3)
ATTR ( 1%, database units/user units, 0.10000002D-02)
ATTR ( 1%, database units/meter, 0.00000000D+00)
IS_A (gdsII.structure, 2%)
ATTR ( 2%, sname, MARILYN_DEV )
IS_A (gdsII.boundary, 3%)
HAS ( 2%, 3%)
LAYER ( 3%, 40)
ATTR ( 3%, datatype, 0)
VERTEX ( 3%, < 5317800, 296000>, 1)
VERTEX ( 3%, < 5317800, -117800>, 2)
VERTEX ( 3%, < 4676000, -117800>, 3)
VERTEX ( 3%, < 4676000, 296000>, 4)
VERTEX ( 3%, < 5317800, 296000>, 5)
IS_A (gdsII.boundary, 4%)
HAS ( 2%, 4%)
LAYER ( 4%, 1)
ATTR ( 4%, datatype, 0)
VERTEX ( 4%, < 5200000, 296000>, 1)
VERTEX ( 4%, < 5200000, 0>, 2)
VERTEX ( 4%, < 4676000, 0>, 3)
VERTEX ( 4%, < 4676000, 296000>, 4)
VERTEX ( 4%, < 5200000, 296000>, 5)
IS_A (gdsII.boundary, 5%)
HAS ( 2%, 5%)
LAYER ( 5%, 1)
ATTR ( 5%, datatype, 0)
VERTEX ( 5%, < 5317800, 296000>, 1)
VERTEX ( 5%, < 5317800, -117800>, 2)
VERTEX ( 5%, < 4676000, -117800>, 3)
VERTEX ( 5%, < 4676000, 296000>, 4)
VERTEX ( 5%, < 5317800, 296000>, 5)
IS_A (gdsII.boundary, 6%)
HAS ( 2%, 6%)
LAYER ( 6%, 10)
ATTR ( 6%, datatype, 0)
VERTEX ( 6%, < 5040000, 296000>, 1)
VERTEX ( 6%, < 5160000, 296000>, 2)
VERTEX ( 6%, < 5160000, 280000>, 3)
VERTEX ( 6%, < 5040000, 280000>, 4)
VERTEX ( 6%, < 5040000, 296000>, 5)
IS_A (gdsII.boundary, 7%)
HAS ( 2%, 7%)

```

```

LAYER ( 7%, 10)
ATTR ( 7%, datatype, 0)
VERTEX ( 7%, < 5040000, 40000>, 1)
VERTEX ( 7%, < 5160000, 40000>, 2)
VERTEX ( 7%, < 5160000, 160000>, 3)
VERTEX ( 7%, < 5040000, 160000>, 4)
VERTEX ( 7%, < 5040000, 40000>, 5)
IS_A (gdsII.sref, 8%)
HAS ( 2%, 8%)
ATTR ( 8%, sname, MARILYN_T18 )
VERTEX ( 8%, < 4680000, 0>, 1)
ORIENT ( 8%, 0.00, 0.00, 0.00)
IS_A (gdsII.path, 9%)
HAS ( 2%, 9%)
LAYER ( 9%, 10)
ATTR ( 9%, datatype, 0)
WIDTH ( 9%, 15000)
VERTEX ( 9%, < 4760000, 296000>, 1)
VERTEX ( 9%, < 4760000, 280000>, 2)
IS_A (gdsII.path, 10%)
HAS ( 2%, 10%)
LAYER ( 10%, 10)
ATTR ( 10%, datatype, 0)
WIDTH ( 10%, 15000)
VERTEX ( 10%, < 5100000, 280000>, 1)
VERTEX ( 10%, < 5100000, 240000>, 2)
VERTEX ( 10%, < 5000000, 240000>, 3)
VERTEX ( 10%, < 5000000, 80000>, 4)
VERTEX ( 10%, < 4960000, 80000>, 5)
IS_A (gdsII.path, 11%)
HAS ( 2%, 11%)
LAYER ( 11%, 10)
ATTR ( 11%, datatype, 0)
WIDTH ( 11%, 15000)
VERTEX ( 11%, < 5100000, 40000>, 1)
VERTEX ( 11%, < 5100000, 0>, 2)
VERTEX ( 11%, < 4760000, 0>, 3)
VERTEX ( 11%, < 4760000, 40000>, 4)

```

```

IS_A (gdsII.boundary, 12%)
HAS ( 2%, 12%)
LAYER ( 12%, 3)
ATTR ( 12%, datatype, 0)
VERTEX ( 12%, < 5317800, 296000>, 1)
VERTEX ( 12%, < 5317800, 202500>, 2)
VERTEX ( 12%, < 4797500, 202500>, 3)
VERTEX ( 12%, < 4797500, 117500>, 4)
VERTEX ( 12%, < 4882500, 117500>, 5)
VERTEX ( 12%, < 4882500, 202500>, 6)
VERTEX ( 12%, < 5317800, 202500>, 7)
VERTEX ( 12%, < 5317800, -117800>, 8)
VERTEX ( 12%, < 4676000, -117800>, 9)
VERTEX ( 12%, < 4676000, 296000>, 10)
VERTEX ( 12%, < 5317800, 296000>, 11)
IS_A (gdsII.boundary, 13%)
HAS ( 2%, 13%)
LAYER ( 13%, 11)
ATTR ( 13%, datatype, 0)
VERTEX ( 13%, < 4797500, 117500>, 1)
VERTEX ( 13%, < 4722500, 117500>, 2)
VERTEX ( 13%, < 4722500, 42500>, 3)
VERTEX ( 13%, < 4797500, 42500>, 4)
VERTEX ( 13%, < 4797500, 117500>, 5)
IS_A (gdsII.boundary, 14%)
HAS ( 2%, 14%)
LAYER ( 14%, 11)
ATTR ( 14%, datatype, 0)
VERTEX ( 14%, < 4957500, 117500>, 1)
VERTEX ( 14%, < 4882500, 117500>, 2)
VERTEX ( 14%, < 4882500, 42500>, 3)
VERTEX ( 14%, < 4957500, 42500>, 4)
VERTEX ( 14%, < 4957500, 117500>, 5)
IS_A (gdsII.boundary, 15%)
HAS ( 2%, 15%)
LAYER ( 15%, 11)
ATTR ( 15%, datatype, 0)
VERTEX ( 15%, < 5157500, 157500>, 1)
VERTEX ( 15%, < 5042500, 157500>, 2)
VERTEX ( 15%, < 5042500, 42500>, 3)
VERTEX ( 15%, < 5157500, 42500>, 4)
VERTEX ( 15%, < 5157500, 157500>, 5)
IS_A (gdsII.boundary, 16%)
HAS ( 2%, 16%)
LAYER ( 16%, 11)
ATTR ( 16%, datatype, 0)
VERTEX ( 16%, < 5157500, 296000>, 1)
VERTEX ( 16%, < 5157500, 282500>, 2)
VERTEX ( 16%, < 5042500, 282500>, 3)
VERTEX ( 16%, < 5042500, 296000>, 4)
VERTEX ( 16%, < 5157500, 296000>, 5)

```



```

IS_A (gdsII.boundary, 17%)
HAS ( 2%, 17%)
LAYER ( 17%, 11)
ATTR ( 17%, datatype, 0)
VERTEX ( 17%, < 4797500, 277500>, 1)
VERTEX ( 17%, < 4722500, 277500>, 2)
VERTEX ( 17%, < 4722500, 202500>, 3)
VERTEX ( 17%, < 4797500, 202500>, 4)
VERTEX ( 17%, < 4797500, 277500>, 5)
IS_A (gdsII.boundary, 18%)
HAS ( 2%, 18%)
LAYER ( 18%, 11)
ATTR ( 18%, datatype, 0)
VERTEX ( 18%, < 4957500, 277500>, 1)
VERTEX ( 18%, < 4882500, 277500>, 2)
VERTEX ( 18%, < 4882500, 202500>, 3)
VERTEX ( 18%, < 4957500, 202500>, 4)
VERTEX ( 18%, < 4957500, 277500>, 5)
IS_A (gdsII.structure, 19%)
ATTR ( 19%, sname, MARILYN_T18 )
IS_A (gdsII.boundary, 20%)
HAS ( 19%, 20%)
LAYER ( 20%, 10)
ATTR ( 20%, datatype, 0)
VERTEX ( 20%, < 200000, 280000>, 1)
VERTEX ( 20%, < 280000, 280000>, 2)
VERTEX ( 20%, < 280000, 200000>, 3)
VERTEX ( 20%, < 200000, 200000>, 4)
VERTEX ( 20%, < 200000, 280000>, 5)
IS_A (gdsII.boundary, 21%)
HAS ( 19%, 21%)
LAYER ( 21%, 10)
ATTR ( 21%, datatype, 0)
VERTEX ( 21%, < 40000, 40000>, 1)
VERTEX ( 21%, < 120000, 40000>, 2)
VERTEX ( 21%, < 120000, 120000>, 3)
VERTEX ( 21%, < 87500, 120000>, 4)
VERTEX ( 21%, < 87500, 152500>, 5)
VERTEX ( 21%, < 137500, 152500>, 6)
VERTEX ( 21%, < 137500, 135000>, 7)
VERTEX ( 21%, < 147500, 135000>, 8)
VERTEX ( 21%, < 147500, 185000>, 9)
VERTEX ( 21%, < 137500, 185000>, 10)
VERTEX ( 21%, < 137500, 167500>, 11)
VERTEX ( 21%, < 72500, 167500>, 12)
VERTEX ( 21%, < 72500, 120000>, 13)
VERTEX ( 21%, < 40000, 120000>, 14)
VERTEX ( 21%, < 40000, 40000>, 15)

```

```

IS_A (gdsII.boundary, 22%)
HAS ( 19%, 22%)
LAYER ( 22%, 10)
ATTR ( 22%, datatype, 0)
VERTEX ( 22%, < 200000, 40000>, 1)
VERTEX ( 22%, < 280000, 40000>, 2)
VERTEX ( 22%, < 280000, 120000>, 3)
VERTEX ( 22%, < 247500, 120000>, 4)
VERTEX ( 22%, < 247500, 167500>, 5)
VERTEX ( 22%, < 182500, 167500>, 6)
VERTEX ( 22%, < 182500, 185000>, 7)
VERTEX ( 22%, < 172500, 185000>, 8)
VERTEX ( 22%, < 172500, 135000>, 9)
VERTEX ( 22%, < 182500, 135000>, 10)
VERTEX ( 22%, < 182500, 152500>, 11)
VERTEX ( 22%, < 232500, 152500>, 12)
VERTEX ( 22%, < 232500, 120000>, 13)
VERTEX ( 22%, < 200000, 120000>, 14)
VERTEX ( 22%, < 200000, 40000>, 15)
IS_A (gdsII.boundary, 23%)
HAS ( 19%, 23%)
LAYER ( 23%, 2)
ATTR ( 23%, datatype, 0)
VERTEX ( 23%, < 132500, 130000>, 1)
VERTEX ( 23%, < 155000, 130000>, 2)
VERTEX ( 23%, < 155000, 190000>, 3)
VERTEX ( 23%, < 132500, 190000>, 4)
VERTEX ( 23%, < 132500, 130000>, 5)
IS_A (gdsII.boundary, 24%)
HAS ( 19%, 24%)
LAYER ( 24%, 2)
ATTR ( 24%, datatype, 0)
VERTEX ( 24%, < 165000, 130000>, 1)
VERTEX ( 24%, < 187500, 130000>, 2)
VERTEX ( 24%, < 187500, 190000>, 3)
VERTEX ( 24%, < 165000, 190000>, 4)
VERTEX ( 24%, < 165000, 130000>, 5)

```

```

IS_A (gdsII.boundary, 25%)
HAS ( 19%, 25%)
LAYER ( 25%, 10)
ATTR ( 25%, datatype, 0)
VERTEX ( 25%, < 40000, 280000>, 1)
VERTEX ( 25%, < 120000, 280000>, 2)
VERTEX ( 25%, < 120000, 235000>, 3)
VERTEX ( 25%, < 167500, 235000>, 4)
VERTEX ( 25%, < 167500, 115000>, 5)
VERTEX ( 25%, < 152500, 115000>, 6)
VERTEX ( 25%, < 152500, 215000>, 7)
VERTEX ( 25%, < 120000, 215000>, 8)
VERTEX ( 25%, < 120000, 200000>, 9)
VERTEX ( 25%, < 40000, 200000>, 10)
VERTEX ( 25%, < 40000, 280000>, 11)
IS_A (gdsII.boundary, 26%)
HAS ( 19%, 26%)
LAYER ( 26%, 4)
ATTR ( 26%, datatype, 0)
VERTEX ( 26%, < 135000, 132500>, 1)
VERTEX ( 26%, < 150000, 132500>, 2)
VERTEX ( 26%, < 150000, 187500>, 3)
VERTEX ( 26%, < 135000, 187500>, 4)
VERTEX ( 26%, < 135000, 132500>, 5)
IS_A (gdsII.boundary, 27%)
HAS ( 19%, 27%)
LAYER ( 27%, 4)
ATTR ( 27%, datatype, 0)
VERTEX ( 27%, < 170000, 132500>, 1)
VERTEX ( 27%, < 185000, 132500>, 2)
VERTEX ( 27%, < 185000, 187500>, 3)
VERTEX ( 27%, < 170000, 187500>, 4)
VERTEX ( 27%, < 170000, 132500>, 5)
IS_A (gdsII.boundary, 28%)
HAS ( 19%, 28%)
LAYER ( 28%, 8)
ATTR ( 28%, datatype, 0)
VERTEX ( 28%, < 140000, 137500>, 1)
VERTEX ( 28%, < 145000, 137500>, 2)
VERTEX ( 28%, < 145000, 182500>, 3)
VERTEX ( 28%, < 140000, 182500>, 4)
VERTEX ( 28%, < 140000, 137500>, 5)
IS_A (gdsII.boundary, 29%)
HAS ( 19%, 29%)
LAYER ( 29%, 8)
ATTR ( 29%, datatype, 0)

```

```

VERTEX ( 29%, < 175000, 137500>, 1)
VERTEX ( 29%, < 180000, 137500>, 2)
VERTEX ( 29%, < 180000, 182500>, 3)
VERTEX ( 29%, < 175000, 182500>, 4)
VERTEX ( 29%, < 175000, 137500>, 5)
IS_A (gdsII.boundary, 30%)
HAS ( 19%, 30%)
LAYER ( 30%, 25)
ATTR ( 30%, datatype, 0)
VERTEX ( 30%, < 117500, 117500>, 1)
VERTEX ( 30%, < 202500, 117500>, 2)
VERTEX ( 30%, < 202500, 202500>, 3)
VERTEX ( 30%, < 117500, 202500>, 4)
VERTEX ( 30%, < 117500, 117500>, 5)
IS_A (gdsII.boundary, 31%)
HAS ( 19%, 31%)
LAYER ( 31%, 8)
ATTR ( 31%, datatype, 0)
VERTEX ( 31%, < 205000, 202500>, 1)
VERTEX ( 31%, < 277500, 202500>, 2)
VERTEX ( 31%, < 277500, 277500>, 3)
VERTEX ( 31%, < 205000, 277500>, 4)
VERTEX ( 31%, < 205000, 202500>, 5)
IS_A (gdsII.text, 32%)
HAS ( 19%, 32%)
LAYER ( 32%, 16)
ATTR ( 32%, vertical presentation, MIDDLE)
ATTR ( 32%, horizontal presentation, CENTER)
ATTR ( 32%, mag, 0.00000)
VERTEX ( 32%, < 150000, 247500>, 1)
ATTR ( 32%, string, T18 )
ORIENT ( 32%, 0.00, 0.00, 0.00)
IS_A (gdsII.boundary, 33%)
HAS ( 19%, 33%)
LAYER ( 33%, 4)
ATTR ( 33%, datatype, 0)
VERTEX ( 33%, < 207500, 205000>, 1)
VERTEX ( 33%, < 275000, 205000>, 2)
VERTEX ( 33%, < 275000, 275000>, 3)
VERTEX ( 33%, < 207500, 275000>, 4)
VERTEX ( 33%, < 207500, 205000>, 5)

```

## APPENDIX E

### TDL DBIF

```
dbid('jkff',tdl,'1','2/21/84:10:05').
content(|descript__class(gate),
dir__name('rph'), pin__dir(in), has('clock',in),
has('j',in), has('k',in), has('ps',in),
has('pc',in), pin__dir(out), has('oq',out),
has('oqb',out),
desc('THE MODULE IS A MASTER / SLAVE JK FLIP-FLOP \
WITH PRESET AND PRECLEAR LINES. ; '),
delay('nandel',3,2,4,'/'), delay('not',3,2,4,'/'),
use('dig__3__nand','='nand',3,1,none,'nandel'),
use('dig__2__nand','='nand',2,1,none,'nandel'),
occ__name('dev1'), connect('dev1','nand__a',out,'1'),
has('dev1','dig__3__nand'),
connect('dev1','j',in,'1'),
connect('dev1','qb',in,'2'),
connect('dev1','clock',in,'3'),
occ__name('dev2'), connect('dev2','nand__b',out,'1'),
has('dev2','dig__3__nand'),
connect('dev2','k',in,'1'),
connect('dev2','q',in,'2'),
connect('dev2','clock',in,'3'),
occ__name('dev3'), connect('dev3','nand__c',out,'1'),
has('dev3','nand'), connect('dev3','ps',in,'1'),
connect('dev3','nand__a',in,'2'),
connect('dev3','nand__d',in,'3'),
occ__name('dev4'), connect('dev4','nand__d',out,'1'),
has('dev4','nand'), connect('dev4','pc',in,'1'),
connect('dev4','nand__b',in,'2'),
connect('dev4','nand__c',in,'3'),
occ__name('dev5'), connect('dev5','i',out,'1'),
has('dev5','not'), connect('dev5','clock',in,'1'),
occ__name('dev6'), connect('dev6','nand__e',out,'1'),
has('dev6','dig__2__nand'),
connect('dev6','nand__c',in,'1'),
connect('dev6','i',in,'2'),
occ__name('dev7'), connect('dev7','nand__f',out,'1'),
has('dev7','dig__2__nand'),
connect('dev7','nand__d',in,'1'),
connect('dev7','i',in,'2'),
occ__name('g__nand'),
```

```

connect('g__nand','q',out,'1'),
has('g__nand','nand'),
connect('g__nand','nand__e',in,'1'),
connect('g__nand','qb',in,'2'),
occ_name('h__nand'),
connect('h__nand','qb',out,'1'),
has('h__nand','nand'),
connect('h__nand','nand__f',in,'1'),
connect('h__nand','q',in,'2'),
occ_name('dev8'), connect('dev8','oq',out,'1'),
has('dev8','not'), connect('dev8','q',in,'1'),
occ_name('dev9'), delay('dev9','1'1,'/'),
connect('dev9','oqb',out,'1'),
has('dev9','not'), connect('dev9','qb',in,'1'),
pin('oqb'), pin('oq'), pin('pc'),
pin('ps'), pin('k'), pin('j'), pin('clock'),
signal('qb'), signal('q'), signal('nand__f'),
signal('nand__e'), signal('i'), signal('nand__d'),
signal('nand__c'), signal('nand__b'),
signal('nand__a'), device('not'),
device('nand'), device('dig__2__nand'),
device('dig__3__nand'), ext_out_pin('oqb'),
ext_out_pin('oq'), ext_in_pin('pc'),
ext_in_pin('ps'), ext_in_pin('k'),
ext_in_pin('j'), ext_in_pin('clock'), dummy}).

```

## APPENDIX F

### DBIF REPRESENTING CIF FILE

```

IS_A (cif.symbol, 2%)
ATTR (2%, symbol_number, 1)
IS_A (cif.polygon, 3%)
  LAYER ( 3%, XA)
  VERTEX ( 3%, < 5317800, 296000>, 1)
  VERTEX ( 3%, < 5317800, -117800>, 2)
  VERTEX ( 3%, < 4676000, -117800>, 3)
  VERTEX ( 3%, < 4676000, 296000>, 4)
  VERTEX ( 3%, < 5317800, 296000>, 5)
IS_A (cif.polygon, 4%)
  LAYER ( 4%, XB)
  VERTEX ( 4%, < 5200000, 296000>, 1)
  VERTEX ( 4%, < 5200000, 0>, 2)
  VERTEX ( 4%, < 4676000, 0>, 3)
  VERTEX ( 4%, < 4676000, 296000>, 4)
  VERTEX ( 4%, < 5200000, 296000>, 5)
IS_A (cif.polygon, 5%)
  LAYER ( 5%, XB)
  VERTEX ( 5%, < 5317800, 296000>, 1)
  VERTEX ( 5%, < 5317800, -117800>, 2)
  VERTEX ( 5%, < 4676000, -117800>, 3)
  VERTEX ( 5%, < 4676000, 296000>, 4)
  VERTEX ( 5%, < 5317800, 296000>, 5)
IS_A (cif.polygon, 6%)
  LAYER ( 6%, XC)
  VERTEX ( 6%, < 5040000, 296000>, 1)
  VERTEX ( 6%, < 5160000, 296000>, 2)
  VERTEX ( 6%, < 5160000, 280000>, 3)
  VERTEX ( 6%, < 5040000, 280000>, 4)
  VERTEX ( 6%, < 5040000, 296000>, 5)
IS_A (cif.polygon, 7%)
  LAYER ( 7%, XC)
  VERTEX ( 7%, < 5040000, 40000>, 1)
  VERTEX ( 7%, < 5160000, 40000>, 2)
  VERTEX ( 7%, < 5160000, 160000>, 3)
  VERTEX ( 7%, < 5040000, 160000>, 4)
  VERTEX ( 7%, < 5040000, 40000>, 5)
IS_A (cif.symbol_call, 8%)

```

```

ATTR (8%, symbol_number, 2)
VERTEX ( 8%, < 4680000, 0>, 1)
ORIENT ( 8%, 0.00, 0.00, 0.00)
IS_A (cif.wire, 9%)
LAYER ( 9%, XC)
WIDTH ( 9%, 15000)
VERTEX ( 9%, < 4760000, 296000>, 1)
VERTEX ( 9%, < 4760000, 280000>, 2)
IS_A (cif.wire, 10%)
LAYER ( 10%, XC)
WIDTH ( 10%, 15000)
VERTEX ( 10%, < 5100000, 280000>, 1)
VERTEX ( 10%, < 5100000, 240000>, 2)
VERTEX ( 10%, < 5000000, 240000>, 3)
VERTEX ( 10%, < 5000000, 80000>, 4)
VERTEX ( 10%, < 4960000, 80000>, 5)
IS_A (cif.wire, 11%)
LAYER ( 11%, XC)
WIDTH ( 11%, 15000)
VERTEX ( 11%, < 5100000, 40000>, 1)
VERTEX ( 11%, < 5100000, 0>, 2)
VERTEX ( 11%, < 4760000, 0>, 3)
VERTEX ( 11%, < 4760000, 40000>, 4)
IS_A (cif.polygon, 12%)
LAYER ( 12%, XD)
VERTEX ( 12%, < 5317800, 296000>, 1)
VERTEX ( 12%, < 5317800, 202500>, 2)
VERTEX ( 12%, < 4797500, 202500>, 3)
VERTEX ( 12%, < 4797500, 117500>, 4)
VERTEX ( 12%, < 4882500, 117500>, 5)
VERTEX ( 12%, < 4882500, 202500>, 6)
VERTEX ( 12%, < 5317800, 202500>, 7)
VERTEX ( 12%, < 5317800, -117800>, 8)
VERTEX ( 12%, < 4676000, -117800>, 9)
VERTEX ( 12%, < 4676000, 296000>, 10)
VERTEX ( 12%, < 5317800, 296000>, 11)

```



```

IS_A (cif.polygon, 13%)
LAYER ( 13%, XE)
VERTEX ( 13%, < 4797500, 117500>, 1)
VERTEX ( 13%, < 4722500, 117500>, 2)
VERTEX ( 13%, < 4722500, 42500>, 3)
VERTEX ( 13%, < 4797500, 42500>, 4)
VERTEX ( 13%, < 4797500, 117500>, 5)
IS_A (cif.polygon, 14%)
LAYER ( 14%, XE)
VERTEX ( 14%, < 4957500, 117500>, 1)
VERTEX ( 14%, < 4882500, 117500>, 2)
VERTEX ( 14%, < 4882500, 42500>, 3)
VERTEX ( 14%, < 4957500, 42500>, 4)
VERTEX ( 14%, < 4957500, 117500>, 5)
IS_A (cif.polygon, 15%)
LAYER ( 15%, XE)
VERTEX ( 15%, < 5157500, 157500>, 1)
VERTEX ( 15%, < 5042500, 157500>, 2)
VERTEX ( 15%, < 5042500, 42500>, 3)
VERTEX ( 15%, < 5157500, 42500>, 4)
VERTEX ( 15%, < 5157500, 157500>, 5)
IS_A (cif.polygon, 16%)
LAYER ( 16%, XE)
VERTEX ( 16%, < 5157500, 296000>, 1)
VERTEX ( 16%, < 5157500, 282500>, 2)
VERTEX ( 16%, < 5042500, 282500>, 3)
VERTEX ( 16%, < 5042500, 296000>, 4)
VERTEX ( 16%, < 5157500, 296000>, 5)
IS_A (cif.polygon, 17%)
LAYER ( 17%, XE)
VERTEX ( 17%, < 4797500, 277500>, 1)
VERTEX ( 17%, < 4722500, 277500>, 2)
VERTEX ( 17%, < 4722500, 202500>, 3)
VERTEX ( 17%, < 4797500, 202500>, 4)
VERTEX ( 17%, < 4797500, 277500>, 5)
IS_A (cif.polygon, 18%)
LAYER ( 18%, XE)
VERTEX ( 18%, < 4957500, 277500>, 1)
VERTEX ( 18%, < 4882500, 277500>, 2)
VERTEX ( 18%, < 4882500, 202500>, 3)
VERTEX ( 18%, < 4957500, 202500>, 4)
VERTEX ( 18%, < 4957500, 277500>, 5)

```

```

IS_A (cif.symbol, 19%)
ATTR (19%, symbol_number, 2)
IS_A (cif.polygon, 20%)
LAYER ( 20%, XC)
VERTEX ( 20%, < 200000, 280000>, 1)
VERTEX ( 20%, < 280000, 280000>, 2)
VERTEX ( 20%, < 280000, 200000>, 3)
VERTEX ( 20%, < 200000, 200000>, 4)
VERTEX ( 20%, < 200000, 280000>, 5)
IS_A (cif.polygon, 21%)
LAYER ( 21%, XC)
VERTEX ( 21%, < 40000, 40000>, 1)
VERTEX ( 21%, < 120000, 40000>, 2)
VERTEX ( 21%, < 120000, 120000>, 3)
VERTEX ( 21%, < 87500, 120000>, 4)
VERTEX ( 21%, < 87500, 152500>, 5)
VERTEX ( 21%, < 137500, 152500>, 6)
VERTEX ( 21%, < 137500, 135000>, 7)
VERTEX ( 21%, < 147500, 135000>, 8)
VERTEX ( 21%, < 147500, 185000>, 9)
VERTEX ( 21%, < 137500, 185000>, 10)
VERTEX ( 21%, < 137500, 167500>, 11)
VERTEX ( 21%, < 72500, 167500>, 12)
VERTEX ( 21%, < 72500, 120000>, 13)
VERTEX ( 21%, < 40000, 120000>, 14)
VERTEX ( 21%, < 40000, 40000>, 15)
IS_A (cif.polygon, 22%)
LAYER ( 22%, XC)
VERTEX ( 22%, < 200000, 40000>, 1)
VERTEX ( 22%, < 280000, 40000>, 2)
VERTEX ( 22%, < 280000, 120000>, 3)
VERTEX ( 22%, < 247500, 120000>, 4)
VERTEX ( 22%, < 247500, 167500>, 5)
VERTEX ( 22%, < 182500, 167500>, 6)
VERTEX ( 22%, < 182500, 185000>, 7)
VERTEX ( 22%, < 172500, 185000>, 8)
VERTEX ( 22%, < 172500, 135000>, 9)
VERTEX ( 22%, < 182500, 135000>, 10)
VERTEX ( 22%, < 182500, 152500>, 11)
VERTEX ( 22%, < 232500, 152500>, 12)
VERTEX ( 22%, < 232500, 120000>, 13)
VERTEX ( 22%, < 200000, 120000>, 14)
VERTEX ( 22%, < 200000, 40000>, 15)

```

```

IS_A (cif.polygon, 23%)
LAYER ( 23%, XF)
VERTEX ( 23%, < 132500, 130000>, 1)
VERTEX ( 23%, < 155000, 130000>, 2)
VERTEX ( 23%, < 155000, 190000>, 3)
VERTEX ( 23%, < 132500, 190000>, 4)
VERTEX ( 23%, < 132500, 130000>, 5)
IS_A (cif.polygon, 24%)
LAYER ( 24%, XF)
VERTEX ( 24%, < 165000, 130000>, 1)
VERTEX ( 24%, < 187500, 130000>, 2)
VERTEX ( 24%, < 187500, 190000>, 3)
VERTEX ( 24%, < 165000, 190000>, 4)
VERTEX ( 24%, < 165000, 130000>, 5)
IS_A (cif.polygon, 25%)
LAYER ( 25%, XC)
VERTEX ( 25%, < 40000, 280000>, 1)
VERTEX ( 25%, < 120000, 280000>, 2)
VERTEX ( 25%, < 120000, 235000>, 3)
VERTEX ( 25%, < 167500, 235000>, 4)
VERTEX ( 25%, < 167500, 115000>, 5)
VERTEX ( 25%, < 152500, 115000>, 6)
VERTEX ( 25%, < 152500, 215000>, 7)
VERTEX ( 25%, < 120000, 215000>, 8)
VERTEX ( 25%, < 120000, 200000>, 9)
VERTEX ( 25%, < 40000, 200000>, 10)
VERTEX ( 25%, < 40000, 280000>, 11)
IS_A (cif.polygon, 26%)
LAYER ( 26%, XG)
VERTEX ( 26%, < 135000, 132500>, 1)
VERTEX ( 26%, < 150000, 132500>, 2)
VERTEX ( 26%, < 150000, 187500>, 3)
VERTEX ( 26%, < 135000, 187500>, 4)
VERTEX ( 26%, < 135000, 132500>, 5)
IS_A (cif.polygon, 27%)
LAYER ( 27%, XG)
VERTEX ( 27%, < 170000, 132500>, 1)
VERTEX ( 27%, < 185000, 132500>, 2)
VERTEX ( 27%, < 185000, 187500>, 3)
VERTEX ( 27%, < 170000, 187500>, 4)
VERTEX ( 27%, < 170000, 132500>, 5)

```

```

IS_A (cif.polygon, 28%)
LAYER ( 28%, XH)
VERTEX ( 28%, < 140000, 137500>, 1)
VERTEX ( 28%, < 145000, 137500>, 2)
VERTEX ( 28%, < 145000, 182500>, 3)
VERTEX ( 28%, < 140000, 182500>, 4)
VERTEX ( 28%, < 140000, 137500>, 5)
IS_A (cif.polygon, 29%)
LAYER ( 29%, XH)
VERTEX ( 29%, < 175000, 137500>, 1)
VERTEX ( 29%, < 180000, 137500>, 2)
VERTEX ( 29%, < 180000, 182500>, 3)
VERTEX ( 29%, < 175000, 182500>, 4)
VERTEX ( 29%, < 175000, 137500>, 5)
IS_A (cif.polygon, 30%)
LAYER ( 30%, XI)
VERTEX ( 30%, < 117500, 117500>, 1)
VERTEX ( 30%, < 202500, 117500>, 2)
VERTEX ( 30%, < 202500, 202500>, 3)
VERTEX ( 30%, < 117500, 202500>, 4)
VERTEX ( 30%, < 117500, 117500>, 5)
IS_A (cif.polygon, 31%)
LAYER ( 31%, XH)
VERTEX ( 31%, < 205000, 202500>, 1)
VERTEX ( 31%, < 277500, 202500>, 2)
VERTEX ( 31%, < 277500, 277500>, 3)
VERTEX ( 31%, < 205000, 277500>, 4)
VERTEX ( 31%, < 205000, 202500>, 5)
IS_A (cif.polygon, 33%)
LAYER ( 33%, XG)
VERTEX ( 33%, < 207500, 205000>, 1)
VERTEX ( 33%, < 275000, 205000>, 2)
VERTEX ( 33%, < 275000, 275000>, 3)
VERTEX ( 33%, < 207500, 275000>, 4)
VERTEX ( 33%, < 207500, 205000>, 5)

```

APPENDIX G  
SAMPLE CIF FILE

DS 1 ;  
L XA ;  
P 5317800 296000 5317800 -117800 4676000 296000  
5317800 296000 ;  
L XB ;  
P 5200000 296000 5200000 0 4676000 0 4676000 296000 5200000 296000 ;  
P 5317800 296000 5317800 -117800 4676000 -117800 1676000 296000  
5317800 296000 ;  
L XC ;  
P 5040000 296000 5160000 296000 5160000 280000 5040000 280000 5040000  
296000 ;  
P 5040000 40000 5160000 40000 5160000 160000 5040000 160000 5040000  
40000 ;  
C 2 T 4680000 0 ;  
W 15000 4760000 296000 4760000 280000 ;  
W 15000 5100000 240000 5000000 240000 5000000 80000  
4960000 80000 ;  
W 15000 5100000 40000 5100000 0 4760000 0 4760000 40000 ;  
L XD ;  
P 5317800 296000 5317800 202500 4797500 202500 4797500 117500 4882500  
117500 4882500 202500 5317800 202500 5317800 -117800 4676000 -117800  
4676000 296000 5317800 296000 ;  
L XE ;  
P 4797500 117500 4722500 117500 4722500 42500 4797500 42500 4797500  
117500 ;  
P 4957500 117500 488250 117500 45882500 42500 4957500 42500 4957500  
117500 ;  
P 5157500 157500 5042500 157500 5042500 42500 5157500 42500 5157500  
157500 ;  
P 5157500 296000 5157500 282500 5042500 282500 5042500 296000 5157500  
296000 ;  
P 4797500 277500 4722500 277500 4722500 202500 4797500 202500 4797500  
277500 ;  
P 4957500 277500 4882500 277500 4882500 202500 4957500 202500 4957500  
277500 ;  
DF ;  
DS 2 ;  
L XC ;  
P 200000 280000 280000 280000 280000 200000 200000 200000 200000  
280000 ;

L XF ;  
P 132500 130000 155000 130000 155000 190000 132500 190000 132500  
130000 ;  
P 165000 130000 187500 130000 187500 190000 165000 190000 165000  
130000 ;  
L XC ;  
P 40000 280000 120000 280000 120000 235000 167500 235000 167500 115000  
152500 115000 152500 215000 120000 215000 120000 200000 40000 200000  
40000 280000 ;  
L XG ;  
P 135000 132500 150000 132500 150000 187500 135000 187500 135000  
132500 ;  
P 170000 132500 185000 132500 185000 187500 170000 187500 170000  
132500 ;  
L XH ;  
P 140000 137500 145000 137500 145000 182500 140000 182500 140000  
137500 ;  
P 175000 137500 180000 137500 180000 182500 175000 182500 175000  
137500 ;  
L XI ;  
P 117500 117500 202500 117500 202500 202500 117500 202500 117500  
117500 ;  
L XH ;  
P 205000 202500 277500 202500 277500 277500 205000 277500 205000  
202500 ;  
L XG ;  
P 207500 205000 275000 205000 275000 275000 207500 275000 207500  
205000 ;  
DF ;  
E ;

APPENDIX H  
TDL PREPROCESSOR BNF EXCERPT

"Reprinted with permission by the Calma Company"

## DEFINE

### Purpose

Introduces and terminates the actual topological description of the module network.

### Syntax

**DEFINE** <descriptor>₁ [<descriptor>₂... <descriptor> ...]

<descriptor>:

<occurrence phrase> -or- <external output phrase> -or-  
<Boolean phrase> -or- <local no-connect phrase> -or-  
<wired connection>

<occurrence phrase>:

<occurrence name> [( <output object list> )] = <device type>  
( <input connections list> ) [ / <delay> / ] ;

<occurrence name>:

1 to 12 alphanumeric characters chosen by the user to represent  
the occurrence of the device or primitive, or a question mark (?)  
which causes the compiler to create an <occurrence name>.

<output object list>:

<explicit object list> -or- <implicit object list>

<explicit object list>:

<object-pin match>₁ [, <object-pin match>₂... ,  
<output occurrence>_a ] 1

<implicit object list>:

<output occurrence>₁ [, <output occurrence>₂... ,  
<output occurrence>_a ] 1

- 
1. The maximum number of primitives in an <output object list>, represented here by "a", cannot exceed the number of outputs of the <device type> given. When the <device type> is a TDL primitive, the number of outputs is either the default (as specified in the TEGAS-5 Simulation Reference manual) or the number of outputs attributed to that primitive in the USE statement (q.v.) of the module being defined. When the <device type> references another module, either by the <module name> or by a <type name>, the number of outputs equals the number of <pin names> given in the OUTPUTS statement of that module.



<object pin-match>:  
   <output object> = <occurrence output pin> [ /<delay>/ ] -or-  
   NC = <occurrence output pin>

<output occurrence>:  
   [ <output object> ] [ /<delay>/ ] -or- NC

<occurrence output pin>:  
   <pin name> of an output of the <device type> given2

<output object>:  
   <signal name> -or- <external output pin> -or-  
   <external input pin>

<signal name>:  
   1 to 12 alphanumeric characters chosen by the user

<external output pin>:  
   a <pin name> given in the OUTPUTS statement of the module being  
   defined

<device type>:  
   <primitive name> -or- <module name> -or- <type name> -or-  
   <module id>

<primitive name>:  
   the standard name of a TDL primitive

<module name>:  
   first segment of a <module id> listed in the USE  
   statement (q.v.)

<type name>:  
   name given in the USE statement (q.v.) to a primitive  
   or module

<input connections list>:  
   <explicit connections list> -or- <implicit connections list>

- 
2. When the <device type> references another module, the  
 <occurrence output pin> and <occurrence input pin> are  
 <pin names> that appeared in the OUTPUTS and INPUTS  
 statements (q.v.) of that module. For TDL primitives,  
 see Appendix B, TDL Primitive Pin Names.

```

<explicit connections list>:
  <source-pin match>1{, <source-pin match>2...
  <source-pin match>b}3
<implicit connections list>:
  <input source>1{, <input source>2... , <input source>b}1
<source-pin match>:
  <input source> = <occurrence input pin>
<occurrence input pin>:
  <pin name> of an input of the <device type> given2
<input source>:
  <external input> -or- <signal name> -or-
  <occurrence output pin reference> -or-
  <no-connect> -or- <external output pin>
<external input pin>:
  <pin name> given in the INPUTS statement of the module being
  defined
<occurrence output pin reference>:
  <occurrence name> [<occurrence output pin>]4
<no-connect>:
  NC [<no-connect value>/]
<no-connect value>:
  1 -or- 0 -or- X -or- Z
<external output phrase>:
  <external output pin> = <gate type>
  (<input connections list>){/<delay>};
<gate type>:
  either a <primitive name> or <type name> referring to any TDL
  primitive or user-defined module that always produces
  exactly one output.

```

---

3. The maximum number of elements in an <input connections list>, represented here by "b", cannot exceed the number of inputs of the <device type> given. The number of inputs is determined in the same way as the number of outputs.

4. When an <occurrence output pin> is not provided, the default is the first output pin of the occurrence.

```

<local no-connect default>:
  NC = <no-connect value>;

<Boolean phrase>:
  <Boolean signal phrase> -or- <Boolean names phrase>

<Boolean signal phrase>:
  <signal name> [<inverter>] == <sum of products> [/<delay>/];

<Boolean names phrase>:
  <occurrence name> (<output object>) [<inverter>] ==
  <sum of products> [/<delay>/]

<sum of products>:
  <product term>1 [+<product term>2... +<product term>y]5

<product term>:
  <binary term>1 [*<binary term>2... *<binary term>g]

<binary term>:
  <input source> [<inverter>]6

<inverter>:
  '      ----   a single quote mark

<delay>:
  <delay value list> -or- <delay name>7

<wired connection>:
  <signal name> * & (<implicit connections list>)8

```

---

5. The maximum number of <product terms> in a <Boolean phrase>, represented here by "y", is directly related to the number of bits per word on the host machine. For 16-bit machines, y=5. For machines with larger words, y=(number of bits per word -1) divided by 3 (truncated).

6. Note: A <no-connect> in a <Boolean phrase> is not allowed.

7. The <delay value list> is explained with the DELAYS statement (q.v.). The <delay name> must have previously been assigned a set of values in the DELAYS statement.

8. The ampersand (&) aborts a message warning that a <device type> was not given. A signal connection is made with the default wired type being inserted (if necessary).

### Comments

The optional <output object list> of an <occurrence phrase> serves four purposes. 1) The <external output pins> or <external input pins> of the module can be tied to the <occurrence output pins> (i.e., the pins of a device) that drive them. 2) The <output object list> can also be used to assign a <signal name> to the output from a particular pin of the device occurrence. This <signal name> can, in turn, be used as an <input source> in any other <descriptor>. A <signal name> or <external output pin> can alternatively be matched to an <occurrence output pin> by a <wired connection> where the <input source> is an <occurrence output pin reference>. 3) In addition, the <output object list> can be used to explicitly no-connect an <occurrence output pin>. 4) Finally, the user can assign specific pin delays to output pins with the <output object list>.

If delay values are not specified in an occurrence statement or for its type, but delay values were specified on one or more output pins in the occurrence, all signals with specified delays will have that exact delay value for their delay and all the other signals for that occurrence statement will have the default value of unit (one).

```
EXAMPLE:
DELAYS:DEL1/1 2/;
USE: ANDN=ANDNAND(2,2);
DEFINE:ANDN1(OUT2,OUT2/DEL1/)=AND(IN1,IN2);
```

In the previous example, signals OUT1 and OUT2 have the following delays:

```
OUT1 = /1 1 1 1 1 1/
OUT2 = /1 1 1 2 2 2/
```

If a delay is placed on the type (with the USE statement) of occurrence statement and one or more output pins has delays, the pins with delays will have their resulting delay equal to the sum of the pin delay and the occurrence delay (the type delay if the occurrence delay was not specified). All output pins without specific assigned delays will take on the value of the occurrence statement delay (the type delay if the occurrence delay is not specified.)

```
EXAMPLE:
USE: ANDN=ANDNAND(2,2)/5 6 7/;
DEFINE: ANDN1(OUT1,OUT2/2 3 3/)=ANDN(IN1,IN2)/4 4/;
ANDN2(/3 4 5/,OUT3)=ANDN(IN1,IN2);
```

The signals, OUT1, OUT2, ANDN2(01), and OUT3 have the following delays:

```
OUT1=/4 4 4 4 4 4/  
OUT2=/6 7 7 6 7 7/  
ANDN2(01)=/8 10 12 8 10 12/  
OUT3=/5 6 7 5 6 7/
```

For more information, see the section on the DELAYS command later in this chapter.

The <input connections list> is not optional as is the <output object list>. Though their form is similar, the two lists serve different functions. Network interconnections (i.e., the matching of the output of one device to the input of another device or devices) are accomplished solely by means of the <input connection list>. The <output object list> is provided as a convenience, but the <input connections list> is essential for <device types> that have inputs.

Note that both lists have two forms; explicit and implicit. In the explicit form, each match is made by giving the <occurrence input (or output) pin> (a <pin name> valid for the <device type>) to which the assignment is to be made. The matches can be listed in any order. In implicit lists, the occurrence pins are not given. They are matched according to their relative order for that <device type> (first pin to first element in the list, second to second, etc.) until all elements in the list are matched to a pin. Additional commas between primitives in an implicit list cause pins to be skipped over during the matching; one pin is skipped for each comma after one obligatory comma. Leftover pins are not matched. A message is issued following module compilation warning when any pin has been unconnected. The pin will be identified by <occurrence name (pin name)>. An unconnected <occurrence input pin> will be tied to X (undetermined value). Only the <output object list> can contain delay assignments. An <input source> cannot be assigned a delay.

When an <occurrence output pin> (the output of a device) that is never matched to a <signal name> is to be given as an <input source>, it can be referenced only by an <occurrence output pin reference>. An <occurrence output pin> that is matched at some point to a <signal name> can be referenced by either the <signal name> or the <occurrence output pin reference>. Forward referencing is allowed; that is, a <signal name> can be used as an <input source> before it has appeared in the <descriptor> in which it will be matched to the <occurrence output pin> or other source that produces it. Likewise, an <occurrence output pin reference> can contain an <occurrence name> that has not appeared. If an <occurrence output pin> is never referenced, it is left unconnected and a warning message (q.v.) is issued.

The ampersand (&) allows any output to be wired to any source(s) of input. Using only one <input source> in the <input connections list> results in simple signal propagation. If two or more <input sources> are given, the default wired gate is inserted. The particular wired gate to be used can be specified in the WIRED statement (q.v.) Note that <no-connect> is not allowed as a <wired connection>.

The <no-connect> is used to tie an <input source> to one of four fixed values: one, zero, X (undetermined), or Z (high impedance). If only the letters 'NC' are used, the default value is assumed. If a <local no-connect default> is given, that value is used. Otherwise the global value, which can be set in the OPTIONS statement of the COMPILER command block (q.v.), is used.

The <Boolean phrase> allows the user to perform limited logical operations on signals, namely NOT(,), AND(*), and OR(+). The result can be given a <signal name> that can be used elsewhere as an <input source>. In a <named Boolean phrase>, the result can feed directly into an <external output pin> or <external input pin> given as the <output object>. In any case, the function must always be expressed as a <sum of products>, which is taken to be the inverse if an <inverter> appears before the '=='. Note that <no-connect> is not allowed in a <Boolean phrase>.

Each <occurrence phrase>, <external output phrase>, or <Boolean phrase> can optionally have a propagation <delay> associated with it. The propagation <delay> determines how long it takes for changes on the inputs to propagate through to affect the outputs. There is also a separate DELAYS statement that allows the user to preset delay values and assign them names. A <delay> in the DEFINE section has precedence over one set in the DELAYS statement. When the <device type> refers to another module, a <delay> is superfluous; a <delay> is associated only with a TDL primitive or a <Boolean phrase>.

## APPENDIX I

### SOURCE DBIF FOR CALMA—CIF DATA TRANSPORT CASE

#### CALMA DBIF:

```
dbid('test.db',gds,'1.4','7/24/84:14:50').
content([fonts('gdsii:font.tx',1),fonts('gdsii:font.tx',2),
generations(3),
db_user unit(0.00001),db_unit__meter(1.0e8),
strct(dev),
has(dev,i_3),
bdry(i_3),
lyr(i_3,40),dtatyp(i_3,0),
xy(i_3,53250,3000,1),xy(i_3,53250,-1250,2),
xy(i_3,46750,-1250,3),xy(i_3,46750,3000,4),
xy(i_3,53250,3000,5),
bdry(i_4),
has(dev,i_4),
lyr(i_4,1),dtatyp(i_4,0),
xy(i_4,52000,3000,1),xy(i_4,52000,0,2),
xy(i_4,46750,0,3),xy(i_4,46750,3000,4),
xy(i_4,52000,3000,5),
bdry(i_6),
has(dev,i_6),
lyr(i_6,10),dtatyp(i_6,0),
xy(i_6,50500,3000,1),xy(i_6,51750,3000,2),
xy(i_6,51750,2750,3),xy(i_6,50500,2750,4),
xy(i_6,50500,3000,5),
aref(i_7,t18),
has(dev,i_7),
rows(i_7,2),
columns(i_7,3),
xy(i_7,35000,0,1),
xy(i_7,46000,0,2),
xy(i_7,35000,3000,3),
sref(i_8,t18),
has(dev,i_8),
xy(i_8,46750,0,1),
angle(i_8,0.0),
path(i_9),
has(dev,i_9),
lyr(i_9,10),dtatyp(i_9,0),
width(i_9,150),
xy(i_9,47500,3000,1),xy(i_9,47500,2750,2),
path(i_10),
```

```

has(dev,i_10),
lyr(i_10,T0),dtatyp(i_10,0),
width(i_10,150),
xy(i_10,51000,2750,1),xy(i_10,51000,2500,2),
xy(i_10,50000,2500,3),xy(i_10,50000,750,4),
xy(i_10,49500,750,5),
path(i_11),
has(dev,i_11),
lyr(i_11,T0),dtatyp(i_11,0),
width(i_11,150),
xy(i_11,51000,500,1),xy(i_11,51000,0,2),
xy(i_11,47500,0,3),xy(i_11,47500,500,4),
bdry(i_12),
has(dev,i_12),
lyr(i_12,3),dtatyp(i_12,0),
xy(i_12,53250,3000,1),xy(i_12,53250,2000,2),
xy(i_12,48000,2000,3),xy(i_12,48000,1250,4),
xy(i_12,48750,1250,5),xy(i_12,48750,2000,6),
xy(i_12,53250,2000,7),xy(i_12,53250,-1250,8),
xy(i_12,46750,-1250,9),xy(i_12,46750,3000,10),
xy(i_12,53250,3000,11),
bdry(i_13),
has(dev,i_13),
lyr(i_13,T1),dtatyp(i_13,0),
xy(i_13,48000,1250,1),xy(i_13,47250,1250,2),
xy(i_13,47250,500,3),xy(i_13,48000,500,4),
xy(i_13,48000,1250,5),
aref(T_14,t18),
has(dev,i_14),
rows(i_14,2),
columns(i_14,2),
xy(i_14,46000,3000,1),
xy(i_14,53250,3000,2),
xy(i_14,46000,8000,3),
angle(i_14,90.0),
mag(i_14,0.5),
bdry(i_15),
has(dev,i_15),
lyr(i_15,T1),dtatyp(i_15,0),
xy(i_15,51500,1500,1),xy(i_15,50500,1500,2),
xy(i_15,50500,500,3),xy(i_15,51500,500,4),
xy(i_15,51500,1500,5),
bdry(i_16),
has(dev,i_16),
lyr(i_16,T1),dtatyp(i_16,0),
xy(i_16,51500,3000,1),xy(i_16,51500,2750,2),
xy(i_16,50500,2750,3),xy(i_16,50500,3000,4),
xy(i_16,51500,3000,5),
bdry(i_17),
has(dev,i_17),
lyr(i_17,T1),dtatyp(i_17,0),
xy(i_17,48000,2750,1),xy(i_17,47250,2750,2),

```



```

xy(i_17,47250,2000,3),xy(i_17,48000,2000,4),
xy(i_17,48000,2750,5),
bdry(i_18),
has(dev,i_18),
lyr(i_18,I1),dtatyp(i_18,0),
xy(i_18,49500,2750,1),xy(i_18,48750,2750,2),
xy(i_18,48750,2000,3),xy(i_18,49500,2000,4),
xy(i_18,49500,2750,5),
strct(t18),
bdry(i_20),
has(t18,i_20),
lyr(i_20,I0),dtatyp(i_20,0),
xy(i_20,2000,2800,1),xy(i_20,2800,2800,2),
xy(i_20,2800,2000,3),xy(i_20,2000,2000,4),
xy(i_20,2000,2800,5),
bdry(i_23),
has(t18,i_23),
lyr(i_23,2),dtatyp(i_23,0),
xy(i_23,1300,1300,1),xy(i_23,1600,1300,2),
xy(i_23,1600,1900,3),xy(i_23,1300,1900,4),
xy(i_23,1300,1300,5),
bdry(i_24),
has(t18,i_24),
lyr(i_24,2),dtatyp(i_24,0),
xy(i_24,1700,1300,1),xy(i_24,1900,1300,2),
xy(i_24,1900,1900,3),xy(i_24,1700,1900,4),
xy(i_24,1700,1300,5),
bdry(i_25),
has(t18,i_25),
lyr(i_25,I0),dtatyp(i_25,0),
xy(i_25,400,2800,1),xy(i_25,1200,2800,2),
xy(i_25,1200,2500,3),xy(i_25,1700,2500,4),
xy(i_25,1700,1200,5),xy(i_25,1500,1200,6),
xy(i_25,1500,2200,7),xy(i_25,1200,2200,8),
xy(i_25,1200,2000,9),xy(i_25,400,2000,10),
xy(i_25,400,2800,11),
bdry(i_26),
has(t18,i_26),
lyr(i_26,4),dtatyp(i_26,0),
xy(i_26,1400,1300,1),xy(i_26,1500,1300,2),
xy(i_26,1500,1900,3),xy(i_26,1400,1900,4),
xy(i_26,1400,1300,5),
text(i_32),
has(t18,i_32),
lyr(i_32,I6),vert_pres(i_32,middle),
horz_pres(i_32,center),
mag(i_32,0.1),
xy(i_32,1400,2475,1),
string(i_32,"T18"),
angle(i_32,0.0),
dummy]).

```

## APPENDIX J

### RULES FOR THE CALMA—CIF FORWARD TRANSPORT TEST CASE

#### *Rules for Translating CALMA Data into Generic Data (CALMA Input Rules).*

```

/* 1 */ macro_def(M):-strct(M).
/* 2 */ scale_(S):-db_unit_meter(S).
/* 3 */ polygon_(P):-bdry(P).
/* 4 */ polygon_(P):-box(P).
/* 5 */ wire_(W):-path(W).
/* 6 */ text_(T):-text(T).
/* 7 */ layer_(S,L):-lyr(S,L).
/* 8 */ width_(S,L):-width(S,L).
/* 9 */ text_(S,L):-text(S,L).
/* 10 */ vertex_(S,X,Y,I):-xy(S,X,Y,I),\+aref(S,_).
/* 11 */ tfont_(T,roman):-font_no(T,1).
/* 12 */ tfont_(T,italic):-font_no(T,2).
/* 13 */ v_just(S,N):-vert_pres(S,N).
/* 14 */ h_just(S,N):-horz_pres(S,N).
/* 15 */ textval_(X,Y):-string(X,Y).
/* 16 */ magnif_(S,M):-mag(S,M).
/* 17 */ orient_(S,Y,0.0,Z):-reflection(S),Y is 180.0,angle(S,Z),\+aref(S,_).
/* 18 */ orient_(S,Y,0.0,Z):-\+reflection(S),Y is 0.0,angle(S,Z),\+aref(S,_).
/* 19 */ macro_call(M,Name):-sref(M,Name).
/* 20 */ macro_call(M,Name):-aref(N,Name),rows(N,RM),columns(N,CM),
    row_gen(N,RM,CM),fail.
/* 21 */ macro_call(M,Name):-retract(current_rc(N,R,C)),aref(N,Name),name(N,NL),
    name(C,CL),name(R,RL),
    append(NL,"_",N1),append(N1,RL,N2),
    append(N2,"_",N3),append(N3,CL,NM),name(M,NM),
    xy_asst(M,R,C,N),has_asst(M,N),mag_asst(M,N).
/* 22 */ row_gen(N,I,C):-C>0,rc_gen(N,I,C).
/* 23 */ row_gen(N,I,C):-I>1,C>0,rc_gen(N,I,C),J is I-1,row_gen(N,J,C).
/* 24 */ rc_gen(N,I,1):-I>0,asserta(current_rc(N,I,1)).
/* 25 */ rc_gen(N,I,C):-I>0,C>1,asserta(current_rc(N,I,C)),J is C-1,rc_gen(N,I,J).
/* 26 */ has_(X,Y):-has(X,Y),\+node(Y),\+aref(Y,_).

/* 27 */ keep(dtatyp(X,Y)):-dtatyp(X,Y).
/* 28 */ keep(boxtype(X,Y)):-boxtype(X,Y).
/* 29 */ keep(pathtype(X,Y)):-pathtype(X,Y).
/* 30 */ keep(nodetype(X,Y)):-nodetype(X,Y).
/* 31 */ keep(texttype(X,Y)):-texttype(X,Y).
/* 32 */ keep(fonts(F,N)):-fonts(F,N).
/* 33 */ keep(generations(G)):-generations(G).

```

```

/* 34 */ keep(db_user_unit(U):-db_user_unit(U).
/* 35 */ keep(aref(N,Name):-aref(N,Name).
/* 36 */ keep(node(N):-node(N).
/* 37 */ keep(propval(S,P,V):-propval(S,P,V).
/* 38 */ keep(rows(X,R):-rows(X,R).
/* 39 */ keep(columns(X,R):-columns(X,R).
/* 40 */ keep(has_(X,Y):-has(X,Y),node(Y).
/* 41 */ keep(xy(S,X,Y,I):-aref(S,_),xy(S,X,Y,I).
/* 42 */ keep(angle(M,A):-aref(M,_),angle(M,A).
/* 43 */ keep(reflection(M):-aref(M,_),reflection(M).

/* 44 */ xy_asst(M,R,C,N):-rows(N,RM),columns(N,CM),xy(N,X1,Y1,1),xy(N,X2,Y2,2),
xy(N,X3,Y3,3),MY is R*(Y3-Y1)/RM,MX is C*(X2-X1)/CM,
xy_asst2(M,X1,Y1,MX,MY,N).

/* 45 */ xy_asst2(M,X1,Y1,MX,MY,N):-angle(N,A),xy_asst3(M,X1,Y1,MX,MY,A,N).
/* 46 */ xy_asst2(M,X1,Y1,MX,MY,N):- \+angle(N,_),reflection(N),
TX is X1+MX, TY is -(Y1+MY), asserta(xy(M,TX,TY,1)).
/* 47 */ xy_asst2(M,X1,Y1,MX,MY,N):- \+angle(N,_),\+reflection(N),
TX is X1+MX, TY is Y1+MY, asserta(xy(M,TX,TY,1)).
/* 48 */ xy_asst3(M,X1,Y1,MX,MY,A,N):-reflection(N),Cnv is Pi/180,
TX is X1+MX*cos(A*Cnv)+MY*sin(A*Cnv),
TY is -Y1+MX*sin(A*Cnv)-MY*cos(A*Cnv), asserta(xy(M,TX,TY,1)).
/* 49 */ xy_asst3(M,X1,Y1,MX,MY,A,N):- \+reflection(N),Cnv is Pi/180,
TX is X1+MX*cos(A*Cnv)-MY*sin(A*Cnv),
TY is Y1+MX*sin(A*Cnv)+MY*cos(A*Cnv), asserta(xy(M,TX,TY,1)).

/* 50 */ has_asst(M,N):-has(X,N),asserta(has(X,M)).
/* 51 */ has_asst(M,N):- \+has(_,N).
/* 52 */ mag_asst(M,N):-mag(N,A),asserta(mag(M,A)).
/* 53 */ mag_asst(M,N):- \+mag(N,_).
/* 54 */ relative_orient(S):-st_ref(S),angle(S,_),\+abs_angl(S).
/* 55 */ relative_magnif(S):-st_ref(S),mag(S,_),\+abs_mag(S).
/* 56 */ st_ref(S):-sref(S,_).
/* 57 */ st_ref(S):-aref(S,_).

```

*Rules for Translating Generic Data into CIF Data (CIF Output Rules).*

```
/* <<cifout.rul @ 7/25/84:17:06>> */
```

```
asst_uniq(P):-retr(P),asserta(P).
```

```
/*=====
| box(B,L,W,CX,CY):-
|   returns all instances of incoming boxes, N, including newly
|   generated instances, B, resulting from scaling and asserts
|   their rotations.
|=====*/
```

```
/* 1 */ box(B,L,W,CX,CY):-box__in(N,L__p,W__p,CX__p,CY__p,DX,DY),mac__link(N,Ref),
    box__scale(Ref,B,L,W,CX,CY,N,L__p,W__p,CX__p,CY__p),
    asserta(rotate(B,DX,DY,1)).
```

```
/*=====
| box__chk(B,L,W,CX,CY,DX,DY):-
|   Checks to see if a polygon B is a box. Returns same
|   values as for "box__in" plus DX and DY which indicate
|   the orientation of the the box off the positive X axis.
|=====*/
```

```
/* 2 */ box__chk(B,L,W,CX,CY,DX,DY):-polygon__(B),vtx(B,X,Y,1),vtx(B,X,Y,5),
    \+vtx(B,__,__,6),vtx(B,X2,Y2,2),vtx(B,X4,Y4,4),
    V1 is X2-X,V2 is Y2-Y,
    DX is X4-X,DY is Y4-Y,L1 is DX*DX+DY*DY,
    L2 is V1*V1+V2*V2,(DX-V1)*(DX-V1)+(DY-V2)*(DY-V2)==
    L1+L2,L is sqrt(L1),W is sqrt(L2),CX is (X4+X2)/2.0,
    CY is (Y4+Y2)/2.0.
```

```
/*=====
| box__in(B,L,W,CX,CY):-
|   1) returns all instances of incoming boxes with
|       name B   width W   length L
|       centerX CX   centerY CY
|       deltaX  DX   deltaY  DY
|   2) also removes all vertices which make up B.
|=====*/
```

```
/* 3 */ box__in(B,L,W,CX,CY,DX,DY):-box__chk(B,L,W,CX,CY,DX,DY),
    retr(vertex__(B,__,__,_)).
```

```
/*=====
| box__scale(Ref,B,L,W,CX,CY,N,L__p,W__p,CX__p,CY__p):-
|   Returns all scaled instances B for an incoming box
|   instance N which is defined in Ref.
|=====*/
```

```
/* 4 */ box__scale(Ref,B,L,W,CX,CY,N,L__p,W__p,CX__p,CY__p):-get__mag(S,Ref,M,Ref2),fail.
/* 5 */ box__scale(Ref,B,L,W,CX,CY,N,L__p,W__p,CX__p,CY__p):-macro__inst(Ref,M,Ref2),
    M\==1.0,L is M*L__p,
```

```

W is M*W_p, CX is M*CX_p, CY is M*CY_p,
gensym(box_,B),
asserta(item_inst(Ref2,B,N)), asserta(has_(Ref2,B)),
layer_chk(N,B).
/* 6 */ box_scale(Ref,B,L_p,W_p,CX_p,CY_p,B,L_p,W_p,CX_p,CY_p).

/*=====
+=====*/
/* 7 */ call_sym(P,N):-macro_call(P,X).map_(X,N),asserta(cur_tr_ord(1)),
chk_mirry(P),chk_mirrx(P),chk_rot(P),retract(cur_tr_ord(J)),
vtx(P,A,B,1),asserta(transl(P,A,B,J)),
retract(vertex_(P,_,_,_)).

/*=====
+=====*/
/* 8 */ chk_mirry(P):-orient_(P,A,_,_),A\==0.0,retract(cur_tr_ord(J)),I is J+1,
asserta(cur_tr_ord(I)),asserta(mirry(P,J)).
/* 9 */ chk_mirry(P):-\+orient_(P,A,_,_).
/* 10 */ chk_mirry(P):-orient_(P,0,_,_).

/* 11 */ chk_mirrx(P):-orient_(P,_,A,_),A\==0.0,retract(cur_tr_ord(J)),I is J+1,/* 1 */
asserta(cur_tr_ord(I)),asserta(mirrx(P,J)).
/* 12 */ chk_mirrx(P):-\+orient_(P,_,A,_.
/* 13 */ chk_mirrx(P):-orient_(P,_,0,_.

/* 14 */ chk_rot(P):-orient_(P,_,_,A),A\==0.0,retract(cur_tr_ord(J)),I is J+1,/* 1 */
asserta(cur_tr_ord(I)),CS is cos(A),SN is sin(A),
asserta(rotate(P,CS,SN,J)).
/* 15 */ chk_rot(P):-\+orient_(P,_,_,A).
/* 16 */ chk_rot(P):-orient_(P,_,_,0).

/*=====
+=====*/
/* 17 */ def_sym(N):-macro_def(S),gensym('#',N),asserta(map_(S,N)),
asserta(scale(N,1,1)).

/*=====
+=====*/
/* 18 */ flash(F,D,CX,CY):-flash_in(N,D_p,CX_p,CY_p), mac_link(N,Ref),
fl_scale(Ref,F,D,CX,CY,N,D_p,CX_p,CY_p).

/*=====
+=====*/
/* 19 */ flash_in(F,D,CX,CY):-flash_chk(F,D,CX,CY),retr(vertex_(F,_,_,_)).

/*=====
+=====*/
/* 20 */ fl_scale(Ref,F,D,CX,CY,N,D_p,CX_p,CY_p):-get_mag(S,Ref,M,Ref2),M\==1.0,
D is M*D_p,
CX is M*CX_p, CY is M*CY_p, gensym(flash_,F),
asserta(item_inst(Ref2,F,N)), asserta(has_(Ref2,F)),
layer_chk(N,F).

```

```

/* 21 */ fl_scale(Ref,F,D_p,CX_p,CY_p,F,D_p,CX_p,CY_p).

/*=====
+=====*/
/* 22 */ flsh_chk(F,D,CX,CY):-polygon_(F),vtx(F,X,Y,1),vtx(F,X,Y,9),\+vtx(F,_,_,10),
    vtx(F,X2,Y2,2),vtx(F,X3,Y3,3),vtx(F,X4,Y4,4),vtx(F,X5,Y5,5),
    vtx(F,X6,Y6,6),vtx(F,X7,Y7,7),vtx(F,X8,Y8,8),
    CX is (X+X5)/2.0,CY is (Y+Y5)/2.0,
    CX:=(X2+X6)/2.0,CY:=(Y2+Y6)/2.0,
    CX:=(X3+X7)/3.0,CY:=(Y3+Y7)/3.0,
    CX:=(X4+X8)/4.0,CY:=(Y4+Y8)/4.0,
    L1 is (X8-X)*(X8-X)+(Y8-Y)*(Y8-Y),
    L2 is (X2-X)*(X2-X)+(Y2-Y)*(Y2-Y),L1==L2,
    L3 is (X3-X2)*(X3-X2)+(Y3-Y2)*(Y3-Y2),L2==L3,
    L4 is (X4-X3)*(X4-X3)+(Y4-Y3)*(Y4-Y3),L3==L4,
    D is sqrt((X8-X4)*(X8-X4)+(Y8-Y4)*(Y8-Y4)).

/*=====
| g_m(S,Prev,Curr,New):-
|   Determines New magnitude based upon whether current
|   magnitude is relative or absolute and upon the
|   previous magnitude in the hierarchy.
+=====*/

/* 23 */ g_m(S,Prev,Curr,New):-relative_magnif(S), magnif_(S,Curr), New is Curr*Prev.
/* 24 */ g_m(S,Prev,Curr,Curr):- \+relative_magnif(S), magnif_(S,Curr).
/* 25 */ g_m(S,Prev,1.0,Prev):- \+magnif_(S,_).

/*=====
| get_mag(Call,Defn,M,Inst):-
|   For a given structure definition, Defn, returns all calls,
|   Call, to Defn, their magnitudes, M, and new instance
|   names, I.
+=====*/

/* 26 */ get_mag(Call,Defn,M,Inst):-macro_call(Call,Defn),mac_link(Call,Up_defn),
    get_mag(Up_call,Up_defn,Mp,Up_inst),
    g_m(Call,Mp,Mc,M),new_inst(Defn,M,Inst),
    new_call(Call,New_call,M,Inst),
    mhas(Call, New_call, Defn, Up_defn).
/* 27 */ get_mag(top,Defn,1.0,Defn):- \+macro_call(_,Defn).

/*=====
+=====*/
/* 28 */ has(X,Y):-has_(A,Y),map_(A,X),\+text_(Y).

/*=====
+=====*/
/* 29 */ keep(scale_(S)):-scale_(S).
/* 30 */ keep(macro_inst(X,Y,Z)):-macro_inst(X,Y,Z).
/* 31 */ keep(item_inst(X,Y,Z)):-item_inst(X,Y,Z).
/* 32 */ keep(ncall(X,Y)):-ncall(X,Y).

```

```

/* 33 */ keep(text_(T)):-text_(T).
/* 34 */ keep(textval_(T,S)):-textval_(T,S).
/* 35 */ keep(h_just(T,S)):-h_just(T,S).
/* 36 */ keep(v_just(T,S)):-v_just(T,S).
/* 37 */ keep(tfont_(T,S)):-tfont_(T,S).
/* 38 */ keep(magnif_(S,M)):-magnif_(S,M).
/* 39 */ keep(orient_(T,A,B,C)):-orient_(T,A,B,C),text_(T).
/* 40 */ keep(relative_magnif(S)):-relative_magnif(S).
/* 41 */ keep(relative_orient(S)):-relative_orient(S).
/* 42 */ keep(has_(X,Y)):-has_(X,Y),text_(Y).
/* 43 */ keep(layer_(T,X)):-layer_(T,X),text_(T).
/* 44 */ keep(vertex_(S,A,B,I)):-vertex_(S,A,B,I),text_(S).
/* 45 */ keep(map(X,Y)):-map_(X,Y).

/*=====
+=====*/
/* 46 */ layer(S,X):-layer_(S,1),\+text_(S),X=nd.
/* 47 */ layer(S,X):-layer_(S,2),\+text_(S),X=np.
/* 48 */ layer(S,X):-layer_(S,3),\+text_(S),X=nm.
/* 49 */ layer(S,X):-layer_(S,N),\+text_(S),N>3,concat(1,N,X).

/*=====
+=====*/
/* 50 */ layer_chk(Old,New):-layer_(Old,L),asserta(layer_(New,L)).
/* 51 */ layer_chk(Old,New):-\+layer_(Old,_).

/*=====
+=====*/
/* 52 */ mac_link(N,Ref):-has_(Ref,N),macro_def(Ref).

/*=====
|
| mhas(Call,Call,Defn,Defn):-
|   Determines whether the new calling macro is the same as
|   the previous calling macro. If so, then a new "has_" is
|   asserted between the next upper macro definition and this
|   new call.
|
+=====*/

/* 53 */ mhas(Call,Call,Defn,Defn).
/* 54 */ mhas(Call,New_call,Defn,Up_defn):-Call\==New_call,Defn=Up_defn,
      mhas2(Up_defn,Call,New_call).
/* 55 */ mhas(Call,New_call,Defn,Up_defn):-Defn\==Up_defn,Call=New_call,
      mhas2(Up_defn,Call,New_call).
/* 56 */ mhas(Call,New_call,Defn,Up_defn):-Defn\==Up_defn,Call\==New_call,
      mhas2(Up_defn,Call,New_call).

/* 57 */ mhas2(Up_defn,Call,Call):-asst_uniq(has_(Up_defn,Call)).
/* 58 */ mhas2(Up_defn,Call,New_call):-Call\==New_call,
      asst_uniq(has_(Up_defn,New_call)),
      retr(has_(Up_defn,Call)),
      asst_uniq(keep(has_(Up_defn,Call))).

```

```

/*=====
new_call(Old_call,New_call,M,Inst):-
    Determines whether a new macro call needs to be asserted
    or whether one already exists for the Old_call with
    magnitude M, calling Inst, the newly scaled instance.
+=====*/

/* 59 */ new_call(Old_call,New_call,M,Inst):- ncall(Old_call, New_call),
    macro_call(New_call,Inst).
/* 60 */ new_call(Old_call,New_call,M,Inst):- \+ncall(Old_call,_), M\==1,
    gensym(mcall_,New_call),asserta(macro_call(New_call,Inst)),
    vertex_(Old_call,X,Y,1),asserta(vertex_(New_call,X,Y,1)),
    orient_chk(Old_call,New_call),
    asserta(ncall(Old_call,New_call)),
    retract(macro_call(Old_call,OI)),
    asserta(keep(macro_call(Old_call,OI))).
/* 61 */ new_call(Old_call,Old_call,1.0,_).

/*=====
new_inst(Defn, M, Inst):-
    If the magnitude, M, is not equal to 1, then new_inst
    determines if a macro instance for Defn of magnitude
    M already exists. If not then one is created.
+=====*/

/* 62 */ new_inst(Defn, M, Inst):-M\==1,macro_inst(Defn,M,Inst),
    macro_call(New_call,Inst).
/* 63 */ new_inst(Defn, M, Inst):-M\==1,\+macro_inst(Defn,M,_),
    gensym(minst_,Inst),asserta(macro_def(Inst)),
    asserta(macro_inst(Defn,M,Inst)).
/* 64 */ new_inst(Defn, 1.0, Defn).

/*=====
+=====*/
/* 65 */ orient_chk(Old_call,New_call):-orient_(Old_call,A,B,C),
    asserta(orient_(New_call,A,B,C)).
/* 66 */ orient_chk(Old_call,New_call):- \+orient_(Old_call,A,B,C).

/*=====
+=====*/
/* 67 */ polygon(P):-poly_map(P,R).
/* 68 */ poly_map(P,R):-polygon_(R),\+box_chk(R,_,_,_,_,_),\+flsh_chk(R,_,_,_),
    asserta(good_poly(R)),fail.
/* 69 */ poly_map(P,R):-good_poly(R),mac_link(R,Ref),get_mag(S,Ref,M,Ref2),fail.
/* 70 */ poly_map(P,R):-good_poly(R),mac_link(R,Ref),macro_inst(Ref,M,Ref2),
    gensym(poly_,P),asserta(item_inst(Ref2,P,R)),
    asserta(has_(Ref2,P)),layer_chk(R,P).
/* 71 */ poly_map(R,R):-good_poly(R).

/*=====
+=====*/
/* 72 */ vertex(P,X,Y,I):-vertex_in(N,X_p,Y_p,I), mac_link(N,Ref),

```



```

        vx__scale(Ref,P,X,Y,N,X_p,Y_p).
/* 73 */ vertex_in(P,X,Y,I):-vtx(P,X,Y,I).
/* 74 */ vx__scale(Ref,P,X,Y,N,X_p,Y_p):-macro_inst(Ref,M,Inst),item_inst(Inst,P,N),
        X is M*X_p,Y is M*Y_p.
/* 75 */ vx__scale(Ref,N,X,Y,N,X,Y).
/* 76 */ vtx(S,X,Y,I):-scale_(V),vertex_(S,A,B,I),\+text_(S),
        X is A*1.0e8/V,Y is B*1.0e8/V.
/* 77 */ vtx(S,X,Y,I):- \+scale_(V),vertex_(S,A,B,I),\+text_(S),
        X is A*1.0e8,Y is B*1.0e8.

/*=====
+=====*/
/* 78 */ wire(W,Wid):-wire_in(N,Wid_p), mac_link(N,Ref), wr_scale(Ref,W,Wid,N,Wid_p).

/*=====
+=====*/
/* 79 */ wire_in(W,Wid):-wire_(W),width_(W,Wid).

/*=====
+=====*/
/* 80 */ wr_scale(Ref,W,Wid,N,Wid_p):-get_mag(S,Ref,M,Ref2), M\==1.0,
        Wid is M*Wid_p,
        gensym(wire_,W), asserta(item_inst(Ref2,W,N)),
        asserta(has_(Ref2,W)),layer_chk(N,W).
/* 81 */ wr_scale(Ref,W,Wid_p,W,Wid_p).

```

## APPENDIX K

### RULES FOR THE CALMA—CIF REVERSE TRANSPORT TEST CASE

*Rules for Translating CIF Data into Generic Data (CIF Input Rules).*

```

/* <<cifn.rul @ 8/7/84:11:20>> */

/*=====
| HAS
+=====*/

has_(D,I):-has(N,I),\+macro_inst(_,_,N),\+item_inst(_,I,_),
           \+ncall(_,I),map(D,N).
has_(D,I):-keep(has_(D,I)).

/*=====
| LAYERS
+=====*/

layer_(S,X):-layer(S,N),\+item_inst(_,S,_),ly2(S,N,X).
layer_(S,X):-keep(layer_(S,X)).
ly2(S,nd,1).
ly2(S,np,2).
ly2(S,nm,3).
ly2(S,N,X):-name(N,NL),append("l",XL,NL),name(X,XL),X>3.

/*=====
| Macro Definitions
+=====*/

macro_def(S):-def_sym(N),map(S,N),\+macro_inst(_,_,S).
macro_def(N):-def_sym(N),\+map(_,N),\+macro_inst(_,_,N).

/*=====
| Magnification and Orientation
+=====*/

magnif_(S,M):-keep(magnif_(S,M)).
relative_magnif(S):-keep(relative_magnif(S)).
relative_orient(S):-keep(relative_orient(S)).
orient_(T,A,B,C):-keep(orient_(T,A,B,C)).

/*=====
| POLYGONS

```

```

+=====*/

polygon_(P):-polygon(P),\+item_inst(,_P,_).
polygon_(P):-box(P,L,W,CX,CY),\+item_inst(,_P,_),rotate(P,A,B,1), C is
sqrt(A*A+B*B), V1 is (A/C)*(W/2), V2 is (B/C)*(L/2),
V3 is (B/C)*(W/2), V4 is (A/C)*(L/2),
X1 is CX-(V4+V3), X2 is CX+V4-V3, X3 is CX+V3+V4, X4 is CX+V3-V4,
Y1 is CY+V1-V2, Y2 is CY+V1+V2, Y3 is CY+V2-V1, Y4 is CY-(V1+V2),
asserta(vertex(P,X1,Y1,1)), asserta(vertex(P,X2,Y2,2)),
asserta(vertex(P,X3,Y3,3)), asserta(vertex(P,X4,Y4,4)),
asserta(vertex(P,X1,Y1,5)).
polygon_(F):-flash(F,D,CX,CY),\+item_inst(,_F,_),SR2 is sqrt(2.0), SR2_3 is
SR2*SR2*SR2, A is D/(2+SR2_3), MA is -A, MD is -D,
asserta(vertex(F,MA,MD,1)), asserta(vertex(F,MD,MA,2)),
asserta(vertex(F,MD,A,3)), asserta(vertex(F,MA,D,4)),
asserta(vertex(F,A,D,5)), asserta(vertex(F,D,A,6)),
asserta(vertex(F,D,MA,7)), asserta(vertex(F,A,MD,8)),
asserta(vertex(F,MA,MD,9)).

/*=====
| SCALE
+=====*/

scale_(Z):-keep(scale_(Z)).

/*=====
| TEXT
+=====*/

text_(T):-keep(text_(T)).
textval_(T,V):-keep(textval_(T,V)).
v_just(T,J):-keep(v_just(T,J)).
h_just(T,J):-keep(h_just(T,J)).

/*=====
| Transformation Matrix
+=====*/

tm(P,I,T11,T12,T13,
T21,T22,T23,
T31,T32,T33):-transl(P,X,Y,I), J is I+1, tm(P,J,T11,T12,T13,T21,T22,
T23,S31,S32,S33), T31 is X*T11+Y*T21+S31,
T32 is X*T12+Y*T22+S32, T33 is X*T13+Y*T23+S33.
tm(P,I,T11,T12,T13,
T21,T22,T23,
T31,T32,T33):-mirrorx(P,I), J is I+1, tm(P,J,S11,S12,S13,T21,T22,
T23,T31,T32,T33), T11 is -S11, T12 is -S12, T13 is -S13.
tm(P,I,T11,T12,T13,
T21,T22,T23,
T11,T12,T13):-mirrory(P,I), J is I+1, tm(P,J,T11,T12,T13,S21,S22,
S23,T31,T32,T33), T21 is -S21, T22 is -S22, T23 is -S23.
tm(P,I,T11,T12,T13,

```

```

T21,T22,T23,
T11,T12,T13):-rotate(P,A,B,I), J is I+1, C is sqrt(A*A+B*B),
tm(P,J,S11,S12,S13,S21,S22,S23,T31,T32,T33),
T11 is (A*S11+B*S21)/C, T12 is (A*S12+B*S22)/C,
T13 is (A*S13+B*S23)/C, T21 is (A*S21-B*S11)/C,
T22 is (A*S22+B*S12)/C, T23 is (A*S23+B*S13)/C.
tm(P,I,1,0,0,
0,1,0,
0,0,1):- \+transl(P,_,_,I),\+mirrorx(P,I),\+mirrory(P,I),
\+rotate(P,_,_I).

/*=====
| VERTICES
+=====*/

vertex_(S,X,Y,I):-vertex(S,A,B,I),\+item_inst(_,S,_),vx2(S,F,G),
X is F*A/G, Y is F*B/G.
vx2(S,F,G):-comptd_scale(S,F,G).
vx2(S,F,G):- \+comptd_scale(S,F,G),has(M,S),scale(M,F1,G),vx3(F1,F),
asserta(comptd_scale(S,F,G)).
vx2(S,F,G):- \+comptd_scale(S,F,G),has(M,S),\+scale(M,_,_),vx3(1,Z),
asserta(comptd_scale(S,F,G)).
vx3(I,J):-keep(scale_(Z)),J is Z*I/1.0e8.
vx3(I,J):- \+keep(scale_(Z)),J is I/1.0e8.

vertex_(S2,X,Y,I):-call_sym(S,M),vtx2(S,S2),tm(S,1,R1,R2,0,R3,R4,0,X,Y,1),
vtxfn(S2,R1,R2,R4),I is 1.
vtx2(S,S):- \+ncall(_,S).
vtx2(S,S2):-ncall(S2,S).
vtxfn(S,0,R2,0):-R2>=0,asserta(orient_(S,0,0,90)).
vtxfn(S,0,R2,0):-R2<0,asserta(orient_(S,0,0,-90)).

/* V */

vtxfn(S,R1,R2,R1):-R1==0,A is atan(R2/R1),A==0.0,asserta(orient_(S,0,0,A)).
vtxfn(S,R1,R2,R1):-R1==0,A is atan(R2/R1),A==0.0.

/* V */

vtxfn(S,R1,R2,R4):-R1\==R4,R1\==0,A is atan(R2/R1),
asserta(orient_(S,180,0,A)).

vertex_(S,X,Y,I):-keep(vertex_(S,X,Y,I)).

/*=====
| WIRES
+=====*/

wire_(W):-wire(W,Wid),\+item_inst(_,W,_).
width_(W,Wid):-wire(W,Wid),\+item_inst(_,W,_).

/*=====

```

```

| Macro Calls
+=====*/

macro_call(S,N):-call_sym(S,X,\+ncall(_ ,S),mc2(N,X).
macro_call(S,N):-keep(macro_call(S,N)).
mc2(N,X):-map(N,X).
mc2(X,X):- \+map(_ ,X).

/*=====
| General
+=====*/

item_inst(X,Y,Z):-keep(item_inst(X,Y,Z)).
ncall(S,T):-keep(ncall(S,T)).
map(X,Y):-keep(map(X,Y)).
macro_inst(X,M,I):-keep(macro_inst(X,M,I)).

```

*Rules for Translating Generic Data into CALMA Data (CALMA Output Rules).*

```

/* <<gdsout.rul @ 8/8/84:12:47>> */

box(Bx):-polygon_(Bx),vertex_(Bx,X,Y,1),vertex_(Bx,X,Y,5),\+vertex_(Bx,_,_,6),
        vertex_(Bx,X2,Y2,2),vertex_(Bx,X4,Y4,4),A is X2-X, B is Y2-Y,
        C is X4-X, D is Y4-Y, L1 is C*C+D*D, L2 is A*A+B*B,
        ((C-A)*(C-A)+(D-B)*(D-B)) == L1+L2.
bdry(B):-polygon_(B),\+box(B).
path(P):-wire_(P).
strct(S):-macro_def(S).
sref(T,N):-macro_call(T,N),name(T,TL),\+contains(TL,"_ _").
aref(S,N):-macro_call(T,N),name(T,TLB),append(TL,"_ _1_ _1",TLB),name(T2,TL),
        keep(aref(T2,N)),asserta(ar(T2,T,N)),fail.
aref(S,N):-macro_call(T,N),name(T,TL),contains(TL,"_ _"),
        \+ar(_,T,_),retr(has_(_,T)),retr(magnif_(T,_)),
        retr(orient_(T,_,_,_)),fail.
aref(S,N):-retract(ar(S,T,N)),reth(S,T),retm(S,T),reto(S,T).
reth(S,T):-retract(has_(A,T)),asserta(has_(A,S)),fail.
reth(S,T).
retm(S,T):-retract(magnif_(T,A)),asserta(magnif_(S,A)),fail.
retm(S,T).
reto(S,T):-retract(orient_(T,A,B,C)),asserta(orient_(S,A,B,C)),fail.
reto(S,T).
text(T):-text_(T).
node(N):-keep(node(N)).
dtyp(S,X):-keep(dtyp(S,X)).
pathype(S,X):-keep(pathype(S,X)).
nodetype(S,X):-keep(nodetype(S,X)).
textype(S,X):-keep(textype(S,X)).
boxtype(S,X):-keep(boxtype(S,X)).
fonts(F,N):-keep(fonts(F,N)).
generations(G):-keep(generations(G)).
db_user_unit(U):-keep(db_user_unit(U)).
db_unit_meter(U):-scale_(U).
xy(S,X,Y,I):-vertex_(S,X,Y,I),\+macro_call(S,_).
xy(S,X,Y,I):-vertex_(S,X,Y,I),macro_call(S,_),name(S,SL),\+contains(SL,"_ _").
xy(S,X,Y,I):-vertex_(M,RX,RY,1),name(M,ML),indexL(ML,"_ _1_ _1",P),
        L is P-1,substrL(ML,SL,1,L),name(S,SL),keep(aref(S,_)),
        keep(rows(S,RM)),keep(columns(S,CM)),
        name(RM,RML),name(CM,CML),append(SL,"_ _",Nm1),
        append(Nm1,RML,Nm2),append(Nm2,"_ _",Nm3),append(Nm3,CML,NmL),
        name(Nm,NmL),
        vertex_(Nm,X2,Y2,1),X is RX-(X2-RX)/(CM-1), I is 1,
        Y is RY-(Y2-RY)/(RM-1),asserta(xy_(S,X2,Y,2)),
        asserta(xy_(S,X,Y2,3)).
xy(S,X,Y,I):-xy_(S,X,Y,I).
lyr(S,L):-layer_(S,L).
has(X,Y):-has_(X,Y).
width(S,W):-width_(S,W).
reflections(S):-orient_(S,180.0,_,_).
abs_mag(S):-macro_call(S),magnif_(S,_),\+relative_magnif(S).

```

```
abs_angl(S):-macro_call(S,orient__(S,_,_,_),\+relative_orient(S).
columns(S,C):-keep(columns(S,C)).
rows(S,R):-keep(rows(S,R)).
font_no(T,1):-tfont(T,roman).
font_no(T,2):-tfont(T,italic).
vert_pres(S,N):-v_just(S,N).
horz_pres(S,N):-h_just(S,N).
string(S,Str):-textval__(S,Str).
mag(S,M):-magnif__(S,M).
angle(S,A):-orient(S,_,_,A).
propval(S,P,V):-keep(propval(S,P,V)).
```

## APPENDIX L

### GENERIC FACTS CREATED FROM SOURCE TDL

```

node(node_1). node(node_2). node(node_3). node(node_4).
node(node_5). node(node_6). node(node_7). node(node_8).
node(node_9). node(node_10).node(node_11).node(node_12).
node(node_13).node(node_14).node(node_15).node(node_16).
node(node_17).node(node_18).node(node_19).node(node_20).
net(qb).net(q). net(nand__f). net(nand__e).
net(i). net(nand__d).net(nand__c).net(nand__b).
net(nand__a).net(net_1). net(net_2). net(net_3).
net(net_4). net(net_5). net(net_6). net(net_7).
net(net_8). net(net_9).
box(dev1). box(dev2). box(dev3). box(dev4).
box(dev5). box(dev6). box(dev7). box(g__nand).
box(h__nand). box(dev8). box(dev9). box(ext).
box__type(not). box__type(nand). box__type(dig__2__nand).
box__type(dig__3__nand). box__type(exterior).
net__type(nt_1). net__type(nt_2). net__type(nt_3).
net__type(nt_4). net__type(nt_5). net__type(nt_6).
net__type(nt_7). net__type(nt_8). net__type(nt_9).
net__type(nt_10). net__type(nt_11). net__type(nt_12).
net__type(nt_13). net__type(nt_14). net__type(nt_15).
net__type(nt_16). net__type(nt_17). net__type(nt_18).
node__type(in1). node__type(out1). node__type(in2).
node__type(in3). node__type(oqb). node__type(oq).
node__type(pc).node__type(ps).node__type(k).
node__type(j). node__type(clock). node__dir(out).
node__dir(in).
connected(dev5,net_9,node_19). connected(dev2,net_8,node_8).
connected(dev1,net_7,node_8). connected(dev1,net_6,node_10).
connected(dev2,net_5,node_10). connected(dev3,net_4,node_16).
connected(dev4,net_3,node_16). connected(dev8,net_2,node_20).
connected(dev9,net_1,node_20). connected(dev1,nand__a,node_11).
connected(dev1,qb,node_9). connected(dev2,nand__b,node_11).
connected(dev2,q,node_9). connected(dev3,nand__c,node_17).
connected(dev3,nand__a,node_15). connected(dev3,nand__d,node_18).
connected(dev4,nand__d,node_17). connected(dev4,nand__b,node_15).
connected(dev4,nand__c,node_18). connected(dev5,i,node_20).
connected(dev6,nand__e,node_14). connected(dev6,nand__c,node_13).
connected(dev6,i,node_12). connected(dev7,nand__f,node_14).
connected(dev7,nand__d,node_13). connected(dev7,i,node_12).
connected(g__nand,q,node_17). connected(g__nand,nand__e,node_16).

```



```

connected(g__nand,qb,node__15).    connected(h__nand,qb,node__17).
connected(h__nand,nand__f,node__16).    connected(h__nand,q,node__15).
connected(dev8,q,node__19).    connected(dev9,qb,node__19).
connected(ext,net__9,node__5).    connected(ext,net__8,node__5).
connected(ext,net__7,node__5).    connected(ext,net__6,node__4).
connected(ext,net__5,node__3).    connected(ext,net__4,node__2).
connected(ext,net__3,node__1).    connected(ext,net__2,node__7).
connected(ext,net__1,node__6).
has__(nand__a,nt__18).    has__(nand__b,nt__17).
has__(nand__c,nt__16).
has__(nand__d,nt__15).    has__(i,nt__14).    has__(nand__e,nt__13).
has__(nand__f,nt__12).    has__(q,nt__11).    has__(qb,nt__10).
has__(net__1,nt__9).    has__(net__2,nt__8).    has__(net__3,nt__7).
has__(net__4,nt__6).    has__(net__5,nt__5).    has__(net__6,nt__4).
has__(net__7,nt__3).    has__(net__8,nt__2).    has__(net__9,nt__1).
has__(not,node__20).    has__(node__20,out1).    has__(not,node__19).
has__(node__19,in1).    has__(nand,node__18).    has__(node__18,in3).
has__(nand,node__17).    has__(node__17,out1).    has__(nand,node__16).
has__(node__16,in1).    has__(nand,node__15).    has__(node__15,in2).
has__(dig__2__nand,node__14).    has__(node__14,out1).
has__(dig__2__nand,node__13).    has__(node__13,in1).
has__(dig__2__nand,node__12).    has__(node__12,in2).
has__(dig__3__nand,node__11).    has__(node__11,out1).
has__(dig__3__nand,node__10).    has__(node__10,in1).
has__(dig__3__nand,node__9).    has__(node__9,in2).
has__(dig__3__nand,node__8).    has__(node__8,in3).
has__(exterior,node__7).    has__(node__7,oq).    has__(exterior,node__6).
has__(node__6,oqb).    has__(exterior,node__5).    has__(node__5,clock).
has__(exterior,node__4).    has__(node__4,j).    has__(exterior,node__3).
has__(node__3,k).    has__(exterior,node__2).    has__(node__2,ps).
has__(exterior,node__1).    has__(node__1,pc).
has__(dev1,dig__3__nand).    has__(dev2,dig__3__nand).
has__(dev3,nand).    has__(dev4,nand).    has__(dev5,not).
has__(dev6,dig__2__nand).    has__(dev7,dig__2__nand).
has__(g__nand,nand).    has__(h__nand,nand).    has__(dev8,not).
has__(dev9,not).
has__(node__10,in).    has__(node__13,in).    has__(node__16,in).
has__(node__19,in).    has__(node__11,out).    has__(node__14,out).
has__(node__17,out).    has__(node__20,out).    has__(node__9,in).
has__(node__12,in).    has__(node__15,in).    has__(node__8,in).
has__(node__18,in).    has__(node__6,out).    has__(node__7,out).
has__(node__1,in).    has__(node__2,in).    has__(node__3,in).
has__(node__4,in).    has__(node__5,in).    has__(ext,exterior).

```

## APPENDIX M

### GENERIC FACTS CREATED FROM SOURCE CALMA DATA

```

polygon_(i_3).polygon_(i_4).polygon_(i_6).polygon_(i_12).      polygon_(i_13).
polygon_(i_15).      polygon_(i_16).      polygon_(i_17).      polygon_(i_18).
polygon_(i_20).      polygon_(i_23).      polygon_(i_24).      polygon_(i_25).
polygon_(i_26).
wire_(i_9).      wire_(i_10).      wire_(i_11).
macro_def(dev).      macro_def(t18).
macro_call(i_8,t18).      macro_call(i_14__1__1,t18).
macro_call(i_14__1__2,t18).      macro_call(i_14__2__1,t18).
macro_call(i_14__2__2,t18).      macro_call(i_7__1__1,t18).
macro_call(i_7__1__2,t18).      macro_call(i_7__1__3,t18).
macro_call(i_7__2__1,t18).      macro_call(i_7__2__2,t18).
macro_call(i_7__2__3,t18).
scale_(100000000).
layer_(i_3,40).layer_(i_4,1).layer_(i_6,10).layer_(i_9,10).
layer_(i_10,10).      layer_(i_11,10).      layer_(i_12,3).layer_(i_13,11).
layer_(i_15,11).      layer_(i_16,11).      layer_(i_17,11).      layer_(i_18,11).
layer_(i_20,10).      layer_(i_23,2).layer_(i_24,2).layer_(i_25,10).
layer_(i_26,4).layer_(i_32,16).
vertex_(i_7__2__3,46000,3000,1).      vertex_(i_7__2__2,42333,3000,1).
vertex_(i_7__2__1,38667,3000,1).      vertex_(i_7__1__3,46000,1500,1).
vertex_(i_7__1__2,42333,1500,1).      vertex_(i_7__1__1,38667,1500,1).
vertex_(i_14__2__3,41000,10250,1).      vertex_(i_14__2__2,41000,6625,1).
vertex_(i_14__2__1,43500,10250,1).      vertex_(i_14__1__3,43500,6625,1).
vertex_(i_3,53250,3000,1).      vertex_(i_3,53250,-1250,2).      vertex_(i_3,46750,-1250,3).
vertex_(i_3,46750,3000,4).      vertex_(i_3,53250,3000,5).      vertex_(i_4,52000,3000,1).
vertex_(i_4,52000,0,2).      vertex_(i_4,46750,0,3).      vertex_(i_4,46750,3000,4).
vertex_(i_4,52000,3000,5).      vertex_(i_6,50500,3000,1).      vertex_(i_6,51750,3000,2).
vertex_(i_6,51750,2750,3).      vertex_(i_6,50500,2750,4).      vertex_(i_6,50500,3000,5).
vertex_(i_8,46750,0,1).      vertex_(i_9,47500,3000,1).      vertex_(i_9,47500,2750,2).
vertex_(i_10,51000,2750,1).      vertex_(i_10,51000,2500,2).      vertex_(i_10,50000,2500,3).
vertex_(i_10,50000,750,4).      vertex_(i_10,49500,750,5).      vertex_(i_11,51000,500,1).
vertex_(i_11,51000,0,2).      vertex_(i_11,47500,0,3).      vertex_(i_11,47500,500,4).
vertex_(i_12,53250,3000,1).      vertex_(i_12,53250,2000,2).      vertex_(i_12,48000,2000,3).
vertex_(i_12,48000,1250,4).      vertex_(i_12,48750,1250,5).      vertex_(i_12,48750,2000,6).
vertex_(i_12,53250,2000,7).      vertex_(i_12,53250,-1250,8).      vertex_(i_12,46750,-1250,9).
vertex_(i_12,46750,3000,10).      vertex_(i_12,53250,3000,11).
vertex_(i_13,48000,1250,1).      vertex_(i_13,47250,1250,2).      vertex_(i_13,47250,500,3).
vertex_(i_13,48000,500,4).      vertex_(i_13,48000,1250,5).      vertex_(i_15,51500,1500,1).
vertex_(i_15,50500,1500,2).      vertex_(i_15,50500,500,3).      vertex_(i_15,51500,500,4).
vertex_(i_15,51500,1500,5).      vertex_(i_16,51500,3000,1).      vertex_(i_16,51500,2750,2).

```

```

vertex_(i_16,50500,2750,3).  vertex_(i_16,50500,3000,4).  vertex_(i_16,51500,3000,5).
vertex_(i_17,48000,2750,1).  vertex_(i_17,47250,2750,2).  vertex_(i_17,47250,2000,3).
vertex_(i_17,48000,2000,4).  vertex_(i_17,48000,2750,5).  vertex_(i_18,49500,2750,1).
vertex_(i_18,48750,2750,2).  vertex_(i_18,48750,2000,3).  vertex_(i_18,49500,2000,4).
vertex_(i_18,49500,2750,5).  vertex_(i_20,2000,2800,1).  vertex_(i_20,2800,2800,2).
vertex_(i_20,2800,2000,3).  vertex_(i_20,2000,2000,4).  vertex_(i_20,2000,2800,5).
vertex_(i_23,1300,1300,1).  vertex_(i_23,1600,1300,2).  vertex_(i_23,1600,1900,3).
vertex_(i_23,1300,1900,4).  vertex_(i_23,1300,1300,5).  vertex_(i_24,1700,1300,1).
vertex_(i_24,1900,1300,2).  vertex_(i_24,1900,1900,3).  vertex_(i_24,1700,1900,4).
vertex_(i_24,1700,1300,5).  vertex_(i_25,400,2800,1).  vertex_(i_25,1200,2800,2).
vertex_(i_25,1200,2500,3).  vertex_(i_25,1700,2500,4).  vertex_(i_25,1700,1200,5).
vertex_(i_25,1500,1200,6).  vertex_(i_25,1500,2200,7).  vertex_(i_25,1200,2200,8).
vertex_(i_25,1200,2000,9).  vertex_(i_25,400,2000,10).  vertex_(i_25,400,2800,11).
vertex_(i_26,1400,1300,1).  vertex_(i_26,1500,1300,2).  vertex_(i_26,1500,1900,3).
vertex_(i_26,1400,1900,4).  vertex_(i_26,1400,1300,5).  vertex_(i_32,1400,2475,1).
width_(i_9,150).  width_(i_10,150).  width_(i_11,150).
orient_(i_8,0,0,0).  orient_(i_32,0,0,0).
has_(dev,i_7__2__3).  has_(dev,i_7__2__2).  has_(dev,i_7__2__1).
has_(dev,i_7__1__3).  has_(dev,i_7__1__2).  has_(dev,i_7__1__1).
has_(dev,i_14__2__2).  has_(dev,i_14__2__1).  has_(dev,i_14__1__2).
has_(dev,i_14__1__1).  has_(dev,i_3).  has_(dev,i_4).  has_(dev,i_6).
has_(dev,i_8).  has_(dev,i_9).  has_(dev,i_10).  has_(dev,i_11).
has_(dev,i_12).  has_(dev,i_13).
has_(dev,i_15).  has_(dev,i_16).  has_(dev,i_17).  has_(dev,i_18).
has_(t18,i_20).  has_(t18,i_23).  has_(t18,i_24).  has_(t18,i_25).
has_(t18,i_26).  has_(t18,i_32).
magnif_(i_14__2__2,0.5).  magnif_(i_14__2__1,0.5).
magnif_(i_14__1__2,0.5).  magnif_(i_14__1__1,0.5).
magnif_(i_14,0.5).  magnif_(i_32,0.1).
relative_orient(i_8).  relative_orient(i_14).  relative_magnif(i_14).
text_(i_32).  textval_(i_32,{84,49,56}).
h_just(i_32,center).  v_just(i_32,middle).

```

## APPENDIX N

### CIF DATA OUTPUT FROM GENERIC FACTS

```

dbid('test.db',cif,'1.0','7/26/84:21:55').
content([polygon(poly_1),
polygon(i_12), polygon(i_25),
box(box_1,300,50,725,800),    box(i_26,600,100,1450,1600),
box(box_2,300,100,900,800),    box(i_24,600,200,1800,1600),
box(box_3,300,150,725,800),    box(i_23,600,300,1450,1600),
box(box_4,400,400,1200,1200),  box(i_20,800,800,2400,2400),
box(i_18,750,750,49125,2375),  box(i_17,750,750,47625,2375),
box(i_16,1000,250,51000,2875), box(i_15,1000,1000,51000,1000),
box(i_13,750,750,47625,875),   box(i_6,250,1250,51125,2875),
box(i_4,5250,3000,49375,1500), box(i_3,6500,4250,50000,875),
wire(i_11,150), wire(i_10,150), wire(i_9,150),
def_sym('#1'), def_sym('#2'), def_sym('#3'),
has('#1',box_4),    has('#3',i_8),    has('#3',i_7__1__1),
has('#3',i_7__1__2),    has('#3',i_7__1__3),    has('#3',i_7__2__1),
has('#3',i_7__2__2),    has('#3',i_7__2__3),    has('#1',box_3),
has('#1',box_2),    has('#1',box_1),    has('#1',poly_1),
has('#3',mcall_4),    has('#3',mcall_3),    has('#3',mcall_2),
has('#3',mcall_1),    has('#2',i_26), has('#2',i_25),
has('#2',i_24), has('#2',i_23), has('#2',i_20),
has('#3',i_18), has('#3',i_17), has('#3',i_16),
has('#3',i_15), has('#3',i_13), has('#3',i_12),
has('#3',i_11), has('#3',i_10), has('#3',i_9),
has('#3',i_6),  has('#3',i_4),  has('#3',i_3),
scale('#3',1,1), scale('#2',1,1), scale('#1',1,1),
call_sym(mcall_4,'#1'),    call_sym(mcall_3,'#1'),    call_sym(mcall_2,'#1'),
call_sym(mcall_1,'#1'),    call_sym(i_7__2__3,'#2'),
call_sym(i_7__2__2,'#2'), call_sym(i_7__2__1,'#2'),
call_sym(i_7__1__3,'#2'), call_sym(i_7__1__2,'#2'),
call_sym(i_7__1__1,'#2'), call_sym(i_8,'#2'),
transl(i_8,46750,0,1),    transl(i_7__1__1,38667,1500,1),
transl(i_7__1__2,42333,1500,1),    transl(i_7__1__3,46000,1500,1),
transl(i_7__2__1,38667,3000,1),    transl(i_7__2__2,42333,3000,1),
transl(i_7__2__3,46000,3000,1),    transl(mcall_1,41000,10250,1),
transl(mcall_2,41000,6625,1),    transl(mcall_3,43500,10250,1),
transl(mcall_4,43500,6625,1),
rotate(i_3,-6500,0,1),    rotate(i_4,-5250,0,1),
rotate(i_6,0,-250,1),    rotate(i_13,0,-750,1),
rotate(i_15,0,-1000,1),    rotate(i_16,-1000,0,1),
rotate(i_17,0,-750,1),    rotate(i_18,0,-750,1),

```

```

rotate(i_20,0,-800,1), rotate(box_4,0,-800,1),
rotate(i_23,0,600,1), rotate(box_3,0,600,1),
rotate(i_24,0,600,1), rotate(box_2,0,600,1),
rotate(i_26,0,600,1), rotate(box_1,0,600,1),
vertex(poly_1,200,1400,11), vertex(i_25,400,2800,11),
vertex(poly_1,200,1000,10), vertex(i_25,400,2000,10),
vertex(poly_1,600,1000,9), vertex(i_25,1200,2000,9),
vertex(poly_1,600,1100,8), vertex(i_25,1200,2200,8),
vertex(poly_1,750,1100,7), vertex(i_25,1500,2200,7),
vertex(poly_1,750,600,6), vertex(i_25,1500,1200,6),
vertex(poly_1,850,600,5), vertex(i_25,1700,1200,5),
vertex(poly_1,850,1250,4), vertex(i_25,1700,2500,4),
vertex(poly_1,600,1250,3), vertex(i_25,1200,2500,3),
vertex(poly_1,600,1400,2), vertex(i_25,1200,2800,2),
vertex(poly_1,200,1400,1), vertex(i_25,400,2800,1),
vertex(i_12,53250,3000,11), vertex(i_12,46750,3000,10),
vertex(i_12,46750,-1250,9), vertex(i_12,53250,-1250,8),
vertex(i_12,53250,2000,7), vertex(i_12,48750,2000,6),
vertex(i_12,48750,1250,5), vertex(i_12,48000,1250,4),
vertex(i_12,48000,2000,3), vertex(i_12,53250,2000,2),
vertex(i_12,53250,3000,1), vertex(i_11,47500,500,4),
vertex(i_11,47500,0,3), vertex(i_11,51000,0,2),
vertex(i_11,51000,500,1), vertex(i_10,49500,750,5),
vertex(i_10,50000,750,4), vertex(i_10,50000,2500,3),
vertex(i_10,51000,2500,2), vertex(i_10,51000,2750,1),
vertex(i_9,47500,2750,2), vertex(i_9,47500,3000,1),
layer(i_4,nd), layer(box_3,np), layer(box_2,np),
layer(i_24,np), layer(i_23,np), layer(i_12,nm),
layer(box_4,l10), layer(box_1,l4), layer(poly_1,l10),
layer(i_26,l4), layer(i_25,l10), layer(i_20,l10),
layer(i_18,l11), layer(i_17,l11), layer(i_16,l11),
layer(i_15,l11), layer(i_13,l11), layer(i_11,l10),
layer(i_10,l10), layer(i_9,l10), layer(i_6,l10),
layer(i_3,l40),
dummy]).

```

## APPENDIX O

### GENERIC DATA FROM CIF DURING REVERSE TRANSLATION

```

polygon_(i_12).      polygon_(i_25).      polygon_(i_26).
polygon_(i_24).      polygon_(i_23).      polygon_(i_20).
polygon_(i_18).      polygon_(i_17).      polygon_(i_16).
polygon_(i_15).      polygon_(i_13).      polygon_(i_6).
polygon_(i_4).polygon_(i_3).
wire_(i_11). wire_(i_10). wire_(i_9).
macro_def(t18).      macro_def(dev).      macro_call(i_7__2__3,t18).
macro_call(i_7__2__2,t18).      macro_call(i_7__2__1,t18).
macro_call(i_7__1__3,t18).      macro_call(i_7__1__2,t18).
macro_call(i_7__1__1,t18).      macro_call(i_8,t18).
macro_call(i_14__2__2,t18).      macro_call(i_14__2__1,t18).
macro_call(i_14__1__2,t18).      macro_call(i_14__1__1,t18).
scale_(100000000).
layer_(i_4,1). layer_(i_24,2).layer_(i_23,2).
layer_(i_12,3).layer_(i_26,4).layer_(i_25,10).
layer_(i_20,10).      layer_(i_18,11).      layer_(i_17,11).
layer_(i_16,11).      layer_(i_15,11).      layer_(i_13,11).
layer_(i_11,10).      layer_(i_10,10).      layer_(i_9,10).
layer_(i_6,10).layer_(i_3,40).layer_(i_32,16).
vertex_(i_3,53250,-1250,5).      vertex_(i_3,53250,3000,4).
vertex_(i_3,46750,3000,3).      vertex_(i_3,46750,-1250,2).
vertex_(i_3,53250,-1250,1).      vertex_(i_4,52000,0,5).
vertex_(i_4,52000,3000,4).      vertex_(i_4,46750,3000,3).
vertex_(i_4,46750,0,2).      vertex_(i_4,52000,0,1).
vertex_(i_6,51750,3000,5).      vertex_(i_6,50500,3000,4).
vertex_(i_6,50500,2750,3).      vertex_(i_6,51750,2750,2).
vertex_(i_6,51750,3000,1).      vertex_(i_13,48000,1250,5).
vertex_(i_13,47250,1250,4).      vertex_(i_13,47250,500,3).
vertex_(i_13,48000,500,2).      vertex_(i_13,48000,1250,1).
vertex_(i_15,51500,1500,5).      vertex_(i_15,50500,1500,4).
vertex_(i_15,50500,500,3).      vertex_(i_15,51500,500,2).
vertex_(i_15,51500,1500,1).      vertex_(i_16,51500,2750,5).
vertex_(i_16,51500,3000,4).      vertex_(i_16,50500,3000,3).
vertex_(i_16,50500,2750,2).      vertex_(i_16,51500,2750,1).
vertex_(i_17,48000,2750,5).      vertex_(i_17,47250,2750,4).
vertex_(i_17,47250,2000,3).      vertex_(i_17,48000,2000,2).
vertex_(i_17,48000,2750,1).      vertex_(i_18,49500,2750,5).
vertex_(i_18,48750,2750,4).      vertex_(i_18,48750,2000,3).
vertex_(i_18,49500,2000,2).      vertex_(i_18,49500,2750,1).
vertex_(i_20,2800,2800,5).      vertex_(i_20,2000,2800,4).

```

```

vertex_(i_20,2000,2000,3).    vertex_(i_20,2800,2000,2).
vertex_(i_20,2800,2800,1).    vertex_(i_23,1300,1300,5).
vertex_(i_23,1600,1300,4).    vertex_(i_23,1600,1900,3).
vertex_(i_23,1300,1900,2).    vertex_(i_23,1300,1300,1).
vertex_(i_24,1700,1300,5).    vertex_(i_24,1900,1300,4).
vertex_(i_24,1900,1900,3).    vertex_(i_24,1700,1900,2).
vertex_(i_24,1700,1300,1).    vertex_(i_26,1400,1300,5).
vertex_(i_26,1500,1300,4).    vertex_(i_26,1500,1900,3).
vertex_(i_26,1400,1900,2).    vertex_(i_26,1400,1300,1).
vertex_(i_25,400,2800,11).    vertex_(i_25,400,2000,10).
vertex_(i_25,1200,2000,9).    vertex_(i_25,1200,2200,8).
vertex_(i_25,1500,2200,7).    vertex_(i_25,1500,1200,6).
vertex_(i_25,1700,1200,5).    vertex_(i_25,1700,2500,4).
vertex_(i_25,1200,2500,3).    vertex_(i_25,1200,2800,2).
vertex_(i_25,400,2800,1).    vertex_(i_12,53250,3000,11).
vertex_(i_12,46750,3000,10). vertex_(i_12,46750,-1250,9).
vertex_(i_12,53250,-1250,8). vertex_(i_12,53250,2000,7).
vertex_(i_12,48750,2000,6).    vertex_(i_12,48750,1250,5).
vertex_(i_12,48000,1250,4).    vertex_(i_12,48000,2000,3).
vertex_(i_12,53250,2000,2).    vertex_(i_12,53250,3000,1).
vertex_(i_11,47500,500,4).    vertex_(i_11,47500,0,3).
vertex_(i_11,51000,0,2).    vertex_(i_11,51000,500,1).
vertex_(i_10,49500,750,5).    vertex_(i_10,50000,750,4).
vertex_(i_10,50000,2500,3).    vertex_(i_10,51000,2500,2).
vertex_(i_10,51000,2750,1).    vertex_(i_9,47500,2750,2).
vertex_(i_9,47500,3000,1).    vertex_(i_14_1_1,43500,6625,1).
vertex_(i_14_1_1,2,43500,10250,1). vertex_(i_14_2_1,41000,6625,1).
vertex_(i_14_2_2,2,41000,10250,1). vertex_(i_7_2_2,3,46000,3000,1).
vertex_(i_7_2_2,2,42333,3000,1). vertex_(i_7_2_1,1,38667,3000,1).
vertex_(i_7_1_1,3,46000,1500,1). vertex_(i_7_1_2,2,42333,1500,1).
vertex_(i_7_1_1,1,38667,1500,1). vertex_(i_8,46750,0,1).
vertex_(i_32,1400,2475,1).    width_(i_11,150).
width_(i_10,150).    width_(i_9,150).
orient_(i_32,0,0,0).
has_(dev,i_8).has_(dev,i_7_1_1).    has_(dev,i_7_1_2).
has_(dev,i_7_1_3).    has_(dev,i_7_2_1).    has_(dev,i_7_2_2).
has_(dev,i_7_2_3).    has_(t18,i_26).    has_(t18,i_25).
has_(t18,i_24).    has_(t18,i_23).    has_(t18,i_20).
has_(dev,i_18).    has_(dev,i_17).    has_(dev,i_16).
has_(dev,i_15).    has_(dev,i_13).    has_(dev,i_12).
has_(dev,i_11).    has_(dev,i_10).    has_(dev,i_9).
has_(dev,i_6).has_(dev,i_4).has_(dev,i_3).
has_(t18,i_32).    has_(dev,i_14_2_2).    has_(dev,i_14_2_1).
has_(dev,i_14_1_2).    has_(dev,i_14_1_1).
magnif_(i_14_2_2,2,0.5). magnif_(i_14_2_1,1,0.5).
magnif_(i_14_1_2,2,0.5). magnif_(i_14_1_1,1,0.5).
magnif_(i_14,0.5).    magnif_(i_32,0.1).
relative_orient(i_8).
relative_orient(i_14).
relative_magnif(i_14).
text_(i_32).
textval_(i_32,[84,49,56]).

```

```
h_just(i_32,center).  
v_just(i_32,middle).
```



## APPENDIX P

### CALMA OUTPUT DATA FROM REVERSE TRANSLATION

```
dbid(test.db,gds,'1.0','8/8/84:13:32').
content({fonts(gdsii:font.tx,2),
fonts(gdsii:font.tx,1),
generations(3),
db_user_unit(1e-05),
db_unit_meter(100000000),
struct(dev),      struct(t18),
bdry(i_25),      bdry(i_12),
path(i_9),      path(i_10),      path(i_11),
sref(i_8,t18),
aref(i_7,t18),  aref(i_14,t18),
text(i_32),
box(i_3),      box(i_4),      box(i_6),
box(i_13),     box(i_15),     box(i_16),
box(i_17),     box(i_18),     box(i_20),
box(i_23),     box(i_24),     box(i_26),
dtatyp(i_26,0), dtatyp(i_25,0), dtatyp(i_24,0),
dtatyp(i_23,0), dtatyp(i_20,0), dtatyp(i_18,0),
dtatyp(i_17,0), dtatyp(i_16,0), dtatyp(i_15,0),
dtatyp(i_13,0), dtatyp(i_12,0), dtatyp(i_11,0),
dtatyp(i_10,0), dtatyp(i_9,0),  dtatyp(i_6,0),
dtatyp(i_4,0), dtatyp(i_3,0),
xy(i_32,1400,2475,1), xy(i_9,47500,3000,1), xy(i_9,47500,2750,2),
xy(i_10,51000,2750,1), xy(i_10,51000,2500,2), xy(i_10,50000,2500,3),
xy(i_10,50000,750,4), xy(i_10,49500,750,5), xy(i_11,51000,500,1),
xy(i_11,51000,0,2), xy(i_11,47500,0,3), xy(i_11,47500,500,4),
xy(i_12,53250,3000,1), xy(i_12,53250,2000,2), xy(i_12,48000,2000,3),
xy(i_12,48000,1250,4), xy(i_12,48750,1250,5), xy(i_12,48750,2000,6),
xy(i_12,53250,2000,7), xy(i_12,53250,-1250,8), xy(i_12,46750,-1250,9),
xy(i_12,46750,3000,10), xy(i_12,53250,3000,11), xy(i_25,400,2800,1),
xy(i_25,1200,2800,2), xy(i_25,1200,2500,3), xy(i_25,1700,2500,4),
xy(i_25,1700,1200,5), xy(i_25,1500,1200,6), xy(i_25,1500,2200,7),
xy(i_25,1200,2200,8), xy(i_25,1200,2000,9), xy(i_25,400,2000,10),
xy(i_25,400,2800,11), xy(i_26,1400,1300,1), xy(i_26,1400,1900,2),
xy(i_26,1500,1900,3), xy(i_26,1500,1300,4), xy(i_26,1400,1300,5),
xy(i_24,1700,1300,1), xy(i_24,1700,1900,2), xy(i_24,1900,1900,3),
xy(i_24,1900,1300,4), xy(i_24,1700,1300,5), xy(i_23,1300,1300,1),
xy(i_23,1300,1900,2), xy(i_23,1600,1900,3), xy(i_23,1600,1300,4),
xy(i_23,1300,1300,5), xy(i_20,2800,2800,1), xy(i_20,2800,2000,2),
xy(i_20,2000,2000,3), xy(i_20,2000,2800,4), xy(i_20,2800,2800,5),
```

```

xy(i_18,49500,2750,1), xy(i_18,49500,2000,2), xy(i_18,48750,2000,3),
xy(i_18,48750,2750,4), xy(i_18,49500,2750,5), xy(i_17,48000,2750,1),
xy(i_17,48000,2000,2), xy(i_17,47250,2000,3), xy(i_17,47250,2750,4),
xy(i_17,48000,2750,5), xy(i_16,51500,2750,1), xy(i_16,50500,2750,2),
xy(i_16,50500,3000,3), xy(i_16,51500,3000,4), xy(i_16,51500,2750,5),
xy(i_15,51500,1500,1), xy(i_15,51500,500,2), xy(i_15,50500,500,3),
xy(i_15,50500,1500,4), xy(i_15,51500,1500,5), xy(i_13,48000,1250,1),
xy(i_13,48000,500,2), xy(i_13,47250,500,3), xy(i_13,47250,1250,4),
xy(i_13,48000,1250,5), xy(i_6,51750,3000,1), xy(i_6,51750,2750,2),
xy(i_6,50500,2750,3), xy(i_6,50500,3000,4), xy(i_6,51750,3000,5),
xy(i_4,52000,0,1), xy(i_4,46750,0,2), xy(i_4,46750,3000,3),
xy(i_4,52000,3000,4), xy(i_4,52000,0,5), xy(i_3,53250,-1250,1),
xy(i_3,46750,-1250,2), xy(i_3,46750,3000,3), xy(i_3,53250,3000,4),
xy(i_3,53250,-1250,5), xy(i_8,46750,0,1), xy(i_7,35001,0,1),
xy(i_14,46000,3000,1), xy(i_14,46000,10250,3), xy(i_14,41000,3000,2),
xy(i_7,35001,3000,3), xy(i_7,46000,0,2),
lyr(i_32,16), lyr(i_3,40), lyr(i_6,10),
lyr(i_9,10), lyr(i_10,10), lyr(i_11,10),
lyr(i_13,11), lyr(i_15,11), lyr(i_16,11),
lyr(i_17,11), lyr(i_18,11), lyr(i_20,10),
lyr(i_25,10), lyr(i_26,4), lyr(i_12,3),
lyr(i_23,2), lyr(i_24,2), lyr(i_4,1),
width(i_9,150), width(i_10,150), width(i_11,150),
columns(i_14,2), columns(i_7,3),
rows(i_14,2), rows(i_7,2),
vert_pres(i_32,middle),
horz_pres(i_32,center),
string(i_32,[84,49,56]),
mag(i_14,0.5), mag(i_32,0.1), mag(i_14,0.5),
has(dev,i_14), has(dev,i_7), has(t18,i_32),
has(dev,i_3), has(dev,i_4), has(dev,i_6),
has(dev,i_9), has(dev,i_10), has(dev,i_11),
has(dev,i_12), has(dev,i_13), has(dev,i_15),
has(dev,i_16), has(dev,i_17), has(dev,i_18),
has(t18,i_20), has(t18,i_23), has(t18,i_24),
has(t18,i_25), has(t18,i_26), has(dev,i_8),
dummy}).

```

## APPENDIX Q

### REVERSE TRANSLATION SYSTEM LOG, CIF TO CALMA

CProlog version 1.4d.edai  
[ Restoring file tran11.env ]

yes  
| ?- translate('cif.out','cifK.out','gds.out','gdsK.out','test.db',gds,'1.0','8/8/84:13:32').  
cif.out consulted 5180 bytes 1 sec.

>> T r a n s l a t e : V1.0 <<

dbid(test.db,cif,1.0,7/26/84:21:55)  
cifn.rul consulted 8572 bytes 1.6 sec.  
cifK.out consulted 2844 bytes 0.45 sec.  
fromdb(test.db,gds,1.4)  
toddb(test.db,cif,1.0)  
Start Up 7.88 sec.  
T=polygon_( _2049) 14 facts.  
T=wire_( _2049) 3 facts.  
T=macro_def( _2049) 2 facts.  
T=macro_call( _2049, _2050) 11 facts.  
T=scale_( _2049) 1 facts.  
T=layer_( _2049, _2050) 18 facts.  
T=vertex_( _2049, _2050, _2051, _2052) 105 facts.  
T=width_( _2049, _2050) 3 facts.  
T=orient_( _2049, _2050, _2051, _2052) 1 facts.  
T=has_( _2049, _2050) 29 facts.  
T=magnif_( _2049, _2050) 6 facts.  
T=relative_orient( _2049) 2 facts.  
T=relative_magnif( _2049) 1 facts.  
T=text_( _2049) 1 facts.  
T=textval_( _2049, _2050) 1 facts.  
T=h_just( _2049, _2050) 1 facts.  
T=v_just( _2049, _2050) 1 facts.  
T=tfont_( _2049, _2050) 0 facts.  
T=end_of_file 0 facts.  
200 facts total.  
Generic 9.15 sec.  
Keep 2.35 sec.  
Unload 2.23 sec.  
gdsout.rul consulted 5588 bytes 1.0167 sec.  
T=fonts( _5625, _5626) 2 facts.

T=generations(_5625) 1 facts.  
T=db_user_unit(_5625) 1 facts.  
T=db_unit_meter(_5625) 1 facts.  
T=strect(_5625) 2 facts.  
T=bdry(_5625) 2 facts.  
T=path(_5625) 3 facts.  
T=sref(_5625,_5626) 1 facts.  
T=aref(_5625,_5626) 2 facts.  
T=text(_5625) 1 facts.  
T=node(_5625) 0 facts.  
T=box(_5625) 12 facts.  
T=dtatyp(_5625,_5626) 17 facts.  
T=pathtype(_5625,_5626) 0 facts.  
T=texttype(_5625,_5626) 0 facts.  
T=nodetype(_5625,_5626) 0 facts.  
T=boxtype(_5625,_5626) 0 facts.  
T=xy(_5625,_5626,_5627,_5628) 101 facts.  
T=lyr(_5625,_5626) 18 facts.  
T=width(_5625,_5626) 3 facts.  
T=reflection(_5625) 0 facts.  
T=abs_mag(_5625) 0 facts.  
T=abs_angl(_5625) 0 facts.  
T=columns(_5625,_5626) 2 facts.  
T=rows(_5625,_5626) 2 facts.  
T=font_no(_5625,_5626) 0 facts.  
T=vert_pres(_5625,_5626) 1 facts.  
T=horz_pres(_5625,_5626) 1 facts.  
T=string(_5625,_5626) 1 facts.  
T=mag(_5625,_5626) 3 facts.  
T=angle(_5625,_5626) 1 facts.  
T=propval(_5625,_5626,_5627) 0 facts.  
T=current_rc(_5625,_5626,_5627) 0 facts.  
T=has(_5625,_5626) 21 facts.  
T=end_of_file 0 facts.  
199 facts total.

dbid(test.db,gds,1.0,8/8/84:13:32)

Phase 2 16.75 sec.

Output Keep 2.53 sec.

Total time is 42.35 sec.

yes

| ?- halt.

[ Prolog execution halted ]

## APPENDIX R

### KEPT FACTS FROM CALMA TO CIF TRANSLATION

```
fromdb('test.db',gds,'1.4').
toddb('test.db',cif,'1.0').
content({keep(angle(i_14,90)),
keep(xy(i_14,46000,8000,3)),
keep(xy(i_14,53250,3000,2)),
keep(xy(i_14,46000,3000,1)),
keep(xy(i_7,35000,3000,3)),
keep(xy(i_7,46000,0,2)),
keep(xy(i_7,35000,0,1)),
keep(columns(i_14,2)),
keep(columns(i_7,3)),
keep(rows(i_14,2)),
keep(rows(i_7,2)),
keep(aref(i_14,t18)),
keep(aref(i_7,t18)),
keep(db__user__unit(1e-05)),
keep(generations(3)),
keep(fonts('gdsii:font.tx',2)),
keep(fonts('gdsii:font.tx',1)),
keep(dtatyp(i_26,0)),
keep(dtatyp(i_25,0)),
keep(dtatyp(i_24,0)),
keep(dtatyp(i_23,0)),
keep(dtatyp(i_20,0)),
keep(dtatyp(i_18,0)),
keep(dtatyp(i_17,0)),
keep(dtatyp(i_16,0)),
keep(dtatyp(i_15,0)),
keep(dtatyp(i_13,0)),
keep(dtatyp(i_12,0)),
keep(dtatyp(i_11,0)),
keep(dtatyp(i_10,0)),
keep(dtatyp(i_9,0)),
keep(dtatyp(i_6,0)),
keep(dtatyp(i_4,0)),
keep(dtatyp(i_3,0)),
keep(map(minst_1,'#1')),
keep(map(t18,'#2')),
keep(map(dev,'#3')),
keep(vertex__(i_32,1400,2475,1)),
```

```

keep(layer_(i_32,16)),
keep(has_(t18,i_32)),
keep(relative_orient(i_8)),
keep(relative_orient(i_14)),
keep(relative_magnif(i_14)),
keep(orient_(i_32,0,0,0)),
keep(magnif_(i_14__2__2,0.5)),
keep(magnif_(i_14__2__1,0.5)),
keep(magnif_(i_14__1__2,0.5)),
keep(magnif_(i_14__1__1,0.5)),
keep(magnif_(i_14,0.5)),
keep(magnif_(i_32,0.1)),
keep(v_just(i_32,middle)),
keep(h_just(i_32,center)),
keep(textval_(i_32,[84,49,56])),
keep(text_(i_32)),
keep(ncall(i_14__2__2,mcall_1)),
keep(ncall(i_14__2__1,mcall_2)),
keep(ncall(i_14__1__2,mcall_3)),
keep(ncall(i_14__1__1,mcall_4)),
keep(item_inst(minst_1,poly_1,i_25)),
keep(item_inst(minst_1,box_1,i_26)),
keep(item_inst(minst_1,box_2,i_24)),
keep(item_inst(minst_1,box_3,i_23)),
keep(item_inst(minst_1,box_4,i_20)),
keep(macro_inst(t18,0.5,minst_1)),
keep(scale_(100000000)),
keep(macro_call(i_14__2__2,t18)),
keep(has_(dev,i_14__2__2)),
keep(macro_call(i_14__2__1,t18)),
keep(has_(dev,i_14__2__1)),
keep(macro_call(i_14__1__2,t18)),
keep(has_(dev,i_14__1__2)),
keep(macro_call(i_14__1__1,t18)),
keep(has_(dev,i_14__1__1)),
dummy]].

```

## References

- [IPC77] "Printed Board Description in Digital Form," ANSI/IPC-D-350B, Institute of Printed Circuits American National Standards Institute (August 1977).
- [HP80] "TESTAID III Programming and Operating Manual," 91075-93018, Hewlett-Packard Company, Loveland, Colorado (January 1980). Order from: Hewlett-Packard Company; P.O. Box 301; Loveland, Colorado 80537.
- [CALM82] "GDSII Reference Guide - Release 4.0," , The CALMA Company, Woburn, Mass. (October 1982).
- [TDL83] "TDL Preprocessor User and Reference Manual," , COMSAT General Integrated Systems, Austin, Texas (August 1983). (3rd Quarter Release) (CALMA now owns the rights to TEGAS/TDL).
- [CVPC83] "CADD5 4 Printed Circuit / Electrical Schematic User Guide," 001-00240, Computervision Corp., Woburn, Mass. (April 1983). Order from: Computervision Corp.; 100 Commerce Way; Woburn, Mass 01801 - ATTN: Technical Publications.
- [CVDB83] "CADD5 4 Data Base Reference - Software Revision 2-03-A," 001-00016-001, Computervision Corp., Woburn, Mass. (April 1983). Order from: Computervision Corp.; 100 Commerce Way; Woburn, Mass 01801 - ATTN: Technical Publications.
- [Soft84] *System 1022™ Data Base Management System, User's Reference Manual*, Software House, Cambridge, MA (October 1984).
- [EDIF84] "EDIF Electronic Design Interchange Format, Version 0.8," , EDIF Technical Committee (May 14, 1984). (Preliminary Specification) (Committee comprised of representatives from Daisy Systems, Mentor Graphics, Motorola, National Semiconductor, Tektronix, and Texas Instruments).

- [Brac83a] Brachman, Ronald J., "What IS-A Is and Isn't: An Analysis of Taxonomic Links in Semantic Networks," *Computer* 16(10), pp.30-36 (October 1983).
- [Brac83b] Brachman, Ronald J., Richard E. Fikes, and Hector J. Levesque, "Krypton: A Functional Approach to Knowledge Representation," *Computer* 16(10), pp.67-73 (October 1983).
- [Caro83] Caro, Marilyn L., "Feasibility of a Knowledge-Based Approach to CAD/CAM Data Transfer," , University of California, Los Angeles (Fall 1983). (A Master's Comprehensive Examination).
- [Ciam76a] Ciampi, P. L. and J. D. Nash, "Concepts in CAD Data Base Structures," pp. 290-294 in *Proceedings 13th Design Automation Conference* (June 1976).
- [Ciam76b] Ciampi, P. L., A. D. Donovan, and J. D. Nash, "Control and Integration of a CAD Data Base," pp. 285-289 in *Proceedings 13th Design Automation Conference* (June 1976).
- [Cloc81] Clocksin, W. F. and C. S. Mellish, *Programming in Prolog*, Springer-Verlag, Berlin (1981).
- [Dahl83] Dahl, Veronica, "Logic Programming as a Representation of Knowledge," *Computer* 16(10), pp.106-111 (October 1983).
- [Date82] Date, C. J., *An Introduction to Database Systems*, Addison-Wesley, Reading, Mass. (February 1982). Volume I, 3rd. Edition.
- [Gutt82] Guttman, A. and M. Stonebraker, "Using a Relational Database Management System for Computer Aided Design Data," *Quarterly Bulletin of the IEEE Comp. Soc. Tech. Comm. on Database Engineering* 5(2), pp.21-28 (June 1982).
- [Hask82] Haskin, R. and R. Lorie, "Using a Relational Database System for Circuit Design," *Quarterly Bulletin of the IEEE Comp. Soc. Tech. Comm. on Database Engineering* 5(2), pp.10-14 (June 1982).
- [Kawa78] Kawano, Ietoshi, Hiroshi Fukushima, and Takeshi Numata, "The Design of a Data Base Organization for an Electronic Equipment DA System," pp. 167-175 in *Proceedings 15th Design Automation Conference* (June 1978).



- [Lacr81] Lacroix, M., "Data Structures for CAD Object Description," pp. 653-659 in *Proceedings 19th Design Automation Conf.* (1981).
- [McCa83] McCalla, G. and N. Cercone, "Guest Editor's Introduction: Approaches to Knowledge Representation," *Computer* **16**(10), pp.12-18 (October 1983).
- [Mead80] Mead, Carver and Lynn Conway, *Introduction to VLSI Systems*, Addison-Wesley, Reading, Mass. (1980). (Section 4.5, pp. 115-127).
- [Mylo83] Mylopoulos, J. and T. Shibahara, "Building Knowledge-Based Structures: The PSN Experience," *Computer* **16**(10), pp.83-89 (October 1983).
- [Rodr81] Rodriguez-Ortiz, G., *The ELKA Model: Approach to the Design of Database Conceptual Models*, University of California, Los Angeles, Los Angeles, California (1981). PhD Dissertation.
- [IGES80] Nagel, Roger N. Ph.D., Walt W. Braithwaite, and Philip R. Kennicott, Ph.D., *Initial Graphics Exchange Specification (IGES), Version 1.0*, U.S. Dept. of Commerce, Nat'l. Bureau of Standards (January 1980).
- [Scow82] Scowen, R. S., "IGES -- A Critical Review and Some Suggestions," pp. 47-58 in *CAD Systems Framework*, ed. F. M. Lillehagen, North-Holland, Amsterdam (June 15-17 1982). Proc. IFIP WG 5.2 Working Conference on CAD Systems Framework.
- [Ston80] Stonebraker, M. R. and K. Keller, "Embedding Expert Knowledge and Hypothetical Data Bases into a Data Base System," pp. 58-66 in *Proceedings ACM-SIGMOD Conference*, Santa Monica, California (May 14-16, 1980).
- [Such79] Sucher, Daniel J. and Donald F. Wann, "A Design Aids Data Base for Digital Components," pp. 414-420 in *Proceedings 16th Design Automation Conference* (June 1979).
- [Vall75] Valle, Giorgio, "Relational Data Handling Techniques in Integrated Circuit Mask Layout Procedures," pp. 407-413 in *Proceedings 12th Design Automation Conference* (June 1975).
- [Webb83] Webber, B. L., "Logic and Natural Language," *Computer* **16**(10), pp.43-46 (October 1983).

- [Wilm79] Wilmore, James A., "The Design of an Efficient Data Base to Support an Interactive LSI Layout System," pp. 445-451 in *Proceedings 16th Design Automation Conference* (June 1979).
- [Wong79] Wong, S. and W. A. Bristol, "A Computer-Aided Design Data Base," pp. 398-405 in *Proceedings 16th Design Automation Conference* (June 1979).
- [Wood83] Woods, William A., "What's Important About Knowledge Representation," *Computer* **16**(10); pp:22-27 (October 1983).