INFORMATION TO USERS

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

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

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

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

Games in Open Systems Verification and Synthesis

by

Yiu-Chung Mang

M. Eng. (Oxford University, England) 1995

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

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:
Professor Thomas A. Henzinger, Chair
Professor Robert K. Brayton
Professor John Steel

Spring 2002

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

Games in Open Systems Verification and Synthesis

Abstract

Games in Open Systems Verification and Synthesis

by

Yiu-Chung Mang

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Thomas A. Henzinger, Chair

This dissertation investigates game-theoretic approaches to the algorithmic analysis of concurrent, reactive systems. A concurrent system comprises a number of components working concurrently; a reactive system maintains an ongoing interaction with its environment. Traditional approaches to the formal analysis of concurrent reactive systems usually view the system as an unstructured state-transition graphs; instead, we view them as collections of interacting components, where each one is an open system which accepts inputs from the other components. The interactions among the components are naturally modeled as games.

Adopting this game-theoretic view, we study three related problems pertaining to the verification and synthesis of systems. Firstly, we propose two novel game-theoretic techniques for the model-checking of concurrent reactive systems, and improve the performance of model-checking. The first technique discovers an error as soon as it cannot be prevented, which can be long before it actually occurs. This technique is based on the key observation that "unpreventability" is a local property to a module: an error is unpreventable in a module state if no environment can prevent it. The second technique attempts to decompose a model-checking proof into smaller proof obligations by constructing abstract modules automatically, using reachability and "unpreventability" information about the concrete modules. Three increasingly powerful proof decomposition rules are proposed and we show that in practice, the resulting abstract modules are often significantly smaller than the concrete modules and can drastically reduce the space and time requirements for verification. Both techniques fall into the category of compositional reasoning.

Secondly, we investigate the composition and control of synchronous systems. An essential property of synchronous systems for compositional reasoning is non-blocking. In the composition of synchronous systems, however, due to circular causal dependency of input and output signals, non-blocking is not always guaranteed. Blocking compositions of systems can be ruled out semantically, by insisting on the existence of certain fixed points, or syntactically, by equipping systems with types, which make the dependencies between input and output signals transparent. We characterize various typing mechanisms in game-theoretic terms, and study their effects on the controller synthesis problem. We show that our typing systems are general enough to capture interesting real-life synchronous systems such as all delay-insensitive digital circuits. We then study their corresponding single-step control problems — a restricted form of controller synthesis problem whose solutions can be iterated in appropriate manners to solve all LTL controller synthesis problems. We also consider versions of the controller synthesis problem in which the type of the controller is given. We show that the solution of these fixed-type control problems requires the evaluation of partially ordered (Henkin) quantifiers on boolean formulas, and is therefore harder (nondeterministic exponential time) than more traditional control questions.

Thirdly, we study the synthesis of a class of open systems, namely, *uninitialized state machines*. The sequential synthesis problem, which is closely related to Church's solvability problem, asks, given a specification in the form of a binary relation between input and output streams, for the construction of a finite-state stream transducer that converts inputs to appropriate outputs. For efficiency reasons, practical sequential hardware is often designed to operate without prior initialization. Such hardware designs can be modeled by uninitialized state machines, which are required to satisfy their specification if started from any state. We solve the sequential synthesis problem for uninitialized systems, that is, we construct uninitialized finite-state stream transducers. We consider specifications given by LTL formulas, deterministic, nondeterministic, universal, and alternating Büchi automata. We solve this *uninitialized synthesis problem* by reducing it to the well-understood initialized synthesis problem. While our solution is straightforward, it leads, for some specification formalisms, to upper bounds that are exponentially worse than the complexity of the corresponding initialized problems. However, we prove lower bounds to show that our simple solutions are optimal for all considered specification formalisms. The lower bound proofs require nontrivial generic reductions.

Professor Thomas A. Henzinger
Dissertation Committee Chair

to Wendy ...

Contents

List of Figures

List of Tables

Acknowledgments

I would like to express my sincere gratitude towards my advisor Prof. Thomas A. Henzinger. I thank him for his generous financial support; for teaching me everything necessary to carry out research work; for his vision and inspiration on the future of formal methods; and for his patience and encouragement during the ups and downs of my entire graduate life.

My gratitude also goes to Luca de Alfaro. We spent many hours working on various research problems; in fact, most results presented in this dissertation were obtained in collaboration with him. He is quick, and is always enthusiastic about every idea. Luca is also a very good friend; He introduced me to photography and Italian cooking. It is wonderful to be able to work with him.

A large portion of the first half of my graduate life went to the development of the model-checker MOCHA, a joint effort with Shaz Qadeer and Sriram Rajamani. It is a great experience working with them. Developing a large software system is a huge undertaking; without them, it would not be possible to have this tool for conducting all the experiments in this dissertation. I thank Sriram Krishnan and Orna Kupferman, whom I have been fortunate enough to work with on one of the problems presented in this dissertation. I also thank Rupak Majumdar and Jean-Francois Raskin for the fun when working on one of the papers.

I gained my first industrial experience on model-checking during my two summer internships at the Intel Strategic CAD Labs in Hillsboro, Oregon. I thank Pei-Hsin Ho for being my mentor at SCL. He taught me much about model-checking, and formal verification in general, from the industrial perspective.

My stay at Berkeley has been enjoyable. I thank all the members of the Berkeley CAD group for contributing to such an inspiring and pleasant research environment. I thank Michael Shilman for organizing the CAD picnic with me; Subarnarekha Sinha for sharing my burden (both physically and financially) in organizing the CAD seminar. I also thank her for listening patiently to all my grumble about life.

Finally, I am very grateful to my parents and Wendy, who have endured my pursuit of doctoral degree. A million words cannot express my gratitude towards them.

Chapter 1

Introduction

My work is a game, a very serious game.
- M.C. Escher (1898 - 1972)

1.1 Background

1.1.1 Verification and synthesis

This dissertation investigates game-theoretic approaches to the algorithmic analysis of concurrent reactive systems. A concurrent system comprises a number of components working concurrently: a reactive system maintains an ongoing interaction with its environment. Such systems abound in real-life: digital circuits, microprocessors, communication protocols, control of nuclear reactors, to name a few. Logical correctness in the design of such systems are of paramount importance: in the case of digital circuits and microprocessors, logical errors may result in loss of millions of dollars [Coe95]: in the case of nuclear reactors, loss of human lives. Hence, for decades, producing correct reactive systems have been the main subject among engineers and computer scientists. Formal methods have evolved to be one of the most prominent tools in the design of correct systems.

Correct systems can be designed by hand followed by verification: they can also be synthesized automatically from the specifications. In formal methods, both the systems and the specifications are given in some formal languages such as modal logic or automata that accept infinite words. In systems verification, the most widely used technique is extensive simulation. This technique serves as a good method to falsify the design: however, correctness can never be guaranteed. The second approach is based on deductive meth-

ods. including theorem proving (cf. [Gor88. Pau94]). term-rewriting (cf. [DJ89. Klo91]) and proof checking (cf. [DFH+93]). These techniques generally regard the systems as a set of axioms and attempt to prove that the specifications are theorems of these axioms. While these techniques. in principle. can handle arbitrary complicated designs. they often require much manual intervention and a great deal of training in mathematical logic.

The third approach to the verification of reactive systems is based on algorithmic methods. These include temporal-logic model-checking [QS81. CES86. VW94]) and refinement checking. In this approach. the systems are modeled as state-transition systems (Kripke structures). while the specifications can be specified in modal logic formulas. ω-automata (automata that accepts infinite words) or even another state-transition system (in the case of refinement checking). The model-checking problem then asks. given a system and a specification. if the specification holds in the system. The refinement-checking problem(cf. [Raj99]). asks. given a system P (implementation) and a system Q (abstract specification). if the behaviors of P are included in the behaviors of Q. The most accepted notions of "inclusion" are trace-containment and simulation relation [Mil71]. In essence. both temporal-logic model-checking and refinement checking consist in exhaustive searches in the state-space of the state-transition systems and can be highly automated: hence they are well-suited for the automatic verification of finite-state system.

Popular modal logics for specifying nonterminating reactive systems include Linear Temporal Logic (LTL) [Pnu77]. Computation Tree Logic (CTL) [CE81. QS81] and modal μ-calculus [Koz83]. LTL is a linear-time logic for expressing properties about linear. infinite computation traces generated from the state-transition systems. Its syntax has. in addition to atomic propositions and oolean operators. a number of modal operators including $\bigcirc\varphi$ (φ holds in the successor state). $\Box\varphi$ (φ holds in all subsequent states) and $\varphi\,\mathcal{U}\,\psi$ (φ holds in the subsequent states until ψ holds). where φ and ψ are LTL formulas. On the other hand. CTL expresses properties about the infinite computation *trees* of the systems. It allows expression of universal as well as existential properties such as "from all states. there exists a path to the reset state." Unlike LTL. its modal operators are quantified by either the universal $\forall$ (all successors) or the existential $\exists$ (some successors) quantifier. The μ-calculus is a modal logic for specifying properties of reactive systems modeled as Kripke structures. The syntax of propositional μ-calculus consists of boolean operators and modalities including $\forall\bigcirc$ (for all successors) and $\exists\bigcirc$ (exists some successors). as well as the greatest and least fixpoint operators. Both LTL and CTL are fragments of μ-calculus since the modal operators $\Box$

and $\mathcal{U}$ can both be encoded in μ-calculus using fixpoints.

The main limiting factor in the algorithmic analysis of finite-state systems is the state-explosion problem: that is, the number of states to be explored can be exponential in the number of concurrent components in the system. To see this, consider a system P with m components, each having at most n states. Then the system P may have n^m states. In fact, for most popular system description languages, both temporal logic model-checking and refinement checking are known to be PSPACE-complete in the size of the description of the reactive systems [SC85, AH98]. To combat the state-explosion problem, a number of techniques have been proposed. For example, the *symbolic* techniques [McM93] represent the states and transitions of the systems implicitly using some succinct data structures, such as decision diagrams [Bry86] or boolean formulas [BCCZ99]; *abstract interpretation* [Dam96, GS97] abstracts away the nonessential part of the systems and therefore reduces the state space to be explored; *compositional or modular verification* attempts to prove the properties of a system by reasoning about behavior of the individual components. One notable example of compositional verification is assume-guarantee reasoning [Sta85, CLM89, AL95, AH99, McM97, HQR00]; the essential idea is to verify each component assuming the presence of a correct environment. This assumption will then be discharged by proving that it is an abstraction of the real environment.

Correct systems can also be synthesized from specifications. Automatic construction of correct systems has attracted much attention in several branches of engineering. For example, in Very Large Scale Integrated (VLSI) circuits where finite state machines constitute the basic building blocks of such circuits, the sequential synthesis problem asks, given a specification characterizing the set of permissible implementation, for the construction of a finite state machine M allowed by the specification such that M satisfies some optimality criteria [dM94]. In control theory, discrete-event systems (DES) [RW89] model the sequencing of events of dynamic systems, such as the arrival or departure of a customer in a queue, the start or completion of a task in an operating system, or the transmission or receipt of a packet in a communication system. The control problem of an DES P asks for synthesizing a controller C (or *supervisor*) such that the composed, closed-loop system $P\|C$, where $\|$ denotes *lock-step composition*, meets certain safety criteria such as stability, mutual exclusion and data consistency. The control problem for DES is highly related to the synthesis problem, and they share much similarity in the formalisms and solutions.

Informally, the synthesis problem can be stated as follows: a stream requirement is

a specification on the input-output behavior of the synthesized system, and the realizability problem asks, given an specification ψ (expressed as modal logic formulas or ω-automata) over sets of input and output signals, whether there exists a reactive system that assigns to every possible input sequence an output sequence so that the resulting computation satisfies ψ. If the answer is Yes, then the synthesis problem asks for the construction of such a system. The synthesis problem, closely related to Church's solvability problem [Chu62], was solved by Büchi and Landweber for stream requirement given as sequential calculus [BL69]. Since then, several algorithmic solutions have been proposed for other specification formalisms including LTL [ALW89, PR89a, PR89b], CTL [KMTV00] and μ-calculus [KV00].

In general, the synthesis problem is harder than the verification problem. For finite-state systems, while LTL verification problem is in PSPACE, the LTL synthesis problem is 2EXPTIME-complete [PR89a]. One source of hardness, like verification, is the state-explosion problem. Also, the synthesized systems are usually required to meet certain minimal behavior such as deadlock-freedom (nonblocking): that is, any finite trace of the synthesized system should always be extensible to a longer finite trace that belongs to the system. The nonblocking requirement is essential in several verification methodology such as assume-guarantee reasoning and simulation generation [KMP98]. In addition, synthesis of controllers often suffers from the *observability* problem [CDFV88, LW88b], in which some events of the system may not be observable by the controller. While the synthesis of a two-component system under partial observation (incomplete information) for LTL may incur exponential costs over that under complete information [RG95, Var95, KS97], the synthesis of a multi-component system (such as decentralized supervision [LW88a, LW90, RW92]) under incomplete information is in general undecidable [Rei84, PR90]. Practical remedies are often architectural in nature: they include hierarchical and modular synthesis [WW96] and various domain specific heuristics [Oku86, CL90, PRSV98].

1.1.2 Open systems and Games

In verification, the inputs from the environment to a system are treated as part of the system, which assigns nondeterministic values to the inputs. We call these systems *closed*. In synthesis, however, systems are treated as *open*: they interact with their environments and cannot constrain their environments' behaviors. Hence, a stream transducer, an open-loop DES, and a component in a closed system are all open systems. Open systems

possess interesting properties such as realizability. Besides, researchers have proposed other properties related to open systems. We give some examples below.

Receptiveness [Dil89, AL93, GSSAL98]. Given a reactive system, specified by a set of *safe* computation traces and a set of *live* computations (typically, expressed by an LTL formula), the receptiveness problem is to determine whether every finite safe computation can be extended to an infinite live computation irrespective of the behavior the environment.

Module checking [KV96]. Given an open system and a CTL formula φ, the module-checking problem is to determine if, no matter how the environment restricts the external choices, the system satisfies φ.

Well-formedness of component interfaces [dAH01a, dAH01b, CdAHM02]. Given an *interface* F specified as a state-transition system, where each transition of F are annotated with an *input assumption*, that is, a set of input signals that F accepts; and an *output guarantee*, that is a set of output signals that F outputs if the input assumption is satisfied. An error state of F is a state where the environment of F cannot produce an input that F accepts. The interface F is well-formed iff the *error* state is not reachable under *some* environment. Well-formedness is less restrictive than receptiveness, which requires F to accept all inputs under *all* environments, and it permits the reasoning about components that have bidirectional ports [CdAHM02].

Alternating temporal-logic [AHK97]. Given a Alternating temporal-logic (ATL) specification φ and a concurrent system $P = P_1 \| P_2$ comprising two interacting components (*agents*) P_1 and P_2, the ATL verification problem asks if the system P satisfies the specification φ. Questions that one can asks in ATL include: Does the agent P_1 (say, a processor in a multi-processor system) have a strategy to meet a requirement (say, the eventual ownership of the bus) no matter what the other agents (say, processors) do? Does the agent P_1 (say, a traffic-light controller) have a strategy to avoid violation of some requirement (say, simultaneous green lights at an intersection) no matter what the other agents (say, vehicles on the road) do?

ATL extends the logic CTL: while CTL allows expression of universal and existential properties of closed systems, ATL also allows expression of adversarial and protagonist properties. The syntax of ATL generalizes the path quantifiers of CTL from $\forall$ and $\exists$ to $\langle\langle P \rangle\rangle \varphi$ (P can ensure that, no matter what the environment of P does, φ holds), and $[\![P]\!]\varphi$ (P cannot avoid φ), where P is an open system. ATL has been applied in several areas of computer sciences such as correctness analysis of security protocols [KR00, KR01] and

feature interactions [CRS00].

All the stated problems arise from the interaction among the various components of the systems, and possess a game-like ($\forall\exists$) interpretation: Does a solution exists no matter what the adversary (e.g. environment) does? This is in contrast to the closed system verification problem which asks for the solution of an $\exists$ problem: for instance, does the system as a whole have a path to reach a bad state? Interactions among the open systems can be modeled as a game, while the participating systems are modeled as the players, and the specification modeled as the winning condition of the game. Then, solving the above stated problems amounts to solving the game, that is, deciding the winner of the game.

We adopt this game-theoretic view of systems in reasoning about complex systems that consists of components: often this view gives new insights in devising new model-checking algorithms. As an example, if one proves that a component P has a strategy to reach a goal no matter what its environment does, then the system $P\|Q$ consisting of P and Q has a path to reach that goal. As a result, there is no need to compute the composition of the components, and state-explosion can be avoided. In this dissertation, we study ways to improve the efficiency and capacity of LTL symbolic model-checking of closed, reactive systems, by reasoning about their components as open systems.

We are interested in the games arising from synchronous systems since they represent a large class of interesting and practical systems, such as electronic circuits. Solving the games arising from synchronous systems often require the solutions to the *single-step games*, which informally asks if the system or its environment can reach a specified goal (e.g. a specified state) in exactly one step. For example, consider the following *multi-step*, reachability game: Does the system have a strategy to reach a specified state s eventually. We can solve the problem by iteratively asking: Is the system already in the state s? Does the system have a strategy to reach s in one step? Does the system have a strategy to reach s in two steps? Does the system have a strategy to reach s in k steps? Clearly, for finite state systems, the number k of questions to ask is bounded by the number of states in the systems.

Hence, for synchronous systems, the algorithm for solving a game consists of two orthogonal parts: a routine for solving single-step games, and a "control structure" that iterates the routine. While the control structure is generally dependent only upon the specification, and indeed for some specification formalisms such as μ-calculus, the control structure is readily obtained from the specification, the single-step games arising from different

types of synchronous systems require different solutions: for Moore systems whose outputs are independent of the current inputs, their single-step games can be solved by evaluating quantified boolean formulas (QBF) with only one $\forall\exists$ switch; whereas for Mealy systems whose outputs may depend on the current inputs, solving the corresponding single-step games may require the evaluation of QBF formulas with multiple switches. The classification of synchronous systems and the study of their single-step games are the main theme of this dissertation.

1.1.3 Games in logic and concurrency theory

Formal verification and synthesis have their roots in mathematical logic and concurrency theory, where games play an important role. We briefly mention some examples of such games which we may touch upon in this dissertation.

Model-checking of μ-calculus. The model-checking problem of μ-calculus asks, given a μ-calculus formula φ and a Kripke structure K, if φ holds in K. The μ-calculus model-checking problem was first studied in [EL86], and it can be casted as the *model-checking game* [Sti95].

μ-calculus Model-checking and Parity games. The exact complexity of the μ-calculus model-checking problem remains unknown after more than a decade of active research. The best known complexity is due to [Jur00], who showed that the problem is UP $\cap$ co-UP. It is an open problem if μ-calculus model-checking can be done in polynomial time. On the other hand, the μ-calculus model-checking game has been shown to be polynomially reducible to a variety of games, including parity game [Sti95, EJS93], mean payoff game [Pur95, Sti95, Jur00], discounted payoff game [ZP96] and stochastic game [Con92, Pur95, Sti95, ZP96], and the search for polynomial time solutions to these games is under active research among logicians, computer scientists and game-theorists.

Equivalence of Kripke structures. Two Kripke structures are indistinguishable by any μ-calculus formula if and only if they are bisimilar. Bisimilarity can be characterized by the *bisimilation game* [Tho93, Sti97]. In essence, the bisimilation game is similar to the the Ehrenfeucht-Fraïssé game for showing elementary equivalence of first-order structures.

Decidability of second-order logic. To prove the decidability of the theories of monadic second order logic with n-successors (SnS), Rabin introduced what is now known as the Rabin tree automata [Rab69]. The proof entails in showing that Rabin tree automata

are closed under disjunction, projection and complementation. While proving the closure of Rabin tree automata under the disjunction and projection is relatively straight-forward, the proof of closure under complementation, also known as the Complementation Lemma, is very complicated and the original proof given by Rabin is highly involved. Many simplified proofs have been suggested [HR72, MS84, Muc84] since then. One of the best known proofs uses infinite games [GH82].

Equivalence of μ-calculus and Rabin Tree Automata. Research on μ-calculus gives new insight to mathematical logic. In automatic-theoretic methods, one associates, for each formula of a temporal logic, a finite automaton that recognizes exactly those infinite structures in which the formula holds. Then the satisfiability problem for the formula can be reduced to checking nonemptiness of the associated automata. When the logic is μ-calculus, one can associate each μ-calculus formula with a Rabin tree automaton. It has been shown that μ-calculus on infinite trees is expressively equivalent to that of Rabin tree automata [EJ91]: given a μ-calculus formula φ, there exists a Rabin tree automata U (and vice versa), such that an infinite tree S is a model of φ iff S is accepted by U. Since μ-calculus is trivially closed under complementation, the equivalence in expressive power provides an alternative proof of the Complementation Lemma.

Games are also used in other branches of computer science such as Finite Model Theory and Descriptive Set Theory. Note that the games used in logic and concurrency are usually turn-based (each turn either play moves) and the single-step games are trivial, whereas the games for characterizing open systems usually involve concurrent movements of the players, and hence have complicated single-step games.

1.2 Thesis Overview

This dissertation studies open systems using game-theoretic means. We study the following three areas of open systems:

- Connection between open and closed systems verification: in particular, we study techniques to improve the performance of traditional model-checking using game-theoretic techniques pertaining to open systems verification. We view a system as a collection of interacting components. Each interacting component is an open system. Based on this view, we propose two novel game-theoretic verification algorithms.

namely, *early error detection* and *automatic proof decomposition*. They will be presented in Chapters 3 and 4, respectively. They are based on the materials presented in [AdAHM99, dAHM00b]. The games play on the state-space of the components and the outcome of the games are used to infer properties about the complete system: hence our algorithms fall into the category of compositional reasoning.

- Composition of *synchronous* open systems (systems that proceed in lock-step) and single-step control problem: a controller provides input to an open system such that the composed system behaves in a way that satisfies the specification. The output signals of the controller depends on the input and output signals of the system and some output signals of the system may not be observed. For instance, an input of the controller cannot observe an output of the system that it does not depend on. We provide game-theoretic definitions of increasingly stronger notions of input-output dependencies, or *types*, and show that systems equipped with the strongest type models exactly all constructive, delay-insensitive digital circuits.

  We then study the problem of single-step controller synthesis (games) for systems equipped with various types and gives the complexity bounds. This will be presented in Chapter 5. Some of this material was presented in [dAHM00a, dAHM01].

- Synthesis of *uninitialized systems*: that is, open systems whose behaviors are independent of the initial states. These systems naturally presents in practical hardware, and form an interesting class of systems both practically and theoretically. We study uninitialized systems using games and the results will be presented in Chapter 6. It is based on [HKKM02].

The chapters can be read independently (except for Chapter 2 which provides all the necessary preliminaries used in all the subsequent chapters). The chapter summaries are given below.

1.2.1 Early error detection in model-checking

Any formal method or tool is almost certainly more often applied in situations where the outcome is failure (a counterexample) rather than success (a correctness proof). In Chapter 3 we present a method for symbolic model-checking that can lead to significant time and memory savings for model-checking runs that fail, while occurring only a small

overhead for model-checking runs that succeed. Our method discovers an error as soon as it cannot be prevented, which can be long before it actually occurs: for example, the violation of an invariant may become unpreventable many transitions before the invariant is violated.

The key observation is that "unpreventability" is a local property of a single module: an error is unpreventable in a module state if no environment can prevent it. Therefore, unpreventability is inexpensive to compute for each module, yet can save much work in the state exploration of the global, compound system. Based on different degrees of information available about the environment, we define and implement several notions of "unpreventability," including the standard notion of uncontrollability from discrete-event control.

1.2.2 Automatic assume-guarantee proof decomposition

Modular techniques for automatic verification attempt to overcome the state-explosion problem by exploiting the modular structure naturally present in many system designs. Unlike other tasks in the verification of finite-state systems, current modular techniques rely heavily on user guidance. In particular, the user is typically required to construct module abstractions that are neither too detailed as to render insufficient benefits in state exploration, nor too coarse as to invalidate the desired system properties. In Chapter 4, we propose a method where abstract modules are constructed automatically, using reachability and controllability information about the concrete modules. This allows us to leverage automatic verification techniques by applying them in layers: first we compute on the state spaces of system components, then we use the results for constructing abstractions, and finally we compute on the abstract state space of the system. Our experimental results indicate that if reachability and controllability information is used in the construction of abstractions, the resulting abstract modules are often significantly smaller than the concrete modules and can drastically reduce the space and time requirements for verification.

1.2.3 The composition and control of synchronous systems

A fundamental question in the study of compositional verification of systems is the semantics of composition. Reactive systems should be non-blocking, in the sense that every state should have at least one successor state [BG88, Hal93, Kur94, Lyn96]. Non-blocking is essential for compositional techniques such as assume-guarantee reasoning. In control, non-

blocking means that the controller should never prevent the plant from moving. However in the composition of synchronous processes, non-blocking is not guaranteed in the product. For example, the composition of an inverter component $y = \neg x$ and an identity component $y = x$ is blocking. Blocking compositions of processes can be ruled out semantically, by insisting on the existence of certain fixed points, or syntactically, by equipping processes with types, which make the dependencies between input and output signals transparent. In Chapter 5, we classify various typing mechanisms and study their effects on the control problem. The semantics of types are given in game-theoretic terms.

A static type enforces fixed, acyclic dependencies between input and output ports. For example, synchronous hardware without combinational loops can be typed statically. A dynamic type may vary the dependencies from state to state, while maintaining acyclicity, as in level-sensitive latches. Then, two dynamically typed processes can be syntactically compatible, if all pairs of possible dependencies are compatible, or semantically compatible, if in each state the combined dependencies remain acyclic. A dependent-type resolves the dependencies gradually through a game among the components, as in gated clocks. We show that dependent-typed modules are equivalent to constructive circuits, i.e., circuits that have a constructive semantics, which are in turn equivalent to all delay-insensitive circuits.

For a given plant process and control objective, there may be a controller of a static type, or only a controller of a syntactically compatible dynamic type, or only a controller of a semantically compatible dynamic type, or only a controller of dependent-type. We show this to be a strict hierarchy of possibilities, and we present algorithms and determine the complexity of the corresponding control problems. Furthermore, we consider versions of the control problem in which the type of the controller (static, dynamic or dependent) is given. We show that the solution of these fixed-type control problems requires the evaluation of partially ordered (Henkin) quantifiers on boolean formulas, and is therefore harder (nondeterministic exponential time) than more traditional control questions. We also show that, contrary to folk wisdoms, the notion of "the most general controller" for safety properties does not exist in general for most synchronous systems.

1.2.4 Synthesis of uninitialized systems

In the synthesis problem, the requirement on the input-output behavior of the synthesized system, called the stream requirement, is specified formally by modal logic for-

mulas or ω-automata. The *Realizability Problem* (RP) asks, given a stream requirement R, to find a state machine that satisfies R. The problem was first stated by Church in [Chu62], for stream requirements specified in the *sequential calculus*. Several solutions for it have been studied [BL69, Rab72], and in [PR89a], Pnueli and Rosner solved it for specifications given in linear temporal logic and suggested its applicability in open system synthesis and control.

In practice, however, sequential hardware is often designed to operate without prior initialization. They can be modeled by *uninitialized state machines*. Such machines require no reset circuitry and therefore have an advantage of smaller area. A well-known example of uninitialized state machines is the IEEE 1149.1 standard for boundary-scan test [Com90], whose specification consists of, among others, a state machine that needs not start from a known state. Uninitialized state machines are also necessary for "safe replaceability" of sequential circuits [SP94], where a state machine is replaced by another in such a way that the surrounding environment is not able to detect the changes. The replacing state machine is an uninitialized state machine because it may power-up in an arbitrary state. Uninitialized state machines have been studied by some researchers before [SP94, QBSP96]. In Chapter 6 we study the synthesis problem of uninitialized state machines.

We define the *Uninitialized Realizability Problem* (URP), which asks given a stream requirement R on the input-output behavior, to find an uninitialized state machine M that satisfies R no matter what the initial state of M is. We study the URP for specifications that are specified by LTL formulas or Büchi automata. We consider deterministic, nondeterministic, universal, and alternating automata. In particular, we provide an algorithm for solving the URP for each of the above formalisms, and prove corresponding lower bounds. The proofs turns out to be non-trivial, and require complicated generic reductions.

Chapter 2

Preliminaries

> Mathematics is a game played according to certain simple rules with meaningless marks on paper.
>
> — David Hilbert (1862–1943)

Let X be a set of variables. In this dissertation all variables range over the set $\mathbb{B}$ of booleans. We denote by $PStates(X)$ the set of partial functions from X to $\mathbb{B}$, and by $States(X)$ the set of total functions. Given $v \in PStates(X)$, we write $Var(v) \subseteq X$ for the set of variables on which v is defined. For $Y \subseteq X$, we write $v[Y]$ for the restriction of v to the variables in Y. We indicate with $\mathcal{P}(X)$ the set of predicates over X. For a boolean formula φ over X, we write $\varphi[v] = \varphi[v(x_1)/x_1, \ldots, v(x_n)/x_n]$ for the formula obtained by replacing each variable $x_i \in Var(v)$ in φ with the truth value $v(x_i)$. If $\varphi[v]$ contains no free variables, then we let $\varphi[\![v]\!]$ denote the truth value of $\varphi[v]$. We sometimes write $v \models \varphi$ iff $\varphi[\![v]\!]$. We write $X' = \{x' \mid x \in X\}$ for the set of corresponding primed variables, and for $v \in PStates(X)$, we write v' for the partial function in $PStates(X')$ such that $v'(x') = v(x)$ for all $x \in Var(v)$, and $v'(x')$ is undefined otherwise. Given a set A and an element x, we often write $A \setminus x$ for $A \setminus \{x\}$, when this generates no confusion.

2.1 Modules and composition

The basic entity for modeling components of concurrent systems are modules. A *module* $P = (O_P, I_P, Init_P, \tau_P)$ consists of the following three components:

- A finite set O_P of *output variables*. These variables are updated by the module.

- A finite set I_P of *input variables*. These variables are updated by the environment. The sets O_P and I_P must be disjoint. We write $X_P = O_P \cup I_P$ for the set of all module variables. The *states* of P are $S_P = States(X_P)$, and the *partial* (*next*) *states* of P are $R_P = PStates(X'_P)$. Unprimed variables represent current-state values; primed variables, next-state values. A pair $\langle s, t' \rangle \in S_P \times R_P$ is called an *extended state*. For any predicate $\varphi \in \mathcal{P}(X_P)$, the state $s \in S_P$ is a φ-state if $\varphi[\![s]\!]$.

- An *initial predicate* $Init_P \in \mathcal{P}(X_P)$, defining the set of initial states of P.

- A boolean formula $\tau_P \in \mathcal{P}(X_P \cup X'_P)$, called *transition predicate*, over the set $X_P \cup X'_P$ of variables; it relates the current-state and next-state values of the module variables. The state $t \in S_P$ is a *macro-step successor*, or simply successor, of the state $s \in S_P$ if $\tau_P[\![s \cup t']\!]$. For a variable $x \in X_P$, the extended state $\langle s, u' \rangle \in S_P \times R_P$ is a (*micro-step*) (x, τ_P)-*successor* of the extended state $\langle s, t' \rangle$ if $x' \notin Var(t')$ and there exists $b \in \mathbb{B}$ such that $u' = t' \cup \{(x', b)\}$ and $\tau_P[s \cup u']$ is satisfiable.

A *Moore module* P is a module whose transition predicate is independent of the next-state values of the inputs, that is, τ_P is over $O_P \cup I_P \cup O'_P$. Two modules P and Q are *composable* if their output variables O_P and O_Q are disjoint. Given two composable modules P and Q, the *synchronous* (*lock-step*) *composition* $P \| Q$ is the module with the components $O_{P\|Q} = O_P \cup O_Q$, $I_{P\|Q} = (I_P \cup I_Q) \setminus O_{P\|Q}$, $Init_{P\|Q} = (Init_P \wedge Init_Q)$, and $\tau_{P\|Q} = (\tau_P \wedge \tau_Q)$.

In this dissertation, we consider modules that are non-blocking. A module P is *non-blocking* if it has at least one initial state, that is, the initial predicate $Init_P$ is satisfiable; and if every state has a successor, that is, for each state s there is a state t such that $\tau_P s \cup t'$.

We assume that all predicates are represented in such a way that boolean operations and existential quantification of variables are efficiently computable. Likewise, we assume that satisfiability of all predicates can be checked efficiently. *Binary decision diagrams* (BDDs) provide a suitable representation [Bry86].

2.2 Verification

2.2.1 Specification: Linear-time temporal logic

We consider specifications expressed by linear-time temporal logic (LTL) [Pnu77] formulas over a set *Prop* of atomic propositions. The set of LTL formulas contains the atomic

propositions *Prop*, and is the smallest set closed under applications of Boolean operators, the unary temporal operator $\bigcirc$, and the binary temporal operator $\mathcal{U}$. LTL formulas are interpreted over infinite computations. A computation is a function $\pi : \mathcal{N} \to 2^{Prop}$, which assigns a subset of *Prop* to each time instant $i \in \mathcal{N}$. For a computation $\pi = w_0, w_1, \ldots$ let π_i be the suffix $w_i, w_{i+1} \ldots$. Then we have:

- $\pi \models p$ for $p \in \Sigma$ iff $p \in w_0$;
- $\pi \models \neg\varphi$ iff $\pi \not\models \varphi$;
- $\pi \models \varphi \vee \psi$ iff $\pi \models \varphi$ and $\pi \models \psi$;
- $\pi \models \bigcirc\varphi$ iff $\pi_1 \models \varphi$; and
- $\pi \models \varphi \mathcal{U} \psi$ iff there exists $k \geq 0$ such that $\pi_k \models \psi$, and for all $0 \leq i < k$, we have $\pi_i \models \varphi$.

We use the abbreviation $\Diamond\varphi$ for $\mathsf{T}\,\mathcal{U}\,\varphi$ which expresses the property that φ will eventually hold; $\Box\varphi$ for $\neg\Diamond\neg\varphi$ which expresses the property that φ always holds.

For a module P a state s of P defines a subset $L(s) = \{x \in X_P \mid x[\![s]\!]\}$ of atomic propositions. A s-trace of P is a sequence of states $\bar{s} = s_0, s_1 \ldots \in (S_P)^\omega$ such that $s = s_0$ and $\tau_P[\![s_k \cup s'_{k+1}]\!]$ for $k \geq 0$. A s-trace is *initial* iff $Init_P[\![s]\!]$. A s-trace induces a computation $L(s_0)L(s_1)\ldots \in (2^{X_P})^\omega$. Given an LTL formula φ over X_P and a s-trace $\bar{s}$ of P, we say that $\bar{s}$ satisfies φ if the computation induced by $\bar{s}$ satisfies φ. We write $P, s \models \varphi$ iff all s-traces of P satisfy φ. The LTL verification problem for a module P w.r.t. the LTL formula φ asks whether all initial traces of P satisfy φ: that is, $P, s \models \varphi$ for all s where $Init_P[\![s]\!]$.

Theorem 2.1 [AH98] *The LTL verification problem for modules is PSPACE-complete.*

2.2.2 Reachability and invariant verification

A state of a module P is *reachable* if it appears in some initial trace of P. We denote by $Reach(P)$ the predicate defining the reachable states of P: this predicate can be computed using standard state-space exploration techniques [CES86] such as Algorithm 2.1.

Algorithm 2.1
Input: Module P.

Output: Predicate $Reach(P)$ over X_P.

Initialization: Let $U_0 = Init_P$.

Repeat: For $k \geq 0$, let

$$U'_{k+1} = U'_k \vee \exists X_P \,.\, (\tau \wedge U_k) \,.$$

Until: $U_{k+1} \equiv U_k$.

Return: U_k.

Algorithm 2.1 computes $Reach(P)$ by successively computing the predicates $U_0, U_1, \ldots, U_k = U_{k+1}$ where U_i defines the set of states that are reachable from any initial state of P in at most i steps. The predicate $Reach(P)$ is therefore the fixpoint of the iterations, i.e., $U_k = U_{k+1}$, when no more additional states can be reached.

One important LTL verification problem is the invariant verification problem which asks, given a module P and a predicate $\varphi \in \mathcal{P}(X_P)$, whether P satisfies the LTL formula $\Box\varphi$. It can be solved with the reachability predicate by checking if the implication $Reach(P) \rightarrow \varphi$ is valid.

Theorem 2.2 [AH98] *The invariant verification problem for modules is PSPACE-complete.*

2.2.3 Single-step vs. multi-step verification

Consider a module P and an LTL formula φ over X_P. We denote by $R(P, \varphi)$ the predicate defining the states of P that satisfy φ. That is, for any state $s \in S_P$, we have $P, s \models \varphi$ iff $R_P(\varphi)[\![s]\!]$. If $\varphi = \bigcirc\psi$, where $\psi \in \mathcal{P}(X_P)$, then $R(P, \varphi)$ can be computed by the formula

$$Pre_P(\psi) = \exists(X'_P) \,.\, (\tau \wedge \psi').$$

The predicate $Pre_P(\psi)$ is called the *predecessor* predicate, which defines the set of states that can reach a ψ-state in exactly one step. We call this the *one-step verification problem*. All other LTL verification problems can be solved by iterating, in an appropriate manner, the solutions to the the one-step verification problems [EJ91]. We call them the *multi-step* verification problems. For example, if $\varphi = \Box w$, where $w \in \mathcal{P}(X_P)$, then Algorithm 2.2 computes the predicate $R(P, \varphi)$:

Algorithm 2.2

Input: Module P and predicate w.

Output: Predicate $R(P, \Box\psi)$ over X_P.

Initialization: Let $U_0 = \psi$.

Repeat: For $k \geq 0$, let

$$U_{k+1} = U_k \wedge Pre_P(U_k).$$

Until: $U_{k+1} \equiv U_k$.

Return: U_k.

Algorithm 2.2 computes successively computes the sequence $U_0, U_1, \ldots$ of stronger predicates: at the k-th iteration, predicate U_k defines the states that can stay in the set of ψ-states in at least k steps. $R(P, \varphi)$ is therefore the conjunction of U_k, for all $k \geq 0$. At the core of the Algorithm 2.2 is the iteration of the predecessor predicate $Pre_P(\varphi)$. All other LTL formulas can also be computed in a similar fashion.

2.3 Controllability

We consider properties of open systems which interacts with its environment. An *environment* for a module P is a non-blocking module E composable with P. We can view the interaction between a module P and its environment as a game. At each round of the game, the module P chooses the next values for output variables O_P, while the environment chooses the next values for the input variables I_P. The game then continues *ad infinitum*. A key property of open systems is the notion of controllability: given an LTL specification φ, we say that a state s of P is *controllable* with respect to φ if the environment can ensure that all traces from s satisfy φ. For Moore modules, the above definition can be formalized using the notion of *strategy*. A module strategy π for P is a mapping $\pi : S_P^+ \mapsto States(O_P)$ that maps each finite sequence $s_0, s_1, \ldots, s_k$ of module states into a state $\pi(s_0, s_1, \ldots, s_k)$ such that $\tau_P[\![s_k \cup \pi'(s_0, s_1, \ldots, s_k)]\!]$. Similarly, an environment strategy η for P is a mapping $\eta : S_P^+ \mapsto States(I_P)$ that maps each finite sequence of module states into a state specifying the next values of the input variables. Given two states s_1 and s_2 over two disjoint sets of variables X_1 and X_2, we denote by $s_1 \bowtie s_2$ the state over $X_1 \cup X_2$ that agrees with s_1 and s_2 over the common variables. With this notation, for all $s \in S_P$ and all module strategies π and environment strategies η, we define $Outcome(s, \pi, \eta) \in S_P^\omega$ to be the trace $s_0, s_1, s_2, \ldots$ defined by $s_0 = s$ and by $s_{k+1} = \pi(s_0, s_1, \ldots, s_k) \bowtie \eta(s_0, s_1, \ldots, s_k)$. Given an LTL formula φ over X_P, the LTL control problem for P with respect to φ at a state $s \in S_P$

asks if there is an environment strategy η such that, for every module strategy π, we have $\varphi[\![Outcome(s, \pi, \eta)]\!]$. If so, then we say that a state $s \in S_P$ is *controllable with respect to* φ. We let the *controllable predicate* $Ctr(P, \varphi)$ be the predicate over X_P defining the set of states of P controllable with respect to φ.

Theorem 2.3 [PR89a, AH98] *The LTL control problem for modules is 2EXPTIME-complete.*

Unlike verification, the LTL control problem is easier if the LTL specification is restricted to invariants.

Theorem 2.4 [AH98] *The invariant control problem for modules is EXPTIME-complete.*

Given a module P, a state s of P and an LTL formula φ over X_P, we call the LTL control problem for P w.r.t. φ at s the *one-step control problem* if $\varphi = \bigcirc\psi$ for some $\psi \in \mathcal{P}(X_P)$; otherwise it is called the *multi-step control problem*. We let the *controllable predecessor* $CPre_P(\psi)$ be the predicate over X_P defining the set of states of P controllable with respect to $\bigcirc\psi$. All multi-step LTL control problems can be solved by repeatedly solving the appropriate one-step control problems. For example, for the invariant specification $\varphi = \Box\psi$, where $\psi \in \mathcal{P}(X_P)$ the following standard algorithm returns the predicate $Ctr(P, \varphi)$ [TW68, Bee80, RW87]:

Algorithm 2.3
Input: Module P and predicate ψ.
Output: Predicate $Ctr(P, \Box\psi)$ over X_P.
Initialization: Let $U_0 = \psi$.
Repeat: For $k \geq 0$, let

$$U_{k+1} = U_k \wedge CPre_P(U_k).$$

Until: $U_{k+1} \equiv U_k$.
Return: U_k.

Algorithm 2.3 computes a sequence $U_0, U_1, U_2, \ldots$ of increasingly strong predicates. For $k \geq 0$, predicate U_k defines the states from which it is possible to control P to satisfy predicate p for at least $k + 1$ steps: at each iteration $k \geq 0$, Algorithm 2.3 lets U_{k+1} define the set of states from which the environment can control P by ensuring that the predicate U_k is satisfied in the successor state.

If the environment of P is a Moore module, then it must decide the next value for the input variables before it can observe the next value of the output variables. Hence, to define the controllable predecessor $CPre_P(p)$, then the environment must "play first," that is,

$$CPre_P(\psi) = \exists(O'_P) \,.\, \forall(I'_P) \,.\, (\tau_P \rightarrow \psi')$$

The controllable predicate $Ctr(P, \varphi)$ can also be computed by complementing the *uncontrollable predicate* $UCtr(P, \varphi)$ which defines the set of *uncontrollable states*, that is, those states that are not *controllable with respect to* φ. The computation of $UCtr(P, \varphi)$ involves iterating the *uncontrollable predecessor* $UPre_P(\psi)$ for a predicate $\psi \in \mathcal{P}(X_P)$. Both $UCtr(P, \varphi)$ and $UPre_P(\psi)$ are defined in Chapter 3.

For general modules, and in particular for *reactive modules* [AH99], it is necessary to modify the definition of the controllable predicate to enable the environment to observe the next value of some output variables before choosing the next value of the input ones. More details can be found in Chapter 5. Nonethess, Algorithm 2.3 remains the same.

Chapter 3

Early Error Detection

Will you greet your doom
As final: set him loaves and wine: knowing
The game is finished when he plays his ace.
And overturn the table and go into the next room?
– Philip Larkin (1922 1986)

3.1 Introduction

It has been argued repeatedly that the main benefit of formal methods is *falsification*, not verification: that formal analysis can only demonstrate the *presence* of errors, not their absence. The fundamental reason for this is, of course, that mathematics can be applied, inherently, only to an abstract formal model of a computing system, not to the actual artifact. Furthermore, even when a formal model is verified, the successful verification attempt is typically preceded by many iterations of unsuccessful verification attempts followed by model revisions. Therefore, in practice, every formal method and tool is much more often applied in situations where the outcome is failure (a counterexample), rather than success (a correctness proof).

Yet most optimizations in formal methods and tools are tuned towards success. For example, consider the use of BDDs and similar data structures in model checking. Because of their canonicity, BDDs are often most effective in applications that involve equivalence checking between complex boolean functions. Successful model checking is such an application: when the set of reachable states is computed by iterating image computations, successful termination is detected by an equivalence check (between the newly explored and

the previously explored states). By contrast, when model checking fails, a counterexample is detected before the image iteration terminates, and other data structures, perhaps non-canonical ones, may be more efficient [BCCZ99]. To point out a second example, much ink has been spent discussing whether "forward" or "backward" state exploration is preferable (see, e.g., [HKQ98]). If we expect to find a counterexample, then the answer seems clear but rarely practiced: the simultaneous, dove-tailed iteration of images and pre-images is likely to find the counterexample by looking at fewer states than either unidirectional method. Third, in compositional methods, the emphasis is almost invariably on how to decompose correctness proofs (see, e.g., [HQR98]), not on how to find counterexamples by looking at individual system components instead of their product. In this work, we address this third issue.

At first glance, it seems that only the least interesting of errors can be caught by looking at a single component, as the more interesting errors typically involve the interaction between multiple components. However, by precomputing information about individual components, we can detect errors that involve multiple components earlier and more efficiently than would otherwise be possible.

To explain several fine points about our method, we need to be more formal. Recall the definition of controllability of a module P in a game between P and its environment: the moves of P consist in choosing new values for the variables output by P; the moves of the environment of P consist in choosing new values for the input variables of P. A state s of P is *controllable with respect to the invariant* $\Box\varphi$ if the environment has a strategy that ensures that φ always holds. Hence, if a state s is not controllable, we know that P from s can reach a $\neg\varphi$-state, regardless of how the environment behaves. The set C_P of controllable states of P can be computed iteratively, using the standard algorithm for solving safety games, which differs from backward reachability only in the definition of the pre-image operator. Symmetrically, we can compute the set C_Q of controllable states of Q w.r.t. $\Box\varphi$. Then, instead of checking that $P \| Q$ stays within the invariant $\Box\varphi$, we check whether $P \| Q$ stays within the stronger invariant $\Box(C_P \wedge C_Q)$. As soon as $P \| Q$ reaches a state s that violates a controllability predicate, say, C_P, by retracing the computation of C_P, taking into account also Q, we can construct a path of $P \| Q$ from s to a state t that violates the specification φ. Together with a path from an initial state to s, this provides a counterexample to $\Box\varphi$. While the error occurs only at t, we detect it already at s, as soon as it cannot be prevented. The method can be extended to arbitrary LTL requirements.

The notion of controllability defined above is classical, but it is often not strong enough to enable the early detection of errors. To understand why, consider an invariant that relates a variable x in module P with a variable y in module Q, for example by requiring that $x = y$, and assume that y is an input variable to P. Consider a state s, in which module P is about to change the value of x without synchronizing this change with Q. Intuitively, it seems obvious that such a change can break the invariant, and that the state should not be considered controllable (how can Q possibly know that this is going to happen, and change the value of y correspondingly?). However, according to the classical definition of controllability, the state s is controllable: in fact, the environment has a move (changing the value of y correspondingly) to control P. This example indicates that in order to obtain stronger (and more effective) notions of controllability, we need to compute the set of controllable states by taking into account the real capabilities of the other modules composing the system. We introduce three such stronger notions of controllability: constrained, lazy, and bounded controllability. Our experimental results demonstrate that there is a distinct advantage in using these stronger notions of controllability.

Lazy controllability can be applied to systems in which all the modules are *lazy*, i.e., if the modules always have the option of leaving unchanged the values of their output variables [AH99]. Thus, laziness models the assumption of speed independence, and is used heavily in the modeling of asynchronous systems. If the environment is lazy, then there is no way of preventing the environment from always choosing its "stutter" move. Hence, we can strengthen the definition of controllability by requiring that the stutter strategy of the environment, rather than an arbitrary strategy, must be able to control. In the above example, the state s of module P is clearly not *lazily controllable*, since a change of x cannot be controlled by leaving y unchanged. *Constrained controllability* is a notion of controllability that can be used also when the system is not lazy. Constrained controllability takes into account, in computing the sets of controllable states, which moves are possible for the environment. To compute the set of *constrainedly controllable states* of a module P, we construct a transition relation that constrains the moves of the environment. This is done by automatically abstracting away from the transition relations of the other modules the variables that are not shared by P. We then define the controllable states by considering a game between P and a so constrained environment. Finally, *bounded controllability* is a notion that can again be applied to any system, and it generalizes both lazy and constrained controllability. It considers environments that have both a set of *unavoidable moves* (such as

the lazy move for lazy systems), and *possible moves* (by considering constraints to the moves, similarly to constrained controllability). We also introduce a technique called *iterative strengthening*, which can be used to strengthen any of these notions of controllability. In essence, it is based on the idea that a module, in order to control another module, cannot use a move that would cause it to leave its own set of controllable states.

We demonstrate the efficiency of the methods with two examples, a distributed database protocol and a wireless communication protocol. In the first example, there are two sites that can sell and buy back seats on the same airplane [BGM92]. The protocol aims at ensuring that no more seats are sold than the total available, while enabling the two sites to exchange unsold seats, in case one site wishes to sell more seats than initially allotted. The second example is from the *Two-Chip Intercom* (TCI) project of the Berkeley Wireless Research Center [Cen, SdSJB+00, dSJSB+00]. The TCI network is a wireless local network which allows approximately 40 remotes, one for each user, to transmit voice with point-to-point and broadcast communication. The operation of the network is coordinated by a base station, which assigns channels to the users through a TDMA scheme. In both examples, we first found errors that occurred in our initial formulation of the models, and then seeded bugs at random. Our methods succeeded in reducing the number of global image computation steps required for finding the errors, often reducing the maximum number of BDD nodes used in the verification process. The methods are particularly effective when the BDDs representing the controllable states are small in comparison to the BDD representing the set of reachable states.

It is worth noting that the techniques developed in this chapter can also be used in an informal verification environment: after computing the uncontrollability states for each of the components, one can *simulate* the design and check if any of these uncontrollable states can be reached. This is similar to the techniques *retrograde analysis* [YSAA97], or *target enlargement* [YD98] in simulation. The main idea of retrograde analysis and target enlargement is that the set of states that violate the invariants are "enlarged" with their preimages, and hence the chances of hitting this enlarged set is increased. Our techniques not only add modularity in the computation of target enlargemen, they also allow one to detect the violation of *liveness* properties through simulation.

The algorithmic control of reactive systems has been studied extensively before (see, e.g., [RW89, EJ91, Tho95]). However, the use of controllability in automatic verification is relatively new (see, e.g., [KV96, AHK97]). The work closest to ours is [ASSSV94].

where transition systems for components are minimized by taking into account if a state satisfies or violates a given CTL property under all environments. In [Dil89], autofailure captures the concept that no environment can prevent failure and is used to compare the equivalence of asynchronous circuits.

3.2 Early Detection of Invariant Violation

3.2.1 Forward and backward state exploration

Given a module R and a predicate φ over X_R, the problem of invariant verification consists in checking whether $R \models \Box\varphi$. We can solve this problem using classic forward or backward state exploration. Forward exploration starts with the set of initial states of R, and iterates a post-image computation, terminating when a state satisfying $\neg\varphi$ has been reached, or when the set of reachable states of R has been computed. In the first case we conclude $R \not\models \Box\varphi$; in the second, $R \models \Box\varphi$. Hence, the algorithm given in Section 2.2.2 is an example of forward exploration. Backward exploration starts with the set $\neg\varphi$ of states violating the invariant, and iterates a pre-image computation, terminating when a state satisfying $Init_R$ has been reached, or when the set of all states that can reach $\neg\varphi$ has been computed. Again, in the first case we conclude $R \not\models \Box\varphi$ and in the second $R \models \Box\varphi$. Hence, the algorithm given in Section 2.2.3 can be seen as an example of backward exploration. If the answer to the invariant verification question is negative, these algorithms can also construct a counterexample $s_0, \ldots, s_m$ of minimal length leading from $s_0 \models Init_R$ to $s_m \models \neg\varphi$, and such that for $0 \leq i < m$ we have $\tau_R[\![s_i \cup s'_{i+1}]\!]$. If our aim is to find counterexamples quickly, an algorithm that alternates forward and backward reachability is likely to explore fewer states than the two unidirectional algorithms. The algorithm alternates post-image computations starting from $Init_R$ with pre-image computations starting from $\neg\varphi$, terminating as soon as the post and pre-images intersect, or as soon as a fixpoint is reached. We denote any of these three algorithms (or variations thereof) by $InvCheck(R, \varphi)$. We assume that $InvCheck(R, \varphi)$ returns answer YES or NO, depending on whether $R \models \Box\varphi$ or $R \not\models \Box\varphi$, along with a counterexample in the latter case.

3.2.2 Controllability and early error detection

Given $n > 1$ modules $P_1, P_2, \ldots, P_n$ and a predicate $\varphi \in \mathcal{P}(\bigcup_{i=1}^{n} X_{P_i})$, the modular version of the invariant verification problem consists in checking whether $P_1 \parallel \cdots \parallel P_n \models \Box\varphi$. We can use the notion of controllability to try to detect a violation of the invariant φ in fewer iterations of post or pre-image computation than the forward and backward exploration algorithms described above. The idea is to pre-compute the states of each module $P_1, \ldots, P_n$ that are controllable w.r.t. $\Box\varphi$. We can then detect a violation of the invariant as soon as we reach a state s that is not controllable for some of the modules, rather than waiting until we reach a state actually satisfying $\neg\varphi$. In fact, we know that from s there is a path leading to $\neg\varphi$ in the global system: for this reason, if a state is not controllable for some of the modules, we say that the state is *doomed*.

To implement this idea, let $R = P_1 \parallel \cdots \parallel P_n$, and for $1 \leq i \leq n$, let $abs_i(\varphi) = \exists(X_R \setminus X_{P_i}) \,.\, \varphi$ be an approximation of φ that involves only the variables of P_i: note that $\varphi \rightarrow abs_i(\varphi)$. For each $1 \leq i \leq n$, we can compute the set $Ctr(P_i, \Box abs_i(\varphi))$ of controllable states of P_i w.r.t. $\Box abs_i(\varphi)$ using a classical algorithm for safety games. For a module P, the algorithm uses the *uncontrollable predecessor operator* $UPre_P : \mathcal{P}(X_P) \mapsto \mathcal{P}(X_P)$, defined by

$$UPre_P(X) = \forall I'_P \,.\, \exists O'_P \,.\, (\tau_P \wedge X') \,.$$

The predicate $UPre_P(X)$ defines the set of states from which, regardless of the move of the environment, the module P can resolve its internal nondeterminism to make X true. Note that a quantifier switch is required to to compute the uncontrollable predecessors, as opposed to the computations of pre-images and post-images, where only existential quantification is required. For a module P and an invariant $\Box\varphi$, we can compute the set $Ctr(P, \Box\varphi)$ of controllable states of P with respect to $\Box\varphi$, by *negating* the set $UCtr(P, \Box\varphi)$ of *uncontrollable states*, which can be computed using the following algorithm:

Algorithm 3.1

Input: Module P and predicate φ over X_P.

Output: Predicate $UCtr(P, \Box\varphi)$.

Initialization: Let $U_0 = \neg\varphi$.

Repeat: For $k \geq 0$, let $U_{k+1} = \neg\varphi \vee UPre_P(U_k)$

Until: $U_{k+1} \equiv U_k$.

Return: U_k.

For $k \geq 0$ the set U_k consists of the states from which the environment cannot prevent module P from reaching $\neg\varphi$ in at most k steps. Note that for all $1 \leq i \leq n$, the computation of $Ctr(P_i, \Box abs_i(\varphi))$ is carried out on the state space of module P_i, rather than on the (larger) state space of the complete system. We can then solve the invariant checking problem $P_1 \parallel \cdots \parallel P_n \models \Box\varphi$ by executing

$$InvCheck\Big(P_1 \parallel \cdots \parallel P_n,\ \varphi \wedge \bigwedge_{i=0}^{n} Ctr(P_i, \Box abs_i(\varphi))\Big)\ . \tag{3.1}$$

It is necessary to conjoin φ to the set of controllable states in the above check, because for $1 \leq i \leq n$, predicate $abs_i(\varphi)$ (and thus, possibly, $Ctr(P_i, \Box abs_i(\varphi))$) may be weaker than φ. If check (3.1) returns answer YES, then we have immediately that $P_1 \parallel \cdots \parallel P_n \models \Box\varphi$. If the check returns answer NO, we can conclude that $P_1 \parallel \cdots \parallel P_n \not\models \Box\varphi$. In this latter case, the check (3.1) also returns a partial counterexample $s_0, s_1, \ldots, s_m$, with $s_m \not\models Ctr(P_j, \Box\varphi_j)$ for some $1 \leq j \leq n$. If $s_m \models \neg\varphi$, this counterexample is also a counterexample to $\Box\varphi$. Otherwise, to obtain a counterexample $s_0, \ldots, s_m, s_{m+1}, \ldots, s_{m+r}$ with $s_{m+r} \not\models \varphi$, we proceed as follows. Let $U_0, U_1, \ldots, U_k$ be the predicates computed by Algorithm 3.1 during the computation of $Ctr(P_j, \Box\varphi_j)$: note that $s_m \models U_k$. For $l > 0$, given s_{m+l-1}, we pick s_{m+l} such that $s_{m+l} \models U_{k-l}$ and $(s_{m+l-1} \cup s'_{m+l}) \models \bigwedge_{i=1}^{n} \tau_{P_i}$. The process terminates as soon as we reach an l such that $s_{m+l} \models \neg\varphi$: since the implication $U_0 \rightarrow \neg\varphi$ holds, this will occur in at most k steps. In the actual implementation, the state s_{m+l} is obtained by a game played between module P_j and the team comprising modules P_i, for $i \neq j$. During round $0 < l \leq k$, the state s_{m+l} is obtained from s_{m+l-1} in two steps: First, an evaluation $t \in States(\bigcup_{i=1}^{n} X_{P_i} \setminus O_{P_j})$ is chosen such that $(s_{m+l-1} \cup t') \models \tau_{P_i}$ for every $1 \leq i \leq n$, $i \neq j$. Second, an evaluation $u \in States(O_{P_j}$ is chosen in such a way that $(s_{m+l-1} \cup u') \models \tau_{P_j}$ and $t \bowtie u \models U_{k-l}$. Then we have $s_{m+l} = t \bowtie u$.

3.3 Lazy and Constrained Controllability

In the previous section, we have used the notion of controllability to compute sets of *doomed states*, from which we know that there is a path violating the invariant. In order to detect errors early, we should compute the largest possible sets of doomed states. To this end, we introduce two notions of controllability that can be stronger than the classical definition of the previous section. The first notion, *lazy controllability*, can be applied to systems that are composed only of *lazy modules*, i.e. of modules that need not react to

their inputs. Several communication protocols can be modeled as the composition of lazy modules. The second notion, *constrained controllability*, can be applied to any system.

3.3.1 Lazy controllability

A module is *lazy* if it always has the option of leaving its output variables unchanged. Formally, a module P is *lazy* if we have $(s \cup s') \models \tau_P$ for every state s over X_P. If all the modules composing the system are lazy, then we can re-examine the notion of controllability described in Section 3.2 to take into account this fact. Precisely, we defined a state to be controllable w.r.t. an LTL property φ if there is a strategy for the environment to ensure that the resulting trace satisfies φ, regardless of the strategy used by the system. But if the environment is lazy, we must always account for the possibility that the environment plays according to its *lazy strategy*, in which the values of the input variables of the module never change. Hence, if all modules are lazy, there is a second condition that has to be satisfied for a state to be controllable: for every strategy of the module, the lazy environment strategy should lead to a trace that satisfies φ. It is easy to see, however, that this second condition for controllability subsumes the first. We can summarize these considerations with the following definition. For $1 \leq i \leq n$, denote by η^ℓ the lazy environment strategy of module P_i, which leaves the values of the input variables of P_i always unchanged. We say that a state $s \in S_{P_i}$ is *lazily controllable with respect to a LTL formula* ψ iff, for every module strategy π, we have $Outcome(s, \pi, \eta^\ell) \models \varphi$. We let $LCtr(P, \varphi)$ be the predicate over X_P defining the set of states of P that are lazily controllable with respect to φ.

We can compute for the invariant $\Box\varphi$ the predicate $LCtr(P, \Box\varphi)$ by replacing the operator $UPre$ in Algorithm 3.1 with the operator $LUPre : \mathcal{P}(X_P) \rightarrow \mathcal{P}(X_P)$, the *lazily uncontrollable predecessor operator*, defined by:

$$LUPre_P(X) = \exists O'_P \,.\, (\tau_P \wedge X')[I_P/I'_P] \,.$$

where $(\tau_P \wedge X')[I_P/I'_P]$ is obtained from $\tau_P \wedge X'$ by replacing each variable $x' \in I'_P$ with $x \in I_P$. Note that $LUPre_P X$ computes a superset of $UPre_P X$, and therefore the set $LCtr(P, \Box\varphi)$ of lazily controllable states is always a subset of the controllable states $Ctr(P, \Box\varphi)$.

Given $n > 1$ lazy modules $P_1, P_2, \ldots, P_n$ and a predicate $\varphi \in \mathcal{P}(\bigcup_{i=1}^{n} X_{P_i})$, let $R = P_1 \| \cdots \| P_n$, and for all $1 \leq i \leq n$. We can check whether $P_1 \| \cdots \| P_n \models \Box\varphi$ by

executing $InvCheck(R, \varphi \wedge \bigwedge_{i=1}^{n} LCtr(P_i, \Box abs_i(\varphi)))$. If this check returns answer NO, we can construct a counterexample to $\Box\varphi$ as in Section 3.2.

3.3.2 Constrained controllability

Consider again $n > 1$ modules $P_1, P_2, \ldots, P_n$, together with a predicate $\varphi \in \mathcal{P}(\bigcup_{i=1}^{n} X_{P_i})$. In Section 3.2, we defined a state to be controllable if it can be controlled by an unconstrained environment, which can update the input variables of the module in an arbitrary way. However, in the system under consideration, the environment of a module P_i is $Q_i = P_1 \| \cdots \| P_{i-1} \| P_{i+1} \| \cdots \| P_n$, for $1 \leq i \leq n$. This environment cannot update the input variables of P_i in an arbitrary way, but is constrained in doing so by the transition predicates of modules P_j, for $1 \leq j \leq n$, $j \neq i$. If we compute the controllability predicate with respect to the most general environment instead of Q_i, we are giving to the environment in charge of controlling P_i more freedom than it really has. To model this restriction, we can consider games in which the environment of P_i is constrained by a transition predicate over $X_{P_i} \cup I'_{P_i}$ that over-approximates the transition predicate of Q_i. We rely on an over-approximation to avoid mentioning all the variables in $\bigcup_{j=1}^{n} X_{P_j}$, since this would enlarge the state space on which the controllability predicate is computed.

These considerations motivate the following definitions. Consider a module P together with a transition predicate H over $X_P \cup I'_P$. An *H-constrained strategy* for the environment of P is a strategy $\eta : S_P^+ \mapsto States(I_P)$ such that, for all $s_0, s_1, \ldots, s_k \in S_P^+$, we have $(s_k \cup \eta'(s_0, s_1, \ldots, s_k)) \models H$. Given an LTL formula φ over X_P, we say that a state $s \in SP$ is *H-controllable* if there is an H-constrained environment strategy η such that, for every module strategy π, we have $Outcome(s, \pi, \eta) \models \varphi$. We let $CCtr(P, \langle\langle H \rangle\rangle \varphi)$ be the predicate over X_P defining the set of H-controllable states of P w.r.t. φ.[1] For invariant properties, the predicate $CCtr(P, \langle\langle H \rangle\rangle \Box\varphi)$ can be computed by replacing in Algorithm 3.1 the operator $UPre$ with the operator $CUPre_P[H] : \mathcal{P}(X_P) \mapsto \mathcal{P}(X_P)$, defined by:

$$CUPre_P[H](X) = \forall I'_P \,.\, (H \rightarrow \exists O'_P \,.\, (\tau_P \wedge X')) \,.$$

When $H = true$, $CUPre_P[H](X) = UPre_P(X)$: for all other stronger predicates H, the H-uncontrollable predecessor operator $CUPre_P[H](X)$ will be a superset of $UPre_P(X)$, and

[1] If E_H is a module composable with P having transition relation H, the predicate $CCtr(P, \langle\langle H \rangle\rangle \varphi)$ defines exactly the same set of states as the ATL formula $\langle\langle E \rangle\rangle \Box\varphi$ interpreted over $P \| E_H$ [AHK97].

therefore the set $CCtr(P, \langle\langle H \rangle\rangle \varphi)$ of H-controllable states will be a subset of the controllable states $Ctr(P, \Box\varphi)$.

Given a system $R = P_1 \| P_2 \| \ldots \| P_n$ and a predicate $\varphi \in \mathcal{P}(X_R)$, for $1 \leq i \leq n$ we let

$$H_i = \bigwedge_{j \in \{1,\ldots,n\} \setminus \{i\}} \exists Y_j \cdot \exists Y_j' \cdot \tau_{P_j}$$

where $Y_j = X_{P_j} \setminus X_{P_i}$. Then we can check whether $R \models \Box\varphi$ by executing $InvCheck(R, \varphi \wedge \bigwedge_{i=1}^{n} CCtr(P_i, \langle\langle H_i \rangle\rangle \Box abs_i(\varphi)))$. If this check returns answer NO, we can construct a counterexample proceeding as in Section 3.2.

3.4 Experiments

We applied our methods for early error detection to two examples: a distributed database protocol and a wireless communication protocol. We implemented all algorithms on top of the model checker MOCHA [AHM+98], which relies on the BDD package and image computation engine provided by VIS [BHSV+96].

3.4.1 Demarcation protocol

The *demarcation protocol* is a distributed protocol for maintaining numerical constraints between distributed copies of a database [BGM92]. We considered an instance of the protocol that manages two sites that sell and buy back seats on the same airplane: each site is modeled by a module. In order to minimize communication, each site maintains a *demarcation* variable indicating the maximum number of seats it can sell autonomously: if the site wishes to sell more seats than this limit, it enters a negotiation phase with the other site. The invariant states that the total number of seats sold is always less than the total available.

In order to estimate the sensitivity of our methods to differences in modeling style, we wrote three models of the demarcation protocol: the models differ in minor details, such as the maximum number of seats that can be sold or bought in a single transaction, or the implementation of the communication channels. In all models, each of the two modules controls over 20 variables, and has 8–10 input variables; the diameter of the set of reachable states is between 80 and 120. We present the number of iterations required for finding errors in the three models using the various notions of controllability in Table 3.1. Some of

Error	L	C	R	G
e1	19	19	19	24
e2	31	31	31	36
e3	19	19	19	24
e4	18	23	24	24

(a) Model 1.

Error	L	C	R	G
e1	30	30	35	35
e2	40	40	44	44
e3	28	28	33	33
e4	18	18	25	25

(b) Model 2.

Error	L	C	R	G
e1	14	18	18	18
e2	14	18	18	18
e3	14	18	18	18
e4	12	16	16	16

(c) Model 3.

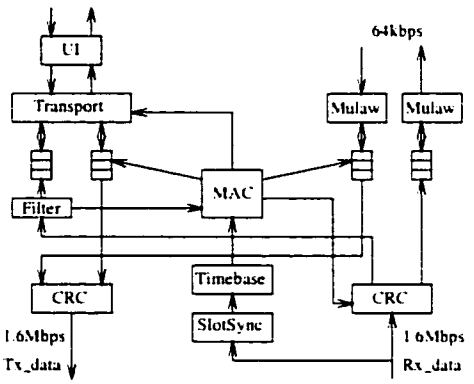
Table 3.1: Number of iterations required in global state exploration to find errors in 3 models of the demarcation protocol. The errors are e1.....e4. The columns are L (lazy controllability), C (constrained controllability), R (regular controllability), and G (traditional global state exploration).

the errors occurred in the formulation of the models, others were seeded at random.

3.4.2 Two-chip intercom

The second example is from the *Two-Chip Intercom* (TCI) project of the Berkeley Wireless Research Center [Cen, SdSJB+00, dSJSB+00]. TCI is a wireless local network which allows approximately 40 remotes to transmit voice with point-to-point and broadcast communication. The operation of the network is coordinated by a base station, which assigns channels to the remotes through a TDMA scheme. Each remote and base station will be implemented in a two-chip solution, one for the digital component and one for the analog. The TCI protocol involves four layers: the functional layer (UI), the transport layer, the medium access control (MAC) layer and the physical layer. The UI provides an interface between the user and the remote. The transport layer accepts service requests from the UI, defines the corresponding messages to be transmitted across the network, and transmits the messages in packets. The transport layer also accepts and interprets the incoming packets and sends the messages to the UI. The MAC layer implements the TDMA scheme. The protocol stack for a remote is shown in Figure 3.1(a). Each of these blocks are described by the designers in Esterel and modeled in *Polis* using *Codesign Finite State Machines* [BCG+97].

There are four main services available to a user: *ConnReq*, *AddReq*, *RemReq* and *DiscReq*. To enter the network, a remote sends a connection request, *ConnReq*, together with the id of the remote, to the base station. The base station checks that the remote is not already registered, and that there is a free time-slot for the remote. It then registers



(a) Protocol Stack.

Error	L	C	R	G
e1	24	25	27	27
e2	39	41	41	41
e3	16	21	23	23
e4	24	25	27	27
e5	57	59	59	59
e6	16	19	21	21

(b) Iterations

Figure 3.1: The TCI protocol stack and the number of iterations of global state exploration to discover the error.

the remote, and sends a connection grant back to the the remote. If a remote wishes to leave the network, it sends *DiscReq* to the base station, which unregisters the remote. If two or more remotes want to start a conference, one of them sends *AddReq* to the base station, together with the id's of the remotes with which it wants to communicate. The base station checks that the remotes are all registered, and sends to each of these remotes an acknowledgment and a time-slot assignment for the conference. When a remote wishes to leave the conference, it sends a *RemReq* request to the base station, which reclaims the time slot allocated to the remote.

We consider a TCI network involving one remote and one base station. The invariant states that if a remote believes that it is connected to the network, then the base station has this remote registered. This property involves the functional and transport layers. In our experiment, we model the network in *reactive modules* [AH99] The modules that model the functional and transport layers for both the remote and the base station are translated directly from the corresponding CFSM models: based on the protocol specification, we provide abstractions for the MAC layer and physical layer as well as the channel between the remote and the base station. Due to the semantics of CFSM, the modules are lazy, and therefore, lazy controllability applies. The final model has 83 variables. The number of iterations required to discover the various errors, some incurred during the modeling and some seeded in, are reported in Figure 3.1(b).

3.4.3 Results on BDD sizes and discussion

In order to isolate the unpredictable effect of dynamic variable ordering on the BDD sizes, we conducted, for each error, two sets of experiments. In the first set of experiments, we turned off dynamic variable ordering, but supplied good initial orders. In the second, dynamic variable ordering was turned on, and a random initial order was given. Since the maximum BDD size is often the limiting factor in formal verification, we give results based on the maximum number of BDD nodes encountered during verification process, taking into account the BDDs composing the controllability predicates, the reachability predicate, and the transition relation of the system under consideration. We only compare our results for the verification using lazy controllability and global state exploration, since these are the most significant comparisons. The results for model 3 of the demarcation protocol as well as the TCI protocol are given.

Without dynamic variable ordering. For each error, we recorded the maximum number of BDD nodes allocated by the BDD manager encountered during verification process. The results given in Table 3.2(a) and 3.2(b) are the averages of four experiment runs, each with a different initial variable order. They show that often the computation of the controllability predicates helps reduce the total amount of required memory by about 10–20%. The reason for this savings can be attributed to the fact that fewer iterations in global state exploration avoids the possible BDD blow-up in subsequent post-image computation.

With dynamic variable ordering. The analysis on BDD performance is more difficult if dynamic variable ordering is used. We present the results in Tables 3.2(c) and 3.2(d) which show the averages of nine experiment runs on the same models with dynamic variable ordering on. Dynamic variable ordering tries to minimize the total size of all the BDDs, taking into account the BDDs representing the controllability and the reachability predicates, as well as the BDDs encoding the transition relation of the system. Hence, if the BDDs for the controllability predicates are a sizeable fraction of the other BDDs, their presence slows down the reordering process, and hampers the ability of the reordering process to reduce the size of the BDD of the reachability predicate. Thus, while our methods consistently reduce the number of iterations required in global state exploration to discover the error, occasionally we do not achieve savings in terms of memory requirements.

When the controllability predicates are small compared to the reachability predicate, they do not interfere with the variable ordering algorithm. This observation suggests

the following heuristics: one can alternate the iterations in the computation of the controllability and reachability predicates in the following manner. At each iteration, the iteration in the controllability predicate is computed only when its size is smaller than a threshold fraction (say, 50%) of the reachability predicate. Otherwise, reachability iterations are carried out. Another possible heuristics to reduce the size of the BDD representation of the the controllability predicates is to allow approximations: our algorithms remain sound and complete as long as we use over-approximations of the controllability predicates.

3.5 Bounded Controllability and Iterative Strengthening

3.5.1 Bounded controllability

In lazy controllability, we know that there is a move of the environment that is always enabled (the move that leaves all input variables unchanged); therefore, that move must be able to control the module. In constrained controllability, we are given the set of possible environment moves, and we require that one of those moves is able to control the module. We can combine these two notions in the definition of *bounded controllability*. In bounded controllability, unlike in usual games, the environment may have some degree of insuppressible internal nondeterminism. For each state, we are given a (nonempty) set A of possible environment moves, as in usual games. In addition, we are also given a (possibly empty) set $B \subseteq A$ of moves that the environment can take at its discretion, even if they are not the best moves to control the module. We say that a state is *boundedly controllable* if (a) there is a move in A that can control the state, and (b) all the moves in B can control the state. The name *bounded controllability* is derived from the fact that the sets B and A are the lower and upper bounds of the internal nondeterminism of the controller.

Given a module P, we can specify the lower and upper bounds for the environment nondeterminism using two predicates $H^l, H^u \in \mathcal{P}(X_P \cup I'_P)$. We can then define the *bounded uncontrollable predecessor operator* $BUPre[H^l, H^u] : \mathcal{P}(X_P) \mapsto \mathcal{P}(X_P)$ by

$$BUPre[H^l, H^u](X) = \left[\forall I'_P \,.\, (H^u \rightarrow \exists O'_P \,.\, (\tau_P \wedge X'))\right] \vee \left[\exists I'_P \,.\, (H^l \wedge \exists O'_P \,.\, (\tau_P \wedge X'))\right].$$

Note that the quantifiers are the duals of the ones in our informal definition, since this operator computes the uncontrollable states, rather than the controllable ones. Note also that in general we cannot eliminate the first disjunct, unless we know that $\exists I'_P \,.\, H^l$ holds at all $s \in S_m p$, as was the case for lazy controllability. By substituting this predecessor

operator to *UPre* in Algorithm 3.1, given a predicate φ over X_P, we can compute the predicate $BCtr[H^l, H^u](P, \Box\varphi)$ defining the states of P that are boundedly controllable w.r.t. $\Box\varphi$. Given a system $R = P_1 \parallel \cdots \parallel P_n$ and a predicate φ over X_R, we can use bounded controllability to compute a set of doomed states as follows. For each $1 \leq i \leq n$, we let as usual $abs_i(\varphi) = \exists(X_R \setminus X_{P_i}) \cdot \varphi$, and we compute the lower and upper bounds by

$$H_i^l = \bigwedge_{j \in \{1,\ldots,n\} \setminus \{i\}} \forall Y_{j,i} \cdot \exists Y'_{j,i} \cdot \tau_{P_j} \cdot \qquad H_i^u = \bigwedge_{j \in \{1,\ldots,n\} \setminus \{i\}} \exists Y_{j,i} \cdot \exists Y'_{j,i} \cdot \tau_{P_j} \cdot$$

where for $1 \leq j \leq n$, the set $Y_{j,i} = X_{P_j} \setminus X_{P_i}$ consists of the variable of P_j not present in P_i. Then we have $R \models \Box\varphi$ iff the check $InvCheck(R, \varphi \wedge \bigwedge_{i=1}^n BCtr[H_i^l, H_i^u](P_i, \Box abs_i(\varphi)))$ returns YES. If this check fails, we can construct counterexamples by proceeding as in Section 3.2.

3.5.2 Iterative strengthening

We can further strengthen the controllability predicates by the process of *iterative strengthening*. This process is based on the following observation. In the system $R = P_1 \parallel \cdots \parallel P_n$, in order to control P_i, the environment of P_i must not only take transitions compatible with the transition relation of the modules P_j, for $j \in \{1, \ldots, n\} \setminus \{i\}$, but these modules must also stay in their own sets of controllable states. This suggests that when we compute the controllable states of P_i, we take into account the controllability predicates already computed for the other modules. For $1 \leq i \leq n$, if δ_i is the controllability predicate of module P_i, we can compute the upper bound to the environment nondeterminism by

$$H_i^u(\bar{\delta}) = \bigwedge_{j \in \{1,\ldots,n\} \setminus \{i\}} \exists Y_{j,i} \cdot \exists Y'_{j,i} \cdot (\tau_{P_j} \wedge \delta_i \wedge \delta'_i) \cdot$$

where $\bar{\delta} = \delta_1, \ldots, \delta_n$. For all $1 \leq i \leq n$, we can compute a sequence of increasingly strong controllability predicates by letting $\delta_i^0 = \mathrm{T}$ and, for $k \geq 0$, by $\delta_i^{k+1} = BCtr[H_i^l, H_i^u(\bar{\delta}^k)](P_i, \Box\varphi)$. For all $1 \leq i \leq n$ and all $k \geq 0$, predicate δ_i^{k+1} is at least as strong as δ_i^k. We can terminate the computation at any $k \geq 0$ (reaching a fixpoint is not needed), and we can verify $R \models \Box\varphi$ by executing $InvCheck(R, \varphi \wedge \bigwedge_{i=1}^n \delta_i^k)$. As k increases, so does the cost of computing these predicates. However, this increase may be offset by the faster detection of errors in the global state-exploration phase.

3.5.3 Discussion

The early error detection techniques presented in the previous sections for invariants can be straightforwardly extended to general linear temporal-logic properties. Given a system $R = P_1 \| \cdots \| P_n$ and a general LTL formula ψ over X_R, we first compute for each $1 \leq i \leq n$ the predicate δ_i, defining the controllable states of P_i with respect to ψ. This computation requires the solution of ω-regular games [EJ91, Tho95]: in the solution, we can use the various notions of controllability developed in this chapter, such as lazy, constrained, or bounded controllability. Then, we check whether $R \models \psi \wedge \Box(\bigwedge_{i=1}^{n} \delta_i)$: as before, if a state that falsifies δ_i for some $1 \leq i \leq n$ is entered, we can immediately conclude that $R \not\models \psi$. For certain classes of properties, such as reachability properties, it is convenient to perform this check in two steps, first checking that $R \models \Box(\bigwedge_{i=1}^{n} \delta_i)$ (enabling early error detection) and then checking that $R \models \psi$.

Err	Lazy				Global	
	Control		Total		Total	
e1	4.4	(0.8)	6.6	(0.1)	7.9	(0.8)
e2	4.1	(0.1)	7.2	(0.6)	9.2	(2.4)
e3	4.4	(0.1)	9.0	(0.3)	14.6	(0.3)
e4	7.3	(0.9)	8.7	(0.1)	11.1	(2.1)

(a) Demarcation Protocol (Off).

Err	Lazy				Global	
	Control		Total		Total	
e1	4.8	(0.4)	6.3	(0.4)	6.7	(0.5)
e2	5.4	(0.6)	8.6	(0.5)	9.0	(0.4)
e3	5.8	(0.4)	6.5	(1.0)	7.7	(1.6)
e4	5.4	(0.1)	10.1	(0.1)	12.0	(0.3)
e5	6.6	(0.5)	40.7	(1.8)	43.8	(0.2)
e6	5.6	(0.6)	6.8	(1.6)	7.7	(1.9)

(b) TCI (Off).

Err	Lazy				Global	
	Control		Total		Total	
e1	3.0	(0.4)	6.9	(0.7)	7.5	(0.4)
e2	3.5	(1.0)	6.7	(0.4)	8.1	(0.8)
e3	3.6	(0.5)	8.9	(1.3)	12.7	(1.9)
e4	4.4	(0.4)	9.0	(0.9)	11.8	(2.6)

(c) Demarcation Protocol (On).

Err	Lazy				Global	
	Control		Total		Total	
e1	4.2	(0.9)	7.2	(0.8)	7.3	(0.9)
e2	3.7	(0.6)	10.1	(2.4)	11.0	(2.3)
e3	4.5	(0.5)	7.4	(1.5)	6.4	(0.6)
e4	3.8	(0.3)	11.4	(2.9)	16.9	(7.4)
e5	4.0	(0.4)	60.2	(19.1)	73.7	(29.8)
e6	4.6	(0.5)	7.9	(0.9)	6.8	(0.9)

(d) TCI (On).

Table 3.2: Average maximum number of BDD nodes required for error detection during the controllability (Control) and reachability computation (Total) phases. Dynamic variable ordering was turned off in (a) and (b), and on in (c) and (d). The results are given for lazy controllability and global state exploration. All data are in thousands of BDD nodes, and the standard deviations are given in parenthesis.

Chapter 4

Automatic Proof Decomposition

4.1 Introduction

The single largest obstacle to the use of automatic methods in system verification is the state-explosion problem, which is the exponential increase in the number of system states caused by a linear increase in the number of system components or variables. Modular verification techniques attempt to overcome the state-explosion problem by exploiting the modular structure naturally present in most system designs. The basic idea is to analyze each module of the system separately, perhaps together with an environment that represents a simplified model of the rest of the system: the results obtained for the individual modules are then combined into a single result about the compound system. Unlike other tasks in the verification of finite-state systems, which have been largely automated, current modular verification techniques still rely heavily on user guidance. Aside from deciding how to break up a system into modules, the user also has to specify the environment in which to study each module, which is usually a difficult task. In this chaper, we present an approach to modular verification that is almost entirely automatic, leaving to the user only the task of specifying which variables of a module should be relevant to the other modules.

For each concrete module, we erase some variables to construct an abstract module, which has a smaller state space: the abstract module is then used to replace the concrete module in the verification process. If this approach is pursued naively, typically one of two things happens. Either one abstracts only variables that do not influence the property to be verified, which is certainly prudent but more often than not leads to insufficient savings, or one abstracts variables that do influence the desired property, in which case the

abstract module may violate the property even though the concrete module does not. We take the second route, but use additional information about the concrete module in order to construct more useful abstractions than could be achieved by simply erasing variables. In the most basic variation of our method, we use reachability information about the concrete module when erasing variables to construct an abstraction. In a more advanced variation, we also use controllability information about the concrete module with respect to the desired property. In all cases, the additional information we use can be obtained fully automatically by looking only at individual modules and the property to be verified —there is no need to involve the compound system. Our experimental results indicate that the use of reachability and controllability information can lead to dramatic improvements in verification: the resulting module abstractions are often much smaller than the concrete modules yet still preserve the desired property.

For the sake of simplicity, we describe systems as the parallel composition of one or more non-blocking *Moore* modules, for which the outputs during a transition depend only on the source state of the transition. Our approach can be adapted with only minor modifications to Mealy-type modules, such as the reactive modules of [AH99]. We consider the verification of invariance properties. An invariance property for the module P is specified by an *invariant predicate* φ over X_P. The module P *satisfies* the invariant predicate φ, written $P \models \Box\varphi$, if P never leaves the set of states defined by φ.

Consider a system $P \parallel Q$ consisting of two modules P and Q, and a desired invariant predicate φ for $P \parallel Q$. To check if $P \parallel Q \models \Box\varphi$ without constructing the global state space of $P \parallel Q$, we can remove a subset $Y_P \subseteq X_P$ of the variables of P and a subset $Y_Q \subseteq X_Q$ of the variables of Q. Formally, the abstract module $(\exists Y_P.P) = (O_P \setminus Y_P, I_P \setminus Y_P, \exists Y_P \,.\, Init_P, \exists Y_P \exists Y'_P \,.\, \tau_P)$ is constructed by existentially quantifying the removed variables in the initial and transition predicates: we say that $(\exists Y_P.P)$ is obtained by *erasing* from P the variables in Y_P. Then we can attempt to use the following standard inference rule:

$$\frac{(\exists Y_P.P) \parallel (\exists Y_Q.Q) \models \Box\varphi}{P \parallel Q \models \Box\varphi} \tag{4.1}$$

This rule is sound, because every reachable state of the concrete system $P \parallel Q$ corresponds to a reachable state of the abstract system $(\exists Y_P.P) \parallel (\exists Y_Q.Q)$. The efficiency advantage of the rule stems from the fact that the premise involves fewer variables than the conclusion, reducing the size of the state space to be explored. However, the premise may fail even

though the conclusion holds, because there may be many reachable states of the abstract system that do not correspond to reachable states of the concrete system. In fact, it is often impossible to choose suitable, reasonable large sets Y_P and Y_Q, because modular designs aggregate naturally within each module only closely interdependent variables. By erasing such dependencies between variables, the number of transitions of the abstract system grows quickly to the point of violating all but trivial invariants. Our goal is to confine this growth in abstract transitions by utilizing additional information about the component modules P and Q.

More precisely, a state s of P can be written as a pair $s = (s_a, s_w)$, where s_a is a state over the set $X_P \setminus Y_P$ of variables, and s_w is a state over the set Y_P of erased variables. The abstract module $(\exists Y_P.P)$ contains a transition from source state s_a to destination state s'_a iff the concrete module P contains a transition from (s_a, s_w) to (s'_a, s'_w) for some s_w and s'_w. As a first improvement, we can include a transition from s_a to s'_a in the abstract module only if, for some s_w and s'_w, there is a transition from (s_a, s_w) to (s'_a, s'_w) in the concrete module *and* the state (s_a, s_w) is reachable in the concrete module. This is because it is certainly not useful to include abstract transitions that have no reachable concrete counterparts. To this end, we compute a predicate $Reach(P)$ over X_P that defines the reachable states of P. The predicate $Reach(P)$ can be computed using standard state-space exploration (symbolic or enumerative). Our experiments based on symbolic methods indicate that this computation is efficient, since the module P is considered in isolation. From the predicate $Reach(P)$ we construct the module $(P \,\&\, Reach(P)) = (O_P, I_P, Init_P, \tau_P \wedge Reach(P))$, which is like P, except that it allows only transitions from reachable states. After erasing the variables in Y_P, we obtain the abstract module $(\exists Y_P.(P \,\&\, Reach(P)))$. In a similar way, we compute the reachability predicate $Reach(Q)$ for Q and construct the abstract module $(\exists Y_Q.(Q \,\&\, Reach(Q)))$. To complete the verification process, we then use the following rule:

$$\frac{(\exists Y_P.(P \,\&\, Reach(P))) \parallel (\exists Y_Q.(Q \,\&\, Reach(Q))) \models \Box\varphi}{P \parallel Q \models \Box\varphi} \tag{4.2}$$

Since the systems $P \parallel Q$ and $(P \,\&\, Reach(P)) \parallel (Q \,\&\, Reach(Q))$ have the same reachable states, rule (4.2) is sound. As we shall see, unlike the simplistic rule (4.1), the improved rule (4.2) can often be successfully applied even when the sets Y_P and Y_Q include variables that contribute to ensure the invariant φ. Yet the savings in checking the premise of rule (4.2) are just as great as those for checking the premise of the earlier rule (4.1), because the same sets of variables are erased. In other words, $(\exists Y_P.(P \,\&\, Reach(P))) \parallel (\exists Y_Q.(Q \,\&\, Reach(Q)))$

is a more accurate but no more detailed abstraction of $P \| Q$ than is $(\exists Y_P.P) \| (\exists Y_Q.Q)$. In our experiments we shall obtain dramatic results by applying rule (4.2) with the simple heuristics of erasing those variables that are not involved in the communication between P and Q. While reachability information is often used in algorithmic verification, the novelty of rule (4.2) consists in the use of such information for the modular construction of abstractions.

The effectiveness of a rule such as (4.1) or (4.2) is directly related to the number of variables that can be erased in a successful application of the rule. Rule (4.2) improves on rule (4.1) by using *reachability* information about the individual modules in the construction of the abstractions, which usually permits the erasure of more variables. It is possible to further improve on the rule (4.2) by using, in addition to reachability information, also information about the *controllability* of the individual modules with respect to the specification $\Box\varphi$. This improvement is based on the following observation. The predicate $Reach(P)$ used in (4.2) defines the reachable states of P *when* P *is in a completely general environment.* However, the module P may exhibit anomalous behaviors in a completely general environment: in particular, more states may be reachable under a completely general environment than under the specific environment provided by Q. Of course, we do not want to compute the reachable states of P when P is composed with Q: doing so would require the exploration of the state space of the global system $P \| Q$, which is exactly what our modular verification rules try to avoid. To study the module P under a suitable confining environment, while still avoiding the exploration of the global state space, we consider the module P *in the most general environment* E *that ensures the invariant* φ: that is, E is the least restrictive module such that $P \| E \models \Box\varphi$. In practice, we need not construct E explicitly, but compute only the predicate D_P that defines the set of reachable states of $P \| E$. Since E is more restrictive than the completely general environment, the predicate D_P is stronger than $Reach(P)$, and the implication $D_P \rightarrow Reach(P)$ holds. The algorithm for computing D_P follows from the standard game-theoretic algorithm for computing the set of states of the module P that are controllable with respect to the invariant φ: it can be implemented symbolically or enumeratively, with a time complexity that is linear in the

size of the state space of P [Bee80]. This leads to the following modular verification rule:

$$\frac{\begin{array}{l}(Init_P \wedge Init_Q) \rightarrow (D_P \wedge D_Q) \\ P \,\|\, (\exists Y_Q.(Q \,\&\, D_Q)) \models \Box D_P \\ Q \,\|\, (\exists Y_P.(P \,\&\, D_P)) \models \Box D_Q\end{array}}{P \,\|\, Q \models \Box\varphi} \tag{4.3}$$

where $Y_P \subseteq X_P$ and $Y_Q \subseteq X_Q$. The soundness of this rule depends on an inductive argument, and it will be proved in detail in the rest of the chapter. Essentially, the first premise ensures that the modules P and Q are initially in states satisfying $D_P \wedge D_Q$. The second premise shows that, as long as Q does not leave the set defined by D_Q, the module P will not leave the set defined by D_P; the third premise is symmetrical. As the implications $D_P \rightarrow \varphi$ and $D_Q \rightarrow \varphi$ hold, the three premises lead to the conclusion. The rule is in fact closely related to inductive forms of assume-guarantee reasoning [Sta85, AL95, AH99, McM97]. The use of the stronger predicates D_P and D_Q in the second and third premises of the rule (4.3) potentially enables the erasure of more variables compared to the earlier rule (4.2). However, in rule (4.3) this erasure can take place only on one side of the parallel composition operator or, in the case of multi-module systems, for all modules but one.

While automatic approaches to the construction of abstractions for model checking have been proposed, for example, in [Kur94, Dam96, GS97, CC99], these approaches do not exploit reachability and controllability information in a modular fashion. In particular, instead of the standard principle "first abstract, then model check the abstraction," our approach follows the more refined principle "first model check the components, then use this information to abstract, then model check the compound abstraction." In this way, our modular verification rules are doubly geared towards automatic verification methods: state-space exploration is used both to compute the reachability and controllability predicates, and to check all temporal premises (those which contain the $\models$ operator). It is worth pointing out that nontemporal premises would result in rules that are considerably less powerful. For example, suppressing variable erasures, the temporal premise $(P \,\&\, Reach(P)) \,\|\, (Q \,\&\, Reach(Q)) \models \Box\varphi$ of rule (4.2) is weaker than the two nontemporal premises $Init_P \wedge Init_Q \rightarrow \varphi$ and $\varphi \wedge Reach(P) \wedge \tau_P \wedge Reach(Q) \wedge \tau_Q \rightarrow \varphi'$ would be (here, φ' results from φ by replacing all variables with their primed versions). Similarly, the second premise of rule (4.3) is weaker than the two nontemporal premises $Init_P \wedge Init_Q \rightarrow D_Q \wedge D_P$ and $D_P \wedge \tau_P \wedge D_Q \wedge \tau_Q \rightarrow D'_P$ would be. It is easy to find examples where our temporal

premises apply, but their nontemporal counterparts do not.

4.2 Overview and Additional Definitions

4.2.1 Chapter Overview

We develop the technical details of the proposed modular verification rules in Section 4.3. The verification rules have been implemented on top of the MOCHA model checker [AHM+98], using BDD-based fixpoint algorithms for the computation of the reachability and controllability predicates. In Section 4.4 we discuss the implementation of the verification rules, and we describe the *script language* we devised in order to be able to experiment efficiently with various modular verification techniques. In Section 4.5 we present experimental results for three examples: an instance of the demarcation protocol described in Section 3.4.1 a token-ring arbiter, and a sliding-window protocol for data communication [Hol91]. We conclude with some insights gathered in the course of the experimentation with the proposed verification rules.

4.2.2 Additional Definitions

We restrict our presentation to *Moore* modules, that is, the variables in I'_P do not appear in τ_P and therefore the next value of the output variables can depend on the present value of the input variables, but not on their next value. This restriction will simplify the notation of subsequent sections. Nevertheless, all the results of this chapter can be adapted to the case of *Mealy* modules, in which the next values of the output variables can depend also on the next values of the input variables. Moreover, the modules we defined do not have *private* variables, that is, the values of all their variables can be inspected by other modules. While this restriction simplifies the presentation of the results, all the techniques we present in this chapter can be applied to modules with private variables simply by disregarding the information of which output variables are private, and which are visible from other modules. However, Proposition 4.1 and the completeness part of Proposition 4.2 need modifications if modules can have private variables.

Given a module $P = (O_P, I_P, Init_P, \tau_P)$ and an LTL formula P over the set X_P of module variables, we write $P \models \varphi$ iff $P.s \models \varphi$ for all state $s \models Init_P$. Given any predicate

H over X_P, we denote by

$$(P \,\&\, H) = \Big(O_P,\ I_P,\ Init_P \wedge H,\ \tau_P \wedge H\Big)$$

the module like P, except that only transitions from states that satisfy H are allowed. Given a module P and a set Y of variables, we let

$$(\exists Y.P) = \Big(O_P \setminus Y, I_P \setminus Y, \exists Y\ .\ Init_P, \exists Y, Y'\ .\ \tau_P\Big)$$

be the module obtained by *erasing* the variables Y in P. Note that the module $(P \,\&\, H)$ can be blocking even if module P is non-blocking. On the other hand, the parallel composition of non-blocking Moore modules is non-blocking, and a module obtained from a non-blocking Moore module by erasing variables is also non-blocking.

In the rest of the chapter, we present modular techniques for verifying whether the relation $P_1 \parallel \cdots \parallel P_n \models \Box\varphi$ holds, where $P_1, P_2, \ldots, P_n$ are composable modules, for $n > 0$, and where φ is defined over the set of variables $\bigcup_{i=1}^{n} X_{P_i}$.

4.3 Modular Rules for Invariant Verification

We present three modular rules for the verification of invariants: the rules are presented in order of increasing sophistication, and of increasing ability of successfully erasing variables. The first rule is a standard rule based on the construction of abstract modules:

$$\frac{(\exists Y_1.P_1) \parallel \cdots \parallel (\exists Y_n.P_n) \models \Box\varphi}{P_1 \parallel \cdots \parallel P_n \models \Box\varphi} \tag{4.4}$$

The second rule is derived from the above rule, by using in the construction of the abstract modules also information about the reachable states of the concrete modules. The third rule constructs the abstract modules using both reachability and controllability information about the concrete modules.

4.3.1 Reachability-based abstractions

In order to improve the ability of rule (4.4) to successfully erase variables, we construct the abstract modules using reachability information about the concrete modules.

Hence, we formulate the following modular verification rule:

$$\frac{(\exists Y_1.(P_1 \,\&\, Reach(P_1))) \parallel \cdots \parallel (\exists Y_n.(P_n \,\&\, Reach(P_n))) \models \Box\varphi}{P_1 \parallel \cdots \parallel P_n \models \Box\varphi} \tag{4.5}$$

This rule is sound. The rule is also complete, since whenever the conclusion holds, the premise also does, with the choice $Y_1 = \cdots = Y_n = \emptyset$. Our experiments indicated that rule (4.5) is often surprisingly effective in enabling the successful erasure of variables, leading to dramatic savings in the space and time requirements of verification. We illustrate this with an example.

Example 4.1 This example is a simplified version of the token-ring example presented in Section 4.5. Consider a system composed of two modules P and Q that circulate a token through a 4-phase handshake protocol. The module P has output variables $O_P = \{grant_1, ack_1, x_1, y_1, c_1\}$ and input variables $I_P = \{grant_2, ack_2\}$. All variables are boolean, except for c_1 that has domain $\{0, 1, 2, 3\}$. The module Q is defined similarly, except that the subscripts 1 and 2 are exchanged. Intuitively, $grant_2$ and ack_1 form the handshake that passes a token from Q to P. Once the token arrives into P, it is stored first in x_1, then in y_1. The handshake variables $grant_1$ and ack_2 are used to pass the token back to Q. The variable c_1 is an auxiliary variable that records the number of tokens in P. The initial condition of P is $Init_P : \neg ack_1 \wedge \neg grant_1 \wedge x_1 \wedge \neg y_1 \wedge (c_1 = 0)$; the initial condition of Q is $Init_Q : \neg ack_2 \wedge \neg grant_2 \wedge \neg x_2 \wedge \neg y_2 \wedge (c_2 = 0)$, so that the token is initially in x_1. We present the transition predicate of P in guarded-commands notation, with the convention that the values of the variables not mentioned in the assignments are not modified, and that the command to be executed is chosen nondeterministically among those whose guards are true:

$$\begin{array}{lll}
[] \; grant_2 \wedge \neg ack_1 \wedge \neg x_1 & \longrightarrow & ack_1' = \mathtt{T};\ x_1' = \mathtt{T};\ c_1' = (c_1 + 1) \mod 4 \\
[] \; \neg grant_2 \wedge ack_1 & \longrightarrow & ack_1' = \mathtt{F} \\
[] \; x_1 \wedge \neg y_1 & \longrightarrow & x_1' = \mathtt{F};\ y_1' = \mathtt{T} \\
[] \; \neg grant_1 \wedge \neg ack_2 \wedge y_1 & \longrightarrow & grant_1' = \mathtt{T};\ y_1' = \mathtt{F};\ c_1' = (c_1 - 1) \mod 4 \\
[] \; grant_1 \wedge ack_2 & \longrightarrow & grant_1' = \mathtt{F} \\
[] \; \mathtt{T} & \longrightarrow &
\end{array}$$

The transition predicate of Q is identical, except that the subscripts 1 and 2 are exchanged. The invariant is $\varphi : [((c_1 + c_2) \mod 4) < 2]$, and states that there is at most one token. To

verify that $P \| Q \models \Box\varphi$, we can apply rule (4.5) with sets of erased variables $Y_P = \{x_1, y_1\}$ and $Y_Q = \{x_2, y_2\}$. Hence, we are able to erase all the variables that are not used for communication, and that do not appear in the invariant. The intuition is that, once the value of c_1 is known, the predicate

$$Reach(P) : \Big(c_1 = 0 \wedge \neg x_1 \wedge \neg y_1\Big) \vee \Big(c_1 = 1 \wedge (x_1 \not\equiv y_1)\Big) \vee \Big(c_1 = 2 \wedge x_1 \wedge x_2\Big)$$

provides sufficient information about the possible values of the erased variables x_1 and y_1 to enable an accurate computation of the successor states. In contrast, rule (4.4) does not enable the erasure of any variables. ∎

4.3.2 Controllability and reachability-based abstractions

Consider an instance $P_1 \| \cdots \| P_n \models \Box\varphi$ of the invariant verification problem, for $n \geq 1$. As mentioned in the introduction, the predicate $Reach(P_i)$ defines the reachable states of module P_i *when the module P_i is in a completely arbitrary environment*, for $1 \leq i \leq n$. However, a module may have many more reachable states when composed with a completely arbitrary environment, than when composed with the other modules of the system. To obtain more precise predicates, we consider the states of P_i that are reachable under the *most general environment under which P_i satisfies the specification* $\Box\varphi$, for $1 \leq i \leq n$. The idea is that, if the system has been properly designed, then the actual environment of P_i is a special case of this most general environment.

Recall that an *environment* for a module P is a non-blocking module E composable with P. Given a module P and a predicate φ, we denote by $Envs(P)$ the set of all environments of P, and we let $Envs_\varphi(P) = \{E \in Envs(P) \mid P \| E \models \Box\varphi\}$ the set of environments of P under which the specification $\Box\varphi$ holds. We define

$$CR(P, \varphi) = \bigvee_{E \in Envs_\varphi(P)} \exists (X_E \setminus X_P) \,.\, Reach(P \| E)$$

with the convention that $CR(P, \varphi) = \mathrm{F}$ if $Envs_\varphi(P) = \emptyset$. The predicate $CR(P, \varphi)$ defines the set of states of P that can be reached when P is composed with an environment under which $\Box\varphi$ holds. Denote by X_φ the variables occurring in φ. The following proposition gives some additional properties of the predicate $CR(P, \varphi)$.

Proposition 4.1 *Given a non-blocking module P and a predicate φ, the following assertions hold.*

1. *There is an environment* $E \in Envs_{\varphi}(P)$ *with* $X_E = X_P \cup X_{\varphi}$ *such that* $CR(P, \varphi) \equiv \exists(X_{\varphi} \setminus X_P) \,.\, Reach(P \parallel E)$.

2. *The implications* $CR(P, \varphi) \rightarrow \exists(X_{\varphi} \setminus X_P) \,.\, \varphi$ *and* $CR(P, \varphi) \rightarrow Reach(P)$ *hold.*

Regarding the second assertion, note that in the introduction we implicitly assumed $X_{\varphi} \subseteq X_{P_i}$ for $1 \leq i \leq n$ for the sake of simplicity, while here we are only assuming the weaker $X_{\varphi} \subseteq \bigcup_{i=1}^{n} X_{P_i}$. We can then formulate the verification rule:

$$\frac{\begin{array}{l} \bigwedge_{i=1}^{n} Init_{P_i} \;\rightarrow\; \bigwedge_{i=1}^{n} CR(P_i, \varphi) \\ P_i \parallel \left(\parallel_{j \in \{1,\ldots,n\} \setminus i} (\exists Y_j \,.\, (P_j \;\&\; CR(P_j, \varphi)))\right) \models \Box CR(P_i, \varphi) \qquad 1 \leq i \leq n \end{array}}{P_1 \parallel \cdots \parallel P_n \models \Box\varphi} \tag{4.6}$$

In the second premise of this rule, for $1 \leq i \leq n$, we cannot erase variables of P_i. In fact, the predicate $CR(P_i, \varphi)$ on the right hand side of $\models$ involves most of the variables in P_i, preventing their erasure. In the experiments described in Section 4.5, the systems were composed of two modules, and rule (4.5) performed better than rule (4.6), since in rule (4.5) the variables could be erased in both the composing modules. In systems composed of many modules, it is conceivable that the advantage derived from using the stronger predicates of rule (4.6) in all modules but one, thus possibly erasing more variables, outweighs the disadvantage of not being able to erase variables in one of the modules.

Proposition 4.2 *Rule (4.6) is sound. If* $P_1, \ldots, P_n$ *are non-blocking, rule (4.6) is also complete: if the conclusion holds, then the premises also hold for* $Y_1 = \cdots = Y_n = \emptyset$.

Proof. It suffices to consider the case $Y_1 = \cdots = Y_n = \emptyset$. To show that the rule is sound, we assume that its premises hold, and we prove by induction on $k \geq 0$ that, if $s_0, s_1, \ldots, s_k$ is an initial trace of $P_1 \parallel \cdots \parallel P_n$, then $s_i \models CR(P_j, \varphi)$ for all $0 \leq i \leq k$ and $1 \leq j \leq n$. The base case follows from the first premise of (4.6). For the induction step, assume that the assertion holds for k, and consider the assertion for $k+1$ for any j, with $1 \leq j \leq n$. The trace $s_0, s_1, \ldots, s_k, s_{k+1}$ is an initial trace of $P_j \parallel \left(\parallel_{i \in \{1,\ldots,n\} \setminus j} (P_j \;\&\; CR(P_j, \varphi))\right)$ Hence, we have that $s_{k+1} \models CR(P_j, \varphi)$, completing the induction step. From $X_{\varphi} \subseteq \bigcup_{i=1}^{n} X_{P_i}$ and from Proposition 4.1, part 2, we have that the implication $(\bigwedge_{i=1}^{n} CR(P_i, \varphi)) \rightarrow \varphi$ holds. This implication, together with the conclusion of the induction proof, leads to the desired

result. The completeness of the rule follows by noticing that if $P_1 \parallel \cdots \parallel P_n \models \Box\varphi$, then by definition of $CR(\cdot, \varphi)$ we have $P_1 \parallel \cdots \parallel P_n \models \Box(CR(P_1, \varphi) \wedge \cdots \wedge CR(P_n, \varphi))$. ∎

To compute the predicate $CR(P, \varphi)$ given P and φ, we proceed in two steps. First, we compute the predicate $Ctr(P, \varphi)$ defining the set of states from which P is *controllable* with respect to the safety property $\Box\varphi$. The predicate $Ctr(P, \varphi)$ can be computed using Algorithm 2.3. Then the predicate $CR(P, \varphi)$ can be computed using the following algorithm, which incorporates Algorithm 2.3 as a subroutine.

Algorithm 4.1

Input: Module P and predicate φ.

Output: Predicate $CR(P, \varphi)$ over X_P.

Initialization: Let $\mathcal{F} = X_\varphi \setminus X_P$, and $V_0 = Init_P \wedge \exists\mathcal{F} \,.\, \forall O_P \,.\, \Big(Init_P \rightarrow (Ctr(P, \varphi) \wedge \varphi)\Big)$.

Repeat: For $k \geq 0$, let

$$V'_{k+1} = V'_k \vee \exists X_P \,.\, \Big[V_k \wedge \tau_P \wedge \exists\mathcal{F}' \,.\, \forall O'_P \,.\, \Big(\tau_P \rightarrow (Ctr'(P, \varphi) \wedge \varphi)\Big)\Big] \;.$$

Until: $V_{k+1} \equiv V_k$.

Return: V_k.

For each $k \geq 0$, the predicate V_k over X_P defines the set of states of P that can be reached in k or less steps when P is composed with an environment E such that $P \parallel E \models \Box\varphi$. To understand how this predicate is computed, note that the predicate $\forall O_P \,.\, (Init_P \rightarrow (Ctr(P, \varphi) \wedge \varphi))$ defines the set of initial valuations for the variables in $I_P \cup \mathcal{F}$ that are *safe for the environment:* if one such valuation is chosen by the environment, the system will start in a controllable state that satisfies φ, regardless of the valuation for the output variables in O_P chosen by the module P. The iteration step follows a similar idea. If V_k defines the set of current states, then the formula $K_1 : \exists X_P \,.\, (V_k \wedge \tau_P)$ over O'_P defines the valuations for the output variables that can be chosen by P for the following state. The environment must choose a valuation for the variables in $I'_P \cup \mathcal{F}'$ that ensures that, regardless of the valuation for O'_P chosen by the module, the successor state satisfies $Ctr'(P, \varphi) \wedge \varphi$. If V_k defines the set of current states, the set of such valuations for $I'_P \cup \mathcal{F}'$ is defined by the formula

$$K_2 : \exists X_P \,.\, \forall O'_P \,.\, \Big((V_k \wedge \tau_P) \rightarrow (Ctr'(P, \varphi) \wedge \varphi)\Big).$$

It is then easy to see that the iteration step of Algorithm 4.1 can be written simply as $V'_{k+1} = K_1 \wedge \exists\mathcal{F}' \,.\, K_2$, so that K_1 constrains the next valuation of the output variables,

and $\exists \mathcal{F}'$. K_2 constrains the next valuation of the input variables. Algorithms 2.3 and 4.1 can be implemented enumeratively or symbolically, and they have running time linear in $|States(X_P \cup X_\varphi)|$. In the next example, we see how rule (4.6) can enable the erasure of variables that could not be erased with rule (4.5).

Example 4.2 Consider the verification problem $P_1 \parallel P_2 \models \Box\varphi$, where the invariant is $\varphi : \neg z_1 \wedge \neg z_2$. The modules have variables $O_{P_i} = \{x_i, y_i, z_i\}$ and $I_{P_i} = \{x_{2-i}, z_{2-i}\}$, for $1 \leq i \leq 2$; all the variables are boolean. Module P_1 has initial predicate $Init_{P_1} : \neg x_1 \wedge \neg y_1 \wedge \neg z_1$, and has transition predicate $\tau_{P_1} : [x_1' \equiv z_2] \wedge [(\neg x_1 \wedge \neg x_2) \rightarrow (y_1' \equiv y_1)] \wedge [\neg y_1 \rightarrow (z_1' \equiv z_1)]$. Module P_2 is defined in a symmetrical fashion. Informally, module P_1 behaves as follows. Initially, all variables are false. At each step, the new value for x_1 is the old value of z_2. If $x_1 \vee x_2$ holds, then y_1 can change value; otherwise, it retains its previous value. If y_1 is true, then z_1 can change value; otherwise, it retains its previous value. It is easy to check that $P_1 \parallel P_2 \models \Box\varphi$ holds.

Consider module P_1. The states where $z_1 = \mathtt{T}$ or $z_2 = \mathtt{T}$ are obviously not controllable. The states where $y_1 = \mathtt{T}$ are also not controllable, since from these states module P_1 can reach a state where $z_1 = \mathtt{T}$ regardless of the values of the input variables x_2 and z_2. Likewise, the states where $x_1 = \mathtt{T}$ or $x_2 = \mathtt{T}$ are not controllable, since from these states the module can reach a state where $y_1 = \mathtt{T}$ regardless of the values of the input variables. The only controllable (and reachable) state of P_1 is thus defined by the predicate $CR(P_1, \varphi) : \neg x_1 \wedge \neg y_1 \wedge \neg z_1 \wedge \neg x_2 \wedge \neg z_2$. Predicate $CR(P_2, \varphi)$ is defined in a symmetrical fashion. The reachability predicates are given simply by $Reach(P_1) : \mathtt{T}$ and $Reach(P_2) : \mathtt{T}$.

Rule (4.6) can be applied by taking $Y_1 = Y_2 = \{y_1, y_2\}$. In fact, the composite module $P_1 \parallel (\exists Y_2.(P_2 \,\&\, CR(P_2, \varphi)))$ admits only the initial traces consisting of repetitions of the state $[x_1 = \mathtt{F}, y_1 = \mathtt{F}, z_1 = \mathtt{F}, x_2 = \mathtt{F}, z_2 = \mathtt{F}]$. This shows that the first premise holds; the case for the second premise is symmetrical. On the other hand, no variable can be successfully erased using rule (4.5). In fact, if we erase variable y_2, then the right hand side exhibits the initial trace $s_0.s_1$, where $s_0 : [x_1 = \mathtt{F}, y_1 = \mathtt{F}, z_1 = \mathtt{F}, x_2 = \mathtt{F}, z_2 = \mathtt{F}]$ and $s_1 : [x_1 = \mathtt{F}, y_1 = \mathtt{F}, z_1 = \mathtt{F}, x_2 = \mathtt{F}, z_2 = \mathtt{T}]$. This trace is possible because the state $t_0 : [x_1 = \mathtt{F}, z_1 = \mathtt{F}, x_2 = \mathtt{F}, y_2 = \mathtt{T}, z_2 = \mathtt{F}]$ over X_{P_2} is reachable, and hence it satisfies $Reach(P_2)$, and agrees with s_0 on the shared variables. The trace is then a consequence of the transition from t_0 to $t_1 : [x_1 = \mathtt{F}, z_1 = \mathtt{F}, x_2 = \mathtt{F}, y_2 = \mathtt{T}, z_2 = \mathtt{T}]$ in P_2. A similar argument shows that it is not possible to erase the variable x_2. ∎

4.4 Implementation of the Verification Rules

We have implemented the algorithms described in this chapter in the verification tool MOCHA [AHM+98]. MOCHA is an interactive verification environment and it enables, among other things, the verification of invariants using both enumerative and symbolic techniques: for the latter, it relies on the BDD package and image computation engine provided by VIS [BHSV+96], which we used in our implementation.

One important technique we use in the implementation of the rules is that, instead of computing the abstract modules explicitly, we compute them *implicitly*. The idea is as follows: suppose we are computing the reachable states of $(\exists Y_P.P) \parallel (\exists Y_Q.Q)$. A straightforward algorithm would be to first compute the two abstract modules, and then compute the reachable states of their composition. This is very inefficient in terms of the usage of space. Transition relations are usually presented as a list of conjuncts rather than as a single, larger conjunct. The explicit computation of the abstract modules would imply conjoining all the transition relations and building a monolithic one: if represented as a BDD, such a monolithic conjunct would often be prohibitively large. Instead, we quantify away the erased variables of the abstract modules only when necessary, as for example in the computation of the reachable states. For instance, we use the following symbolic algorithm to compute the reachable states of the parallel composition of two abstract modules:

Algorithm 4.2

Input: Modules P and Q, and variables $Y_P \subseteq X_P \setminus O_Q$ and $Y_Q \subseteq X_Q \setminus O_P$.

Output: $Reach((\exists Y_P.P) \parallel (\exists Y_Q.Q))$.

Initialization: Let $U_0 = \exists(Y_P \cup Y_Q)\,.\,(Init_P \wedge Init_Q)$.

Repeat: For $k \geq 0$, let

$$U'_{k+1} = U'_k \vee \exists(X_P \cup X_Q \cup Y'_P \cup Y'_Q)\,.\,(U_k \wedge \tau_P \wedge \tau_Q)\,.$$

Until: $U_{k+1} \equiv U_k$.

Return: U_k.

In the body of the loop, we rely on the early quantification algorithm in VIS to keep the intermediate BDDs small. With this scheme, a monolithic transition relation is never built. In particular, our implementation represents abstract modules as pairs consisting of a concrete module and of a list of variables that have been erased from it: such pairs are called *extended modules*.

In order to experiment with the verification rules proposed in this chapter, we implemented a simple script language, called **sl**, built on top of MOCHA and based on the Tcl/Tk API. The algorithms and methodologies described in this chapter provide the theoretical basis of the commands provided by **sl**. The verification rules proposed in this chapter can be implemented as **sl** scripts, and the language **sl** provides invaluable flexibility for experimenting with alternative forms of the rules. An example of script is the following, which verifies the correctness of the *demarcation protocol* using rule (4.5) (the demarcation protocol is described in Section 4.5.1).

```
read_module  demarc.rm
sl_em        P Q Spec
sl_reach     phi    em_Spec s
sl_reach     rp     em_P s
sl_restrict  Prest  rp em_P
sl_erase     Pabs   Prest  P/xw P/xr P/req1 P/grant1 P/req2 \
                           P/grant2 P/xlupd1 P/xlupd2 P/busy
sl_reach     rq     em_Q s
sl_restrict  Qrest  rq em_Q
sl_erase     Qabs   Qrest  Q/xw Q/xr Q/req1 Q/grant1 Q/req2 \
                           Q/grant2 Q/xlupd1 Q/xlupd2 Q/busy
sl_compose   Rabs   Pabs Qabs
sl_checkinv  Rabs   phi s
```

The command **read_module** parses the file **demarc.rm**, containing the declarations of the modules **P** and **Q**, composing the protocol, and **Spec**, whose reachable states constitute the invariant. The command **sl_em P Q Spec** builds the extended modules **em_P**, **em_Q**, and **em_Spec** from **P**, **Q**, and **Spec**; of course, these extended modules have empty sets of erased variables. The command **sl_reach phi em_Spec s** computes the predicate **phi** = *Reach*(**em_Spec**). The parameter **s** of this and other commands means "silent", i.e., no diagnostic information is printed. The rest of the script checks that $\texttt{em\_P} \parallel \texttt{em\_Q} \models \Box\texttt{phi}$ using rule (4.5). First, the commands **sl_reach** and **sl_restrict** are used to compute **rp** = *Reach*(**em_P**) and **Prest** = (**em_P** & **rp**). Then, the command **sl_erase** erases a specified list of variables from **Prest**, producing the extended module **Pabs**. As discussed earlier, the command **sl_erase** performs no actual computation, but simply adds the specified variables to the list of erased variables. The extended module **Qabs** is constructed in an analogous fashion. Finally, the command **sl_compose** composes **Pabs** and **Qabs** into a single extended module **Rabs**, which is checked against the specification $\Box$**phi** by command **sl_checkinv**.

Apart from these commands, we also have implemented commands including `sl_wcontr` and `sl_contrreach`, which together compute the predicate $CR(P, \varphi)$ given a module P and a predicate φ.

4.5 Experimental Results

To demonstrate the effectiveness of the proposed approach to modular verification, we compare the time and memory requirements of global state-space exploration with those of rule (4.5) and rule (4.6). We do not compare our approach with other modular verification approaches, since these approaches involve user intervention for the construction of the environments. By manually constructing the environments or the abstractions it is possible to improve on our results.

We consider three examples: a demarcation protocol used in distributed databases, a token-ring arbiter, and a sliding-window protocol for data communication. All experiments have been run on a 233 MHz Pentium® II PC with 128MB memory running Linux. We report the memory usage by giving the maximum number of BDD nodes used in any fixpoint computation or predicate: this is essentially the maximum number of BDD nodes used at any single time during verification. We also report the total CPU time: this time does not include swap activity (swap activity was in any case very limited for all examples reported). The automatic variable reordering heuristics of MOCHA were enabled during the experiments. We remark that differences in time or memory usage of up to a factor of 2 are not significant, since they can easily be produced by a variation in the automatic choice of variable ordering.

4.5.1 Demarcation protocol

We consider an instance of the demarcation protocol described in Section 3.4.1. The protocol ensures that two databases, residing at sites 1 and 2, never sell more than the maximum available number of seats m aboard a plane. The variables x_1 and x_2 indicate the number of seats that have been sold at sites 1 and 2. Each site can both sell seats, and receive seats returned due to cancellations. In order to minimize the communication between two sites, each site $i = 1, 2$ maintains a variable xl_i indicating the maximum number of seats it can sell autonomously. If a site wishes to sell more seats than this limit allows, the site can send a request to the other site for more seats. Depending on the number of

unsold seats, the other site has the option of rejecting the request, or of granting it in part or in full.

We model each site $i = 1, 2$ by a module P_i; the specification is $\Box[(x_1 \leq xl_1) \wedge (x_2 \leq xl_2) \wedge (xl_1 + xl_2 \leq m)]$. Each of P_1 and P_2 controls 20 variables, of which 8 are used for communication with the other module or appear in the invariant, and 12 are internal. Rule (4.5) enable the erasure of 9 of these 12 variables in each of P_1 and P_2: all of these variables are in the cone of influence of the specification. Table 4.1 below compares the time and space requirements of global state space exploration with those of rules (4.5) and (4.6), for various values of m. To check the robustness of rule (4.5) against changes in the system model, we also wrote an alternative, somewhat more complex model for the demarcation protocol. For $m = 4$, the verification of the alternative model required 136156 BDD nodes and 2009 seconds with the global approach, and 18720 BDD nodes and 211 seconds with rule (4.5).

	Global		Rule (4.5)		Rule (4.6)	
m	BDD nodes	seconds	BDD nodes	seconds	BDD nodes	seconds
4	20881	97	2847	25	8695	75
6	64345	439	3338	40	20953	218
8	179364	1671	8367	81	43915	517
10	633102	8707	10475	112	65410	1878
12	space-out	--	15923	174	93295	1980
14	space-out		22205	300	145676	3913

Table 4.1: Experiment results on the demarcation protocol.

4.5.2 Token ring arbiter

The second example is a synchronous token-ring arbiter. It involves a ring of m stations, around which a single *token* is passed unidirectionally through four-phase handshake protocols. The invariant states that there is at most one token present in the stations. A straightforward invariant would involve nearly all the variables in the system, and be rather tedious to write. Hence, we introduce *observer modules* that observe the number of tokens in the system. To enable the decomposition of the ring into two modules P_1 and P_2 representing the half-rings, we introduce two such observers, one for each half. We were able to erase all the variables used for the internal communications and state of the half-rings.

even though these variables clearly belong to the cone of influence of the invariant. Each half ring controls $1 + 5m/2$ variables; of these, all but 4 could be erased. In Table 4.2 we compare the performance of global state-space exploration and of rules (4.5) and (4.6).

	Global		Rule (4.5)		Rule (4.6)	
m	BDD nodes	seconds	BDD nodes	seconds	BDD nodes	seconds
16	657	8	979	7	608	8
20	466	10	1619	9	308	12
24	1138	22	1297	26	473	20
28	1300	39	3486	24	519	29
32	1187	110	3190	143	772	143
36	1323	611	8230	242	1346	195

Table 4.2: Experiment results on the token-ring arbiter.

4.5.3 Sliding window protocol

Our last example is a classical sliding windows protocol from [Hol91], whose encoding is taken from the MOCHA distribution. The protocol assumes a sender sending messages to a receiver through a lossy channel with delays. Each message has a sequence number which is used by the receiver to reorder the received messages as well as for acknowledgment. There is a *window size* m which specifies the maximum number of outstanding acknowledgment the sender can tolerate: once there are m outstanding acknowledgment, the sender stops sending messages until it receives acknowledgment for the oldest, unacknowledged message. Let n be the sequence number for this message. Let k be the sequence number of the oldest message that has not arrived on the receiver side yet. The invariant states essentially that the windows are not over-run by the protocols, that is, $k - n \leq m$.

In both the sender and the receiver, roughly half of the variables not used for communication with the other module can be erased when applying our modular approach. The comparison between the performance of global state-space exploration and rules (4.5) and (4.6) is presented in Table 4.3.

4.5.4 Discussion

The experimental results indicate that the proposed approach leads to a considerable reduction in the time and space requirements for the verification process. In the

	Global		Rule (4.5)		Rule (4.6)	
m	BDD nodes	seconds	BDD nodes	seconds	BDD nodes	seconds
3	8992	35	776	12	2443	33
4	11831	99	1723	41	3740	42
5	36359	1911	3843	84	8503	105
6	94684	4994	7048	156	18316	500
7	95667	2630	8282	513	22289	771
8	space-out	—	26611	1582	47605	6245

Table 4.3: Experiment reults on the sliding window protocol.

examples we considered, we identified which variables could be erased in the application of rule (4.5) by a simple trial-and-error process. We can automate this process by providing, for each module P, a list $\{x_1, \ldots, x_k\} \subseteq O_P$ of variables of P that are not part of the specification, and that are not accessed by other modules. We list first the variables that are more likely to be successfully erased: those that are more "internal" to the module, and that interact with fewer other variables. We then apply rule (4.5) successively with the sets of erased variables $\{x_1, \ldots, x_k\}$, $\{x_1, \ldots, x_{k-1}\}$, $\{x_1, \ldots, x_{k-2}\}$, ..., until the rule succeeds. This process is efficient in practice. In fact, the more variables are erased, the smaller is the state space of the abstract modules: hence if too many variables are erased, the rule will fail in a fraction of the time required for a successful proof.

In the three examples considered, the stronger reachability predicates used to construct the abstract modules in rule (4.6) did not enable the erasure of any additional variable. In the demarcation protocol and in the sliding window protocol examples, the ability of rule (4.5) to erase variables on both sides of the parallel composition operator led to superior results compared with rule (4.6). In the token ring arbiter example, module P_i has many more reachable states in a completely general environment than in an environment compatible with the specification, for $i = 1, 2$. Hence, the predicates $Reach(P_i)$ are much weaker (and take more time and space to compute) than the predicates $CR(P_i, \varphi)$, for $i = 1, 2$. For this reason, rule (4.6) performs better than rule (4.5) in this example.

If the premise of rule (4.5) does not hold, we can construct automatically a trace over the variables in $\bigcup_{i=1}^{n}(X_{P_i} \setminus Y_i)$, leading to a state that does not satisfy φ. This trace is a trace over a partial set of system variables, and it does not necessarily correspond to a counterexample to the conclusion. If the first premise of rule (4.6) does not hold, then

using facts about controllability we can reconstruct automatically a counterexample trace over the complete set of system variables. On the other hand, if the second premise of rule (4.6) does not hold for some $1 \leq i \leq n$, then we obtain a trace over a partial set of system variables that leads to a state t_i where the predicate $CR(P_i, \varphi)$ does not hold. From t_i, using facts about controllability we can again construct a trace over the complete set of system variables that leads to a state where φ does not hold. When confronted with a trace over a partial set of variables, we have taken the naïve approach of selectively un-erasing some variables in the premises, until either the premises became valid, or the design error could be identified.

Chapter 5

Composition and Control

5.1 Introduction

The formulation of the control problem builds on the notion of parallel composition: given a transition system P (the "plant"), is there a transition system Q (the "controller") such that the compound system $P\|Q$ meets a given objective? Hence it is not surprising that even small variations in the definition of composition may influence the outcome of the control problem, as well as the hardness of its solution. (The latter distinguishes control from verification, whose complexity — PSPACE for invariant verification — is remarkably resilient against changes in the definition of parallel composition.) At the highest level, one can distinguish between asynchronous and synchronous forms of composition. Pure asynchronous (or interleaving) composition is disjunctive: one component proceeds at a time, so that an action of the compound system is an action of some component. Pure synchronous (or lock-step) composition is conjunctive: all components proceed simultaneously, so that an action of the compound system is a tuple of actions, one for each component. While many concurrency models exhibit mixed forms of composition (e.g., interleaving of internal actions and synchronization of communication actions [Mil89]), it is natural to start by considering the control problem for the two pure forms of composition. The study of these control problems corresponds to the study of winning conditions of games, where the two players (plant vs. controller) choose moves (actions) to prevent (resp. accomplish) the control objective.

In practice, the most important control objective is invariance: the controller strives to forever keep the plant within a safe set of states. The problem of invariance

control can be solved by a fixed-point iteration: first, we find a strategy that keeps the plant safe for a single step: then, a strategy that keeps the plant safe for two steps: etc. We henceforth refer to invariance control as the "multi-step" control problem, and to the problem of keeping the plant safe for a single step, as the "single-step" control problem. This allows us to separate concerns: the definition of parallel composition enters the solution of the single-step problem, but independently of the type of composition, the multi-step problem can always be solved by iteratively solving single-step problems. In other words, we can independently study (1) the single-step control problem, and the definition of parallel composition plays a central role in this study, *or* (2) the multi-step control problem (for invariance or even more general, ω-regular objectives), assuming to be given a solution to the single-step problem. While (2) has been researched extensively in the literature [BL69, GH82, RW87, EJ91, McN93, TW94, Tho95], it is (1) we focus on in this chapter.

We assume that the plant P is specified in a compact form, by a transition predicate on boolean variables, so that the state space of P is exponentially larger than the description of P, which is the input to the control problem. For solving the multi-step control problem, the number of single-step iterations is bound by the number of states. Therefore, if the single-step problem can be solved in exponential time, then so can the multi-step problem. Conversely, it can be shown that even if the single-step problem can be solved in constant time, the multi-step problem is still complete for EXP (deterministic exponential time). This seems to indicate that the single-step problem is of little interest, and it may explain why not much attention has been paid to the single-step problem previously. To our surprise, we found that for certain natural forms of parallel composition, the single-step control problem can *not* be solved in (deterministic) exponential time, and therefore its complexity dominates also the one of multi-step control.

An essential property of systems is to be *non-blocking*, in the sense that every state should have at least one successor state [BG88, Hal93, Kur94, Lyn96]. Non-blocking is essential for compositional techniques such as assume-guarantee reasoning [AL95, McM97, AH99]. In control, non-blocking means that the controller should never prevent the plant from moving. While the asynchronous composition of non-blocking processes is always non-blocking, synchronous composition needs to be restricted to ensure non-blocking. A second kind of restriction arises from modeling "typed" components, where the type specifies the input ports and output ports of a component, as well as permissible and impermissible dependencies between input and output signals [AH99]. In particular, hardware components

are usually typed in this way, for example, in order to avoid combinational loops or zero-delay cycles. In control, if we restrict our attention to typed controllers, then a controller may not exist even when an untyped controller would exist. These two kinds of common restrictions on synchronous composition, non-blocking and typing, are related, as typing can be used for syntactically enforcing the semantic concept of non-blocking for synchronously composed systems.

If the plant is given by a boolean transition predicate, and parallel composition is asynchronous, then single-step control amounts to evaluating the conjunction of a $\forall$ formula ("all actions of the plant are safe") and an $\exists$ formula ("some action of the controller is safe"). Hence, the complexity class of asynchronous single-step control is DP (which contains the differences of languages in NP). For synchronous systems, the various restrictions on composition give rise to different control problems. One way of ensuring non-blocking is to consider only Moore processes. A Moore process is a non-blocking process in which the next values of the output signals do not depend on the next values of the input signals. The composition of Moore processes is again Moore, and therefore non-blocking. If both the system and the controller are Moore processes, then the single-step control formula has the quantifier prefix $\exists\forall$ ("the controller can choose the new input signals, so that regardless of the new output signals, the system is safe").

A more liberal way of ensuring non-blocking is to consider typed processes, i.e., processes that explicitly specify the dependencies between the new values of input and output signals. We distinguish between "static" types, where the input-output dependencies are fixed, "dynamic" types, where the dependencies can change from state to state, and "dependent" types, where the dependencies unfold in steps as the next-state variables acquire values. Dynamic types may be composed either "syntactically" (by requiring that all possible combinations of dependency relations of the component processes are acyclic), or "semantically" (by requiring acyclicity at all states of the compound system). Static, dynamic and dependent types ensure that that the compound system is again typed, and therefore non-blocking. We consider two variants of the typed control problems: one in which we are free to choose both the controller and its type, and one in which we must find a controller of a specified type. If we can choose the type of the controller, the control problem can be solved by considering for the controller an exponential number of types of a simple form, namely, types that represent linearly ordered input-output dependencies. The single-step control problem resulting from each linear order of dependencies gives rise to a

boolean formula with a linear quantifier prefix, with any number of $\forall\exists$ alternations, which puts the problem into PSPACE. If the type of the desired controller is given, the single-step control problem becomes considerably harder. This is because a (static or dynamic) type may specify partially ordered input-output dependencies. These partially-ordered dependencies correspond to boolean formulas with partially ordered (Henkin) quantifiers [Hen61, Wal70, BG86], whose complexity class for satisfiability is NE (a weak form of non-deterministic exponential time) [GLV95].

The solution of control problems in presence of types gives rise to additional surprising phenomena. For example, with static or syntactically composed dynamic types, two states s and t may both be controllable even though there is not a single controller that controls both s and t (two different controllers are required). Hence, while types provide an efficient mechanism for ensuring the non-blocking of synchronously composed systems, they cause difficulties in control. On the other hand, these difficulties are often not artificial, but they correspond to real input/output constraints in the design of controllers.

Combinational or zero-delay loops in synchronous systems have been studied before. Wolf [Wol95] in her thesis recognized that some behaviors of a synchronous circuits cannot be represented by boolean relations: for instance, the behavior that some signals may not stabilize cannot be captured by boolean relations. She called this the "disappearing loop behavior" problem and they resolve it by working on three-value logic. Malik [Mal94] gave a procedure to to determine if a circuit with combinational loop is well-behaved — nonblocking in our terminology. Shiple et. al.[SBT96] built on Mailk's work and defined the constructive semantics for synchronous languages such as Esterel [Ber99]. Constructive semantics gives a sufficient and necessary condition for a model described in a synchronous language, such as Esterel and Lustre, to be realizable as a physical object: more specifically, an electrical circuit described by an synchronous language is constructive if and only if it stabilizes in bounded time for all gate and wire delays in the circuit. However, the definition of constructive semantics is not based on games and hence is not applicable for studying control. We show that the constructive semantics [Ber99] can be defined using dependent types, and hence our results are also applicable to synchronous systems such as Esterel which have constructive semantics.

A note on the definition of a module. We require non-blocking to be a property that holds at every state of a module. Moreover, our main focus is on the single-step control problem. Hence, the the initial states of a module do not play any role in our investigation.

To simplify presentation, we therefore modify the definition of a module $P = (O_P, I_P, \tau_P)$ to be a triple that contains the output variables O_P, input variables I_P and a transition predicate τ_P. All other definitions defined in Chapter 2 remain the same.

5.2 Types for Synchronous Composition

5.2.1 Asynchronous modules

Given two composable modules P and Q. The *asynchronous (interleaving) composition* $P|Q$ is the module with the same output and input variables as $P||Q$, but with the transition predicate $\tau_{P|Q} = ((\tau_P \wedge (O'_Q = O_Q)) \vee (\tau_Q \wedge (O'_P = O_P)))$. Recall that a module P is *non-blocking* if every state has a successor. The asynchronous composition of two composable non-blocking modules is again non-blocking. Hence, we say that any two composable non-blocking modules are *async-composable*. However, there are composable non-blocking modules whose synchronous composition is not non-blocking.

Example 5.1 Let module P be such that $O_P = \{x\}$, $I_P = \{y\}$, and $\tau_P = ((y' \wedge \neg x') \vee (\neg y' \wedge x'))$. Let module Q be such that $O_Q = \{y\}$, $I_Q = \{x\}$, and $\tau_Q = ((x' \wedge y') \vee (\neg x' \wedge \neg y'))$. Then P and Q are non-blocking and composable. However, the transition predicate of $P||Q$ is unsatisfiable, i.e., no state of $P||Q$ has a successor. ∎

It requires exponential time to check if a module P is non-blocking, which amounts to evaluating the boolean Π_2^p formula $(\forall X_P)(\exists X'_P)\, \tau_P$. To eliminate the need for this exponential check whenever two modules are composed synchronously, we define five increasingly larger classes of modules for which the non-blockingness of synchronous composition can be checked efficiently.

5.2.2 Moore modules

A *Moore module* is a module which (a) is non-blocking, and (b) the next values of output variables do not depend on the next values of input variables: that is, for all states s, t, and u, if $\tau_P[\![s \cup t']\!]$ and $t[O_P] = u[O_P]$, then $\tau_P[\![s \cup u']\!]$. These two conditions can be enforced syntactically, in a way that permits checking in linear time. For example, the transition predicate of a Moore module may be specified as a set of nondeterministic guarded commands, one for each primed output variable x' in O'_P. The guarded command

for x' assigns a value to x' such that (a) one of the guards negates the disjunction of the other guards, and (b) the guards and the right-hand sides of all assignments contain no primed variables. The synchronous composition of two composable Moore modules is again a Moore module, and therefore non-blocking. Hence, we say that any two composable Moore modules are *moore-composable*. However, since many non-blocking modules are not Moore modules, more general types of modules are of interest.

Example 5.2 In digital circuits, a typical example of a Moore module is a buffer. For example, it has output variables $O_P = \{x\}$, input variables $I_P = \{y\}$ and transition predicate $\tau_P = (y \wedge x') \vee (\neg y \wedge \neg x')$. In guarded commands, this can be written as $[\!] \ \mathrm{T} \ \rightarrow \ x' = y$. ∎

5.2.3 Statically typed modules (or Reactive Modules [AH99])

A *dependency relation* for a module P is an acyclic binary relation $\succ \subseteq O_P \times X_P$ between the output variables and the module variables (acyclicity means that the transitive closure is irreflexive). The module P *respects* the dependency relation $\succ$ at state s if, for all states t with $\tau_P[\![s \cup t']\!]$, for each subset $Y^i \subseteq I_P$ of input variables, and for each truth-value assignment u^i to the variables in Y^i, there is a state u with $\tau_P[\![s \cup u']\!]$ such that $u[Y^i] = u^i$, and $u[Z] = t[Z]$ for $Z = \{z \in X_P \mid (\text{not } z \succ^* y) \text{ for all } y \in Y^i\}$, where $\succ^*$ is the reflexive-transitive closure of $\succ$. A *statically typed module* $(P, \succ_P)$ consists of a module P and a dependency relation for P, such that (a) the module P is non-blocking, and (b) the module P respects the dependency relation $\succ_P$ at all states. These two conditions, as well as the acyclicity requirement on dependency relations, can be enforced syntactically in a way that permits checking in linear time. For example, we may use guarded commands as with Moore modules, except that the guards and the right-hand sides of assignments are allowed to contain primed variables with the following proviso: if the guarded command for x' contains a primed variable y', then $x \succ_P y$. We refer to the dependency relation $\succ_P$ of a statically typed module $(P, \succ_P)$ as a *static type* for the module P. Note that if $\succ'$ is a dependency relation for P, and $\succ_P$ is a subset of $\succ'$, then $\succ'$ is also a static type for P.

Every non-blocking module has a static type (have each output variable depend on all input variables). Hence, there are composable modules with static types whose synchronous composition does not have a static type. However, static types suggest a sufficient condition for the existence of compound static types which can be checked efficiently. Two statically typed modules $(P, \succ_P)$ and $(Q, \succ_Q)$ are statically composable,

or *static-composable*, if (1) the modules P and Q are composable, and (2) the relation $\succ_{P||Q} = \succ_P \cup \succ_Q$ is acyclic. Then, the relation $\succ_{P||Q}$ is a static type for the synchronous composition $P||Q$. Since acyclicity can be checked in linear time, so can the requirement if two statically typed modules are *static*-composable. However, two statically typed modules $(P, \succ_P)$ and $(Q, \succ_Q)$ may not be *static*-composable even though the compound module $P||Q$ is non-blocking.

Example 5.3 An inverter $(P, \succ_P)$ can be modeled as a statically typed module. In guarded commands, it can defined as: $[]\ \mathrm{T} \rightarrow\ y' := \neg x'$, from which we can derive its dependency relation: $y \succ_P x$. Note that P is not a Moore module.∎

Example 5.4 A module may have two static types, neither of which is a subset of the other. Let module P be such that $O_P = \{x_0, x_1\}$, $I_P = \{y\}$, and $\tau_P = (x_0' \oplus x_1' \oplus y_0')$. Using guarded commands, we can specify P in two ways:

$$(P, \succ_P) = \left\{ \begin{array}{l} []\ \mathrm{T} \rightarrow\ x_0' := \neg(x_1' \oplus y') \\ \quad \wedge \\ []\ \mathrm{T} \rightarrow\ x_1' := \mathrm{T} \\ []\ \mathrm{T} \rightarrow\ x_1' := \mathrm{F} \end{array} \right\} \quad (P, \succ'_P) = \left\{ \begin{array}{l} []\ \mathrm{T} \rightarrow\ x_0' := \mathrm{T} \\ []\ \mathrm{T} \rightarrow\ x_0' := \mathrm{F} \\ \quad \wedge \\ []\ \mathrm{T} \rightarrow\ x_1' := \neg(x_0' \oplus y') \end{array} \right\}$$

The two statically-typed modules have the same transition relation, namely, τ_P, but they have different static types: $\succ_P = \{x_0 \succ x_1,\ x_0 \succ y\}$ while $\succ'_P = \{x_1 \succ x_0,\ x_1 \succ y\}$. Choosing different static types (i.e., implementations of the transition predicate) can have implications on composability with other modules. Let module $(Q, \succ_Q)$ be such that $O_Q = \{y\}$, $I_Q = \{x_0, x_1\}$, and in guarded commands, $(Q, \succ_Q) = []\ \mathrm{T} \rightarrow\ y' := x_0'$. The static type $\succ_Q$ for Q is $\{y \succ x_0\}$. Then $(P, \succ_P)$ is not *static*-composable with $(Q, \succ_Q)$, but $(P, \succ'_P)$ is. ∎

5.2.4 Dynamically typed modules

Example 5.4 suggests the following generalization of static types. A *composite dependency relation* for a module P is a set $D = \{(\psi^1, \succ^1), \ldots, (\psi^m, \succ^m)\}$ of pairs, where each ψ^i is a predicate over the module variables X_P, and each $\succ^i$ is a dependency relation for P, such that for each state s of P, there is exactly one predicate ψ^i, $1 \leq i \leq m$, with $\psi^i[\![s]\!]$. If $\psi^i[\![s]\!]$, then we write $\succ^s$ for the corresponding dependency relation $\succ^i$. A *dynamically typed module* (P, D_P) consists of a module P and a composite dependency relation $D_P = \{(\psi_P^i, \succ_P^i) \mid 1 \leq i \leq m\}$, such that (a) the module P is non-blocking,

and (b) at every state s, the module P respects the dependency relation $\succ_P^s$. These two conditions, as well as the requirements on a composite dependency relation, can again be enforced syntactically in a way that permits checking in polynomial time. For example, each predicate ψ_P^i, $1 \leq i < m$, may be required to contain the conjunct $\bigwedge_{j \neq i} \neg \psi_P^j$, and ψ_P^m may be required to be equal to $\bigwedge_{1 \leq i < m} \neg \psi_P^i$. If we use guarded commands to specify the transition predicate, then for each guarded command, the guard may be required to contain a conjunct of the form ψ_P^i, for some $1 \leq i \leq m$, and together with the right-hand sides of assignments satisfy the proviso for the corresponding dependency relation $\succ_P^i$.

Example 5.5 Level-sensitive latches are commonly used in the design of high performance systems such as pipelined microprocessors. Typically different parts of a system are active depending on the phase of the clock. As an example, consider a circuit consisting of three modules P_1, P_2, and P_3. Module P_1 is an inverter which connects the output of the $\neg c$-clocked level-sensitive latch P_3 to the input of the c-clocked level-sensitive latch P_2. The output of the latch P_2 is connected to the input of the latch P_3. Using guarded commands, the three modules can be specified as follows:

$$(P_1, D_{P_1}) = \left\{ [\!] \ \mathrm{T} \ \rightarrow \ x' := \neg z' \right\} \qquad (P_2, D_{P_2}) = \left\{ \begin{array}{lll} [\!] \ c & \rightarrow & y' := x' \\ [\!] \ \neg c & \rightarrow & y' := y \end{array} \right\}$$

$$(P_3, D_{P_3}) = \left\{ \begin{array}{lll} [\!] \ c & \rightarrow & z' := z \\ [\!] \ \neg c & \rightarrow & z' := y' \end{array} \right\}$$

The dynamic types for the modules are $D_{P_1} = \{(\mathrm{T}, x \succ z)\}$, $D_{P_2} = \{(c, y \succ x), (\neg c, \emptyset)\}$, and $D_{P_3} = \{(c, \emptyset), (\neg c, z \succ y)\}$. ∎

We refer to the composite dependency relation D_P of a dynamically typed module (P, D_P) as a *dynamic type* for the module P. Like static types, dynamic types suggest sufficient conditions for the non-blocking of synchronous composition. Furthermore, the conditions for the composability of dynamic types are more liberal than *static*-composability, and thus they are applicable in more situations. Consider two dynamically typed modules (P, D_P) and (Q, D_Q) with $D_P = \{(\psi_P^i, \succ_P^i) \mid 1 \leq i \leq m\}$ and $D_Q = \{(\theta_Q^j, \succ_Q^j) \mid 1 \leq j \leq n\}$. We write $\succ^{i,j}$ for the union $\succ_P^i \cup \succ_Q^j$, where $1 \leq i \leq m$ and $1 \leq j \leq n$. We provide two definitions of composability for dynamically typed modules, one purely syntactic, and the other in part semantic.

- The dynamically typed modules (P, D_P) and (Q, D_Q) are syntactically dynamically composable, or *dsynt-composable*, if (1) the modules P and Q are composable, and (2) the relation $\succ^{i,j}$ is acyclic for all $1 \leq i \leq m$ and $1 \leq j \leq n$. Then, $D_{P||Q} = \{(\psi_P^i \wedge \theta_Q^j, \succ^{i,j}) \mid 1 \leq i \leq m \text{ and } 1 \leq j \leq n\}$, is a dynamic type for the synchronous composition $P||Q$.

- The dynamically typed modules (P, D_P) and (Q, D_Q) are semantically dynamically composable, or *dsem-composable*, if (1) the modules P and Q are composable, and (2) the relation $\succ^{i,j}$ is acyclic for all $1 \leq i \leq m$ and $1 \leq j \leq n$ for which the conjunction $\psi_P^i \wedge \theta_Q^j$ is satisfiable. Then, $D_{P||Q} = \{(\psi_P^i \wedge \theta_Q^j, \succ^{i,j}) \mid 1 \leq i \leq m \text{ and } 1 \leq j \leq n \text{ and } (\exists X_{P||Q})(\psi_P^i \wedge \theta_Q^j)\}$ is a dynamic type for $P||Q$.

Note that it can be checked in quadratic time whether two dynamically typed modules are *dsynt*-composable, while it requires exponential time (by evaluating a quadratic number of boolean Π_1^p formulas) to check if they are *dsem*-composable. However, checking if two dynamically typed modules are *dsem*-composable is still simpler than checking if the synchronous composition of two untyped modules is non-blocking (Π_1^p vs. Π_2^p).

Proposition 5.1 *The following assertions hold:*

- *There are two dynamically typed modules (P, D_P) and (Q, D_Q) which are dsynt-composable but not static-composable, even though the union of all dependency relations in D_P is a static type for P, and the union of all dependency relations in D_Q is a static type for Q.*

- *There are two dynamically typed modules which are dsem-composable but not dsynt-composable.*

Example 5.6 The dynamically typed modules (P_1, D_{P_1}) and (P_2, D_{P_2}) of Example 5.5 are *dsynt*-composable, and the compound module $Q = P_1||P_2$ has the dynamic type $D_Q = \{(c, y \succ x \succ z), (\neg c, x \succ z)\}$. The modules (Q, D_Q) and (P_3, D_{P_3}) are not *dsynt*-composable, but they are *dsem*-composable. The compound module $Q||P_3$ has the dynamic type $\{(c, y \succ x \succ z), (\neg c, x \succ z \succ y)\}$. ∎

5.2.5 Dependent type modules

Dynamic types can be further generalized by allowing the dependency relation to be a function not only of the current state, but also of the partial next state. We define the

variable dependency relation, which establishes the possible orders in which the variables can be assigned a value in order to determine the next state, and the dependencies among the values chosen. A *variable dependency relation* for P is a set $C = \{(\psi^1, \succcurlyeq^1), \ldots, (\psi^m, \succcurlyeq^m)\}$ of pairs, where each ψ^i is a boolean formula over the unprimed and primed module variables $X_P \cup X'_P$, and each $\succcurlyeq^i \subseteq O_P \times 2^{X_P}$ is a binary relation with the intention that if ψ^i holds in an extended state, and $x \succcurlyeq^i Y$, then x can be given a next value, and this value can depend on the next values of the variables in Y.

A variable dependency relation is a syntactic object; to make the variable dependencies more explicit, we define the corresponding *dependency function* $\tilde{C}$: $S_P \times R_P \times X_P \to 2^{X_P}$ as the function that, given an extended state $\langle s, t' \rangle$ and a variable x, specifies the set of variables on which x depends, as $\tilde{C}(s, t', x) = \bigcup\{Y \mid (\psi, (x, Z)) \in C \text{ and } \psi[\![s \cup t']\!]\}$.

The variable x is *enabled* for (P, C) at the extended state $\langle s, t' \rangle$ if $\tilde{C}(s, t', x) \subseteq Var(t)$. The module P *respects* the variable dependency relation C if for every pair of extended states $\langle s, t' \rangle$ and $\langle s, u' \rangle$ of P, for every variable $x \in X_P$ that is enabled for (P, C) at both extended states, and for every $b \in \mathbb{B}$, if $t[\tilde{C}(s, t', x)] = u[\tilde{C}(s, u', x)]$, then the extended state $\langle s, t' \cup \{(x', b)\}\rangle$ is an (x, τ_P)-successor of $\langle s, t' \rangle$ iff the extended state $\langle s, u' \cup \{(x', b)\}\rangle$ is an (x, τ_P)-successor of $\langle s, u' \rangle$. If the transition predicate of a module is specified by a set Γ of nondeterministic guarded commands, then the variable dependency relation can be deduced from Γ as follows: for each guarded command $[\![g \to x' = e$ in Γ, let $(g, \succcurlyeq) \in C$, where $\succcurlyeq = \{(x, Y) \mid y \in Y$ iff y' occurs in g or in $e\}$.

Example 5.7 [Mal94] Cyclic circuits are often used in hardware systems for minimizing the circuit size. As an example, consider the circuit in Figure 5.1. The output w is a function of the inputs s and x. The circuit consists of three dependent-type modules P, Q and R. In guarded commands, they are

$$(P, C_P) = \begin{array}{lll} [\![\neg s' & \to & y' = F(x') \\ [\![\ s' & \to & y' = F(z') \end{array} \qquad (Q, C_Q) = \begin{array}{lll} [\![\neg s' & \to & z' = G(y') \\ [\![\ s' & \to & z' = G(x') \end{array}$$

$$(R, C_R) = \begin{array}{lll} [\![\neg s' & \to & w' = z' \\ [\![\ s' & \to & w' = y' \end{array}$$

Module P has the variables $O_P = \{y\}$ and $I_P = \{s, x, z\}$, and the variable dependency relation $C_P = \{(\neg s', \{(y, \{x, s\})\}), (s', \{(y, \{z, s\})\})\}$. Module Q has the variables $O_Q = \{z\}$

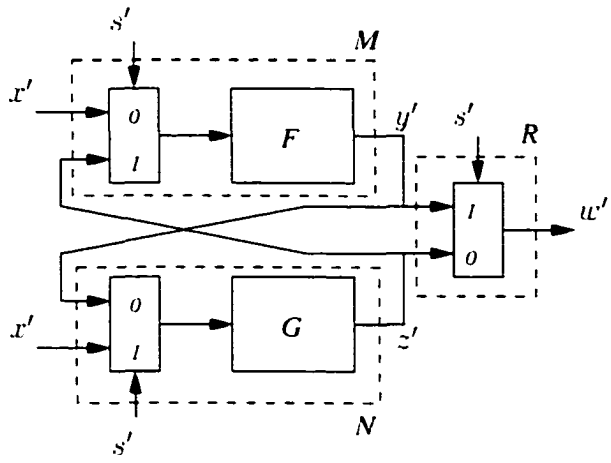


Figure 5.1: A cyclic circuit composed of three modules P, Q, and R. It performs the following function: if s' then $w' = F(G(x'))$ else $w' = G(F(x'))$, where F and G are two combinational blocks, such as a shifter and adder.

and $I_Q = \{s, x, y\}$, and the variable dependency relation $C_Q = \{(\neg s', \{(z, \{y, s\})\}), (s', \{(z, \{x, s\})\})\}$. Module R has the variables $O_R = \{w\}$ and $I_R = \{y, z\}$, and the variable dependency relation $C_R = \{(\neg s', \{(w, \{z, s\})\}), (s', \{(w, \{y, s\})\})\}$. ∎

To define dependent-type modules, and to define composition of two dependent-type modules, we need to know how the variable dependency changes as the state transitions take place. Therefore we define the *micro-step graph*, a graph depicting the sequence of partial states traversed as a new macro-step successor is determined.

Micro-step graph. Consider a module P and a variable dependency relations C_P for P. For a state $s \in S_P$, the *micro-step graph* $MG_s(P, C_P)$ is a directed acyclic graph whose vertices are the pairs $\langle s, t' \rangle$, where $t' \in R_P$ together with the additional distinguished vertex $\perp$, which is used to denote an illegal configuration. The edges of $MG_s(P, C_P)$ are partitioned into P-edges and E-edges: they are defined as follows, for all vertices $\alpha = \langle s, t' \rangle$ and all variables $x \in X_P$:

- If $x \in O_P$ and x is enabled for (P, C_P) at the extended state $\langle s, t' \rangle$, then for each (x, τ_P)-successor $\langle s, u' \rangle$ of α, there is an P-edge from α to the vertex $\langle s, u' \rangle$.
- If $x \in I_P$ and $x' \notin Var(P)$, then for each boolean constant $b \in \mathbb{B}$, if $\beta = \langle s, t' \cup \langle x', b \rangle \rangle$ is a (x, τ_P)-successor of α, then there is an E-edge from α to β. If β is not a (x, τ_P)-successor of α, then there is an E-edge from α to $\perp$.

A vertex of $MG_s(P.C_P)$ is *terminal* if it does not have any outgoing edges. Note that the micro-step graph has the following properties: there are at most $3^{|X_P|}$ vertices, the size of each vertex is at most $2 \cdot |X_P|$, and the depth of the graph is at most $|X_P| + 1$.

A *dependent-type module* $(P.C_P)$ consists of a module P and a variable dependency relation C_P such that (a) the module P respects the variable dependency relation C_P and (b) P is well-typed w.r.t. C_P at every state $s \in S_P$. A module P is said to be well-typed w.r.t. C_P at the state s, if there is a path in the micro-step graph $MG_s(P.C_P)$ from the initial vertex $\langle s, \emptyset \rangle$ to a terminal vertex α, then $\alpha \neq \perp$, and $\alpha = \langle s, t' \rangle$ for some state $t \in S_P$. Condition (b) states that for all environment inputs, the module P does not block. We refer to the variable dependency relation C_P as a *dependent type* for P.

Composition. Two dependent-type modules $(P.C_P)$ and $(Q.C_Q)$ are dependent-type composable, or *dep-composable*, if (1) P and Q are composable and (2) the composition $P \| Q$ is well-typed w.r.t. $C_{P\|Q}$ at every state $s \in S_{P\|Q}$. Then the variable dependency relation $C_{P\|Q}$ is a dependent type for the composite module $P \| Q$. The following theorem shows that checking composability for dependent-type modules has the same worst-case complexity as for dynamically typed modules.

Dependent types capture a larger class of non-blocking synchronous composition than dynamically typed modules, as shown by the following proposition. For a dependent-type module $(P.\pi_P.C_P)$, the variable $x \in X_P$ depends on $y \in X_P$ at a state $s \in S_P$, written $x \succ^s y$, if there exists a partial state $t' \in R_P$ and a pair $(\psi, \succcurlyeq) \in C_P$ such that $\psi[\![s \cup t']\!]$ and $x \succcurlyeq y$. Then $D_P = \{(s, \succ^s) \mid s \in S_P\}$ is a dynamic type for P.

Proposition 5.2 *There are two dependent-type modules that are dep-composable but not dsem-composable. There are two dependent-type modules $(P.C_P)$ and $(Q.C_Q)$ which are not dep-composable, even though the synchronous composition $P \| Q$ is non-blocking.*

Example 5.8 The dependent-type modules P, Q, and R from Example 5.7 are *dep*-composable. If these modules are viewed as dynamically typed modules, the output of each module will depend on their respective inputs. Hence there will be a cyclic dependency in the union of their dependency relations, namely, $y \succ z$ and $z \succ y$ at all states. Since dynamically typed modules do not permit cyclic dependencies, these modules are not *dsem*-composable. ∎

Example 5.9 A latch implemented as two NAND gates can be described as (Figure 5.2):

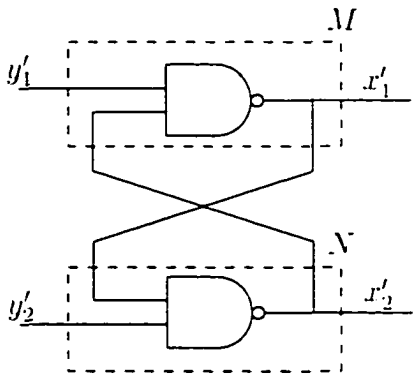


Figure 5.2: A latch implemented as two NAND gates.

$$(P.C_P) = [\![\mathrm{T} \rightarrow x'_1 = \neg(y'_1 \wedge x'_2) \qquad (Q.C_Q) = [\![\mathrm{T} \rightarrow x'_2 = \neg(y'_2 \wedge x'_1).$$

$(P.C_P)$ and $(Q.C_Q)$ are not *dep*-composable, although the composition $P\|Q$ is non-blocking at all states of $P\|Q$. The reason is that there is an inter-dependency of the variables x_1 and x_2 that cannot be resolved if $y'_1 = y'_2 = \mathrm{T}$. In fact, no dependent-type modules can describe this two-NAND gate system. ∎

Proposition 5.3 *Checking if two dependent-type modules are dep-composable is complete for coNP.*

Proof. To show that two dependent-type modules $(P.C_P)$ and $(Q.C_Q)$ are not *dep*-composable, one can check that either P and Q are not composable, or guess a state $s \in S_{P\|Q}$ and a path $\langle s.\emptyset\rangle.\langle s.t'_1\rangle.\langle s.t'_2\rangle.\ldots.\langle s.t'_n\rangle = \alpha$ in the micro-step graph $MG_s(P\|Q.C_{P\|Q})$, and check that $Var(t_n) \subseteq X_{P\|Q}$, and that there is no outgoing edge from α. This last condition can be checked by checking that $Var(t_n)$ contains all input variables $I_{P\|Q}$, and that no output variable undefined in t'_n is enabled at the extended state $\langle s.t'_n\rangle$. Note that this check requires only polynomial time, because a variable can be enabled only when all variables that occur in its enabling condition are assigned a value by $s \cup t'_n$.

Hardness comes from the fact that dynamically-typed module is a special case of dependent-type module, and therefore *dsem*-composability is a special case of *dep*-composability. We give a proof based on the reduction from tautology checking of boolean formulas. Given a boolean formula $\varphi(x_1,\ldots,x_n)$, we construct dependent-typed modules $(P_i.C_{P_i}).1 \leq i \leq n$ and $(Q.C_Q)$ and $(R.C_R)$ defined as follows: Module $(P_i.C_{P_i})$ has one output variable x_i and no input variable, and in guarded commands.

$$(P_i.C_{P_i}) = \begin{array}{lll} [\!] \ \mathrm{T} & \rightarrow & x_i' = \mathrm{T} \\ [\!] \ \mathrm{T} & \rightarrow & x_i' = \mathrm{F} \end{array}$$

Module $(Q_i.C_{Q_i})$ has one output variable x and one input variable y, and in guarded commands,

$$(Q.C_Q) = \begin{array}{lll} [\!] \ y' & \rightarrow & x' = \mathrm{T} \\ [\!] \ \neg y' & \rightarrow & x' = z' \end{array}$$

Module $(R.C_R)$ has $O_R = \{y, z\}$, $I_R = \{x_1, \ldots, x_n, x\}$, and guarded commands

$$(R.C_R) = \left\{ \begin{array}{lll} [\!] \ \mathrm{T} & \rightarrow & y' = \varphi(x_1', \ldots, x_n') \\ & \wedge & \\ [\!] \ \mathrm{T} & \rightarrow & z' = x' \end{array} \right\}$$

It is not hard to see that the modules $(P_i.C_{P_i})$, $1 \leq i \leq n$, $(Q.C_Q)$ and $(R.C_R)$ are *dep*-composable iff φ is a tautology. ∎

5.2.6 Summary of types.

Let $\Lambda = \{async, moore, static, dsynt, dsem, dep\}$ be the set of *module classes*. We summarize this section by defining, for each module class $\alpha \in \Lambda$, a set $\mathcal{M}_\alpha$ of modules: for $\alpha = async$, let $\mathcal{M}_{async}$ be the set of non-blocking modules; for $\alpha = moore$, let $\mathcal{M}_{moore}$ be the set of Moore modules; for $\alpha = static$, let $\mathcal{M}_{static}$ be the set of statically typed modules; for $\alpha = dsynt$ and $\alpha = dsem$, let $\mathcal{M}_{dsynt} = \mathcal{M}_{dsem}$ be the set of dynamically typed modules; and for $\alpha = dep$, let $\mathcal{M}_{dep}$ be the set of dependent-type modules. Define the *module class ordering* $async < moore < static < dsynt < dsem < dep$. Then, for $\alpha, \beta \in \Lambda$ with $\alpha < \beta$, every module $P \in \mathcal{M}_\alpha$ can be considered to be a module in $\mathcal{M}_\beta$ by adjusting its type, or the semantics of composition, if necessary. Precisely: an *async*-module can be considered as a *moore*-module by changing the semantics of composition; a *moore*-module can be considered a *static*-module with the empty dependency relation; a *static*-module can be considered a dynamically typed module with a single dependency relation, and a *dsem*-module can be considered a *dep*-module whose variable dependency relation is only a function of the state. We also define for each module class $\alpha \in \Lambda$ a corresponding composition operator $\|_\alpha$: if $\alpha = async$, then $\|_\alpha = |$: otherwise, $\|_\alpha = \|$.

5.3 Application: Constructive Semantics

We provide an alternative definition of constructive semantics [SBT96, Ber98] of synchronous languages using our type systems. Constructive semantics can be defined in three different ways: constructive operational semantics, based on constructive boolean logic; constructive behavioral semantics, based on Scott's fixed point semantics; and circuit semantics, based on the up-bounded inertial delay model. The circuit semantics captures exactly the set of programs described in synchrouns languages such as Esterel that can be translated into delay-insensitive digital circuits. It was shown that the three semantics were equivalent.

We show that dependent types can be used to define constructive semantics, by showing the equivalence between constructive operational semantics and dependent types. The following formulation of constructive semantics is taken from [Ber99].

5.3.1 Boolean circuits

A boolean circuit $\mathcal{C}$ is defined by a set $X_{\mathcal{C}}$ of variables or wires and a set of wire definitions. The set $X_{\mathcal{C}}$ is partitioned into a set $O_{\mathcal{C}}$ of output wires and a set $I_{\mathcal{C}}$ of input wires. Boolean expressions e are composed of wires w, constants F and T, and connectives $\neg$, $\vee$ and $\wedge$. Each output wire is defined by exactly one wire definition. There are two kinds of wire definitions:

- An *equality* definition $w = e$, i.e., the *next* value of wire is the value of e evaluated with the *next* values of the wires X. The wire w is called a combinational wire.
- An *register* definition $w := e$, i.e., the next value of w is the value of e evaluated with the *current* values of the wires X. The wire w is called a register.

The set of combinational wire is denoted by $\mathcal{S}_{\mathcal{C}}$, while the set of registers is denoted by $\mathcal{R}_{\mathcal{C}}$. An input v for $\mathcal{C}$ is a truth value assignment to the the input wires $I_{\mathcal{C}}$. An state r of $\mathcal{C}$ is a truth value assignment to the the registers $\mathcal{R}_{\mathcal{C}}$.

5.3.2 Constructive operational semantics

Given a boolean circuit $\mathcal{C}$, an input v for $\mathcal{C}$, a state r of $\mathcal{C}$, a wire expression e, and a Boolean value b, the constructive evaluation relation $v, r \vdash e \hookrightarrow b$ is defined inductively as follows.

$$
\begin{array}{llll}
v, r & \vdash_{\mathcal{C}} & b \hookrightarrow b & \\
v, r & \vdash_{\mathcal{C}} & w \hookrightarrow b & \text{if } w \in I_{\mathcal{C}} \text{ and } v(w) = b \\
v, r & \vdash_{\mathcal{C}} & w \hookrightarrow b & \text{if } w \in \mathcal{R}_{\mathcal{C}} \text{ and } r(w) = b \\
v, r & \vdash_{\mathcal{C}} & w \hookrightarrow b & \text{if } w = e \in \mathcal{C} \text{ and } v, r \vdash_{\mathcal{C}} e \hookrightarrow b \\
v, r & \vdash_{\mathcal{C}} & \neg e \hookrightarrow b & \text{if } v, r \vdash_{\mathcal{C}} e \hookrightarrow \neg b \\
v, r & \vdash_{\mathcal{C}} & e_1 \vee e_2 \hookrightarrow 1 & \text{if } v, r \vdash_{\mathcal{C}} e_1 \hookrightarrow 1 \text{ or } v, r \vdash_{\mathcal{C}} e_2 \hookrightarrow 1 \\
v, r & \vdash_{\mathcal{C}} & e_1 \vee e_2 \hookrightarrow 0 & \text{if } v, r \vdash_{\mathcal{C}} e_1 \hookrightarrow 0 \text{ and } v, r \vdash_{\mathcal{C}} e_2 \hookrightarrow 0 \\
v, r & \vdash_{\mathcal{C}} & e_1 \wedge e_2 \hookrightarrow 1 & \text{if } v, r \vdash_{\mathcal{C}} e_1 \hookrightarrow 1 \text{ and } v, r \vdash_{\mathcal{C}} e_2 \hookrightarrow 1 \\
v, r & \vdash_{\mathcal{C}} & e_1 \wedge e_2 \hookrightarrow 0 & \text{if } v, r \vdash_{\mathcal{C}} e_1 \hookrightarrow 0 \text{ or } v, r \vdash_{\mathcal{C}} e_2 \hookrightarrow 0
\end{array}
$$

A boolean circuit $\mathcal{C}$ is constructive w.r.t. v and r if, for any wire w, $v, r \vdash_{\mathcal{C}} w \hookrightarrow b$ for some b. Moreover, if $v, r \vdash_{\mathcal{C}} w \hookrightarrow b$ and $v, r \vdash_{\mathcal{C}} w \hookrightarrow b'$ then $b = b'$.

5.3.3 From boolean circuits to modules

We translate a boolean circuit $\mathcal{C}$ to a set $\mathcal{M}(\mathcal{C})$ of dependent-type modules. To facilitate the translation, we give names to each subexpression that appears in the wire definitions of the boolean circuit $\mathcal{C}$. More precisely, we define a *extended circuit* $\bar{\mathcal{C}}$ as the following: For each wire definition $w = e$ (resp. $w := e$) , we have the wire definition $w = w_e$ (resp. $w := w_e$) in $\bar{\mathcal{C}}$. For each subexpression f of e, we introduce a fresh auxillary variable w_f, and add the wire definition d_f to $\bar{\mathcal{C}}$, where d_f is defined as follows:

$$
d_f = \begin{cases}
w_f = \tau & \text{if } f = b \in \mathbb{B} \\
w_f = w & \text{if } f = w \\
w_f = \neg w_{f_1} & \text{if } f = \neg f_1 \\
w_f = w_{f_1} \vee w_{f_2} & \text{if } f = f_1 \vee f_2 \\
w_f = w_{f_1} \wedge w_{f_2} & \text{if } f = f_1 \wedge f_2
\end{cases}
$$

Clearly, $X_{\bar{\mathcal{C}}} \supseteq X_{\mathcal{C}}$, $I_{\bar{\mathcal{C}}} = I_{\mathcal{C}}$ and $\mathcal{R}_{\bar{\mathcal{C}}} = \mathcal{R}_{\mathcal{C}}$. It is not hard to see that given an input v and a state r, $\mathcal{C}$ is constructive w.r.t. v and r iff $\bar{\mathcal{C}}$ is constructive w.r.t. v and r. To prove this, note that for any wire $w \in X_{\mathcal{C}}$, $v, r \vdash_{\mathcal{C}} w \hookrightarrow b$ iff $v, r \vdash_{\bar{\mathcal{C}}} w \hookrightarrow b$. Moreover, for each auxilliary variable w_e, since it is a boolean combination of the boolean constants, wires w or auxilliary variables w_f corresponding to the subexpressions f of e, and every w has a unique value, w_e also constructively evaluates to some unique boolean value. Therefore, if $\mathcal{C}$ is constructive, then $\bar{\mathcal{C}}$ is also constructive. The other direction is trivial.

Now we define the translation from a boolean circuit $\mathcal{C}$ to a set $\mathcal{M}(\mathcal{C})$ of dependent-type modules represented in guarded commands. For any wire definition $d \in \bar{\mathcal{C}}$, we have in $\mathcal{M}(\mathcal{C})$ the corresponding module $M(d)$ defined as follows. In the following, the variable x is the output variable of the module $M(d)$, while the variables w, x_1, x_2 and w_e are the input variables (if they appear in the guarded command of $M(d)$).

$$
\begin{aligned}
M(x = b) &= [\![\mathsf{T} \rightarrow x' = b. \text{ where } b \in \mathbb{B} \\
M(x = w) &= [\![\mathsf{T} \rightarrow x' = w'. \text{ where } w \in \mathcal{S}_{\bar{\mathcal{C}}} \\
M(x = w) &= [\![\mathsf{T} \rightarrow x' = w. \text{ where } w \in \mathcal{R}_{\bar{\mathcal{C}}} \\
M(x = \neg x_1) &= [\![\mathsf{T} \rightarrow x' = \neg x_1' \\
M(x = x_1 \vee x_2) &= \left\{ \begin{array}{lll} [\![x_1' & \rightarrow & x' = \mathsf{T} \\ [\![x_2' & \rightarrow & x' = \mathsf{T} \\ [\![\neg x_1' \wedge \neg x_2' & \rightarrow & x' = \mathsf{F} \end{array} \right\} \\
M(x = x_1 \wedge x_2) &= \left\{ \begin{array}{lll} [\![\neg x_1' & \rightarrow & x' = \mathsf{F} \\ [\![\neg x_2' & \rightarrow & x' = \mathsf{F} \\ [\![x_1' \wedge x_2' & \rightarrow & x' = \mathsf{T} \end{array} \right\} \\
M(x := w_e) &= [\![\mathsf{T} \rightarrow x' = w_e'
\end{aligned}
$$

Given an input v for $\mathcal{C}$, let $\mathcal{M}_I(v)$ be the set of modules which provide inputs to the modules in $\mathcal{M}(\mathcal{C})$. It is defined as follows: For each assignment $x = b$ in v, we have the module $[\![\mathsf{T} \rightarrow x' = b$ in $\mathcal{M}_I(v)$. The following theorem establishes the relationship between constructive semantics and dependent-type modules.

Theorem 5.1 *Let $\mathcal{C}$ be a boolean circuit and $\mathcal{M}(\mathcal{C})$ be the corresponding dependent-type modules. Then $\mathcal{C}$ is constructive w.r.t. v, r iff $(P.C_P) = \|\{(Q.C_Q) \mid (Q.C_Q) \in \mathcal{M}(\mathcal{C}) \vee (Q.C_Q) \in \mathcal{M}_I(v)\}$ is well-typed w.r.t. C_P at the state r.*

Proof. Consider the microstep graph $MG_r(P.C_P)$. To prove the if direction, we show by induction that for any extended state $\langle r, t' \rangle$ of P that, if there exists a path from the extended state $\langle r, \emptyset \rangle$ to $\langle r, t' \rangle$, then for all truth value assignment $(x', b) \in t'$, if $x \notin \mathcal{R}$, then x constructively evaluates to b in the boolean circuit $\bar{\mathcal{C}}$. Consider the extended state $\langle r, t' \cup (x', b) \rangle$:

- Suppose the wire definition $x = y_1 \vee y_2$ is in $\bar{\mathcal{C}}$. Consider the module $M(x = y_1 \vee y_2)$: x is enabled at $\langle r, t' \rangle$ when (1) $(y_1', \mathsf{T}) \in t'$, or (2) $(y_2', \mathsf{T}) \in t'$, or (3) both $(y_1', \mathsf{F}) \in t'$ and

$(y_2', \mathsf{F}) \in t'$. If (1) is true, then by induction hypothesis, y_1 constructively evaluates to T in $\bar{\mathcal{C}}$ and therefore x evaluates to $b = \mathsf{T}$ in $\bar{\mathcal{C}}$. The cases for (2) and (3) are similar.

- The cases for the other wire definitions are similar.

Therefore, if P is well-typed w.r.t. C_P at the state r, then $\bar{\mathcal{C}}$ and hence $\mathcal{C}$ is constructive with respect to v, r.

To prove the only-if direction, assume that the combinational wires in $\bar{\mathcal{C}}$ constructively evaluate to their respective values w.r.t. v, r in the order $x_{\eta_1}, x_{\eta_2}, \ldots$, i.e., a proof sequence of the values of the combinational wires. Assume that there is a path in the microstep graph $MG_r(P, C_P)$ from the extended state $\langle r, \emptyset \rangle$ to an extended state $\langle r, t' \rangle$, and that for all truth value assignment $(x', b) \in t'$, if $x \notin \mathcal{R}_{\bar{\mathcal{C}}}$, then x constructively evaluates to b in $\bar{\mathcal{C}}$ w.r.t. v, r.

- If there exists combination wire $x \in \mathcal{S}_{\bar{\mathcal{C}}}$ such that $x' \notin Var(t')$, then let i be the least integer such that $x'_{\eta_i} \notin Var(t')$. Suppose $\bar{\mathcal{C}}$ contains the wire definition $x_{\eta_i} = x_p \vee x_q$, then there exists $j, k \leq i$ such that (1) $p = \eta_j \wedge x_{\eta_j} = \mathsf{T}$, or (2) $q = \eta_k \wedge x_{\eta_k} = \mathsf{T}$, or (3) $p = \eta_j \wedge q = \eta_k \wedge x_{\eta_j} = x_{\eta_k} = \mathsf{F}$. Suppose (1) holds, then by induction hypothesis, $((x'_{\eta_j}, \mathsf{T}) \in t')$ and therefore x'_{η_i} is enabled at the extended state $\langle r, t' \rangle$. Moreover, it can only be assigned the value which x_{η_i} constructively evaluates to in $\bar{\mathcal{C}}$. Cases (2) and (3) are similar. Also, the cases for the other equality definitions are similar.

- Otherwise, if there exists register $x \in \mathcal{R}_{\bar{\mathcal{C}}}$ such that $x' \notin Var(t')$, since $w'_e \in Var(t')$, where $x := w_e \in \bar{\mathcal{C}}$, then x is enabled at the extended state $\langle r, t' \rangle$.

Therefore, if $\mathcal{C}$ (and hence $\bar{\mathcal{C}}$) is constructive with respect to v, r, then P is well-typed w.r.t. C_P at the state r. ∎

5.4 Untyped and Typed Control Problems

Single-step vs. multi-step verification. Given a module P, a state s of P, and a predicate φ over the module variables X_P, the *single-step verification problem* (P, s, φ) asks whether for all states t, if $\tau[\![s \cup t']\!]$, then $\varphi[\![t]\!]$. The single-step verification problem amounts to evaluating the boolean Π_1^p formula $(\forall X'_P)(\tau_P \rightarrow \varphi')[s]$, where φ' results from φ by replacing all variables with their primed counterparts. A *run* r of a module P is a

finite sequence $s_0 s_1 \dots s_k$ of states of P such that $\tau_P[\![s_i \cup s'_{i+1}]\!]$ for all $0 \leq i < k$. The run r is *s-rooted*, for a state s of P, if $s_0 = s$. The run r *stays in* φ, for a predicate φ over the set X_P of module variables, if $\varphi[\![s_i]\!]$ for all $0 \leq i \leq k$. Given a module P, a state s of P, and a predicate φ over X_P, the *multi-step (invariant) verification problem* (P, s, φ) asks whether all s-rooted runs of P stay in φ. The multi-step verification problem can be solved by iterating the solution for the single-step verification problem. The number of states, which is exponential, gives a tight bound on the number of iterations.

Theorem 5.2 (cf. [AH98]) *The single-step verification problem is complete for coNP. The multi-step verification problem is complete for PSPACE.*

In control, it is natural to require that the controller falls into the same module class as the plant. Consider a module class $\alpha \in \Lambda$ and a module $P \in \mathcal{M}_\alpha$. The module $Q \in \mathcal{M}_\alpha$ is an *α-controller* for P if (1) P and Q are α-composable, and (2) $O_Q = I_P$ and $I_Q = O_P$. According to this definition, a controller for P is an environment of P which has no state on its own. For the control problems we consider in this chaper, the results would remain unchanged if we were to consider controllers with state. As in verification, we distinguish between single-step and multi-step control. The single-step (resp. multi-step) control problem asks if there is a controller for a module that ensures that, starting from a given state, a given predicate holds after one step (resp. any number of steps). Precisely, for a module class α, a module $P \in \mathcal{M}_\alpha$, a state s of P, and a predicate φ over the set X_P of module variables, the *single-step* (resp. *multi-step*) *α-control problem* (P, s, φ) asks whether there is an α-controller Q for P such that the answer to the single-step (resp. multi-step) verification problem $(P \|_\alpha Q, s, \varphi)$ is Yes. If the answer is Yes, then the state s is single-step (resp. multi-step) *controllable* by Q with respect to the *control objective* φ.

Fixed-type control. For $\alpha \in \{static, dsynt, dsem, dep\}$, we also consider a variant of the control problems in which the type of the controller module is known (but its transiton relation is not). An instance (P, γ, s, φ) of the *fixed-type* single-step (resp. multi-step) α-control problem consists of an instance (P, s, φ) of the single-step (resp. multi-step) α-control problem together with a type γ for the controller. For $\alpha = static$, the type γ is a dependency relation for an α-controller for P: for $\alpha \in \{dsynt, dsem\}$, the type γ is a composite dependency relation for an α-controller for P: for $\alpha = dep$, the type γ is a variable dependency relation for an α-controller for P. The instance (P, γ, s, φ) asks whether there

is an α-controller Q of type γ for P such that the answer to the single-step (resp. multi-step) verification problem $(P \|_\alpha Q, s, \varphi)$ is Yes.

Generality of controllers. For a module class α, consider a module $P \in \mathcal{M}_\alpha$, a single-step (resp. multi-step) control objective φ, and two α-controllers Q and Q' for P. The controller Q is *as state-general as* Q' if all states s of P which are single-step (resp. multi-step) controllable by Q' with respect to φ are also single-step (resp. multi-step) controllable by Q with respect to φ. Moreover, if Q and Q' are equally state-general (i.e., Q is as state-general as Q', and vice versa), then Q is *as choice-general as* Q' if the transition predicate $\tau_{Q'}$ implies τ_Q (i.e., Q permits as much nondeterminism as Q'). An α-controller is *most state-general* if it is as state-general as any other α-controller. A α-controller is *most general* if (1) it is most state-general, and (2) it is as choice-general as any other most state-general α-controller.

Summary of results. In the following section, we present algorithms for solving the various types of control problems. The complexity results are summarized in Table 5.1(a). We recall that the complexity class DP consists of the languages which are intersections of an NP language and a coNP language. If n is the input size, the complexity class NE is $\bigcup_{k>0} \mathrm{NTIME}(2^{kn})$, and the complexity class EXP is $\bigcup_{k>0} \mathrm{DTIME}(2^{n^k})$. By the padding argument, any problem complete for NE is also complete for NEXP $= \bigcup_{k>0} \mathrm{NTIME}(2^{n^k})$ [Pap94]. Hence, assuming P $\neq$ NP, for the module classes *static*, *dsynt*, and *dsem*, the fixed-type multi-step control problems are harder than the multi-step control problems with arbitrary controller type. In addition, we summarize in Table 5.1(b) all results on the existence of most state-general and most general controllers.

5.5 Algorithms and Complexity of Control

We determine the complexity for solving the single-step and multi-step α-control problems for all six module classes $\alpha \in \Lambda$. In each case, the multi-step control problem can be solved by iterating an exponential number of times the solution for the corresponding single-step control problem.

Class	Composability Check	Single-Step		Multi-Step	
		Arbitrary	Fixed	Arbitrary	Fixed
async	$\mathcal{O}(n)$	DP	—	EXP	—
moore	$\mathcal{O}(n)$	Σ_2^p	—	EXP	—
static	$\mathcal{O}(n)$	PSPACE	NE	EXP	NE
dsynt	$\mathcal{O}(n^2)$	PSPACE	NE	EXP	NE
dsem	coNP	PSPACE	NE	EXP	NE
dep	coNP	PSPACE	NE	EXP	NE

(a) Complexity Results.

Class	MSG	MG
async	yes	yes
moore	yes	yes
static	no	no
dsynt	no	no
dsem	yes	no
dep	yes	no

(b) Existence of controllers.

Table 5.1: (a) Complexity of composability checking, as well as single-step and multi-step control for the various module classes. For statically and dynamically typed modules, we consider both arbitrary and fixed controller types. The quantity n is the size of the module description. Each problem is complete for the corresponding complexity class. (b) Existence of most state-general (MSG) and most general (MG) controllers.

5.5.1 Asynchronous control

Given a non-blocking module P, a state s of P, and a predicate φ over the module variables X_P, the single-step *async*-control problem amounts to evaluating the boolean formula

$$\Big((\forall O'_P)(\tau_P \wedge (I'_P = I_P) \rightarrow \varphi') \wedge (\exists I'_P)(\tau_P \wedge (O'_P = O_P) \wedge \varphi')\Big)[s].$$

Hence, in the asynchronous case, the single-step control problem is complete for DP. It follows from [CKS81] that the multi-step version is complete for exponential time (cf. [HK97]).

Theorem 5.3 *The single-step async-control problem is complete for DP. The multi-step async-control problem is complete for EXP.*

Proposition 5.4 *For every non-blocking module and every control objective, there is a most general single-step async-controller, and there is a most general multi-step async-controller.*

5.5.2 Moore control

Given a Moore module P, a state s of P, and a predicate φ over X_P, the single-step *moore*-control problem amounts to evaluating the boolean Σ_2^p formula $(\exists I'_P)(\forall O'_P)(\tau_P \rightarrow \varphi')[s]$. The multi-step hardness proof is similar to the asynchronous case.

Theorem 5.4 *The single-step moore-control problem is complete for Σ_2^p. The multi-step moore-control problem is complete for EXP.*

Proposition 5.5 *For every Moore module and every control objective, there is a most general single-step moore-controller, and there is a most general multi-step moore-controller.*

5.5.3 Statically typed control

Consider a statically typed module $(P, \succ_P)$, and let $X_P = \{x_1, \ldots, x_n\}$. A linear order $x_{i_1}, x_{i_2}, \ldots, x_{i_n}$ of the variables in X_P is *compatible* with the dependency relation $\succ_P$ if each output variable follows in the ordering the variables on which it depends. Precisely, $x_{i_1}, x_{i_2}, \ldots, x_{i_n}$ is compatible with $\succ_P$ if for all $1 \leq j, k \leq n$, if $x_{i_j} \succ x_{i_k}$, then $k < j$. Given a predicate φ over X_P, for each linear order $\ell = x_{i_1}, x_{i_2}, \ldots, x_{i_n}$, we define the boolean formula

$$C(\ell, \varphi) \;=\; (\lambda_{i_1} x'_{i_1})(\lambda_{i_2} x'_{i_2}) \cdots (\lambda_{i_n} x'_{i_n})(\tau_P \rightarrow \varphi'),$$

where for $1 \leq k \leq n$, we have $\lambda_{i_k} = \forall$ if $x_{i_k} \in O_P$, and $\lambda_{i_k} = \exists$ if $x_{i_k} \in I_P$. The following lemma states that, in order to decide whether a state is single-step *static*-controllable, it suffices to consider all linear orders of variable dependencies.

Lemma 5.1 *Given a statically typed module $(P, \succ_P)$, a control objective φ over X_P, and a state s of P, the state s is single-step static-controllable with respect to φ iff there is a linear order ℓ of X_P compatible with $\succ_P$ such that $C(\ell, \varphi)[\![s]\!]$.*

The lemma is proved by showing that (1) if a state of the statically typed module $(P, \succ_P)$ is output by a statically typed module $(Q, \succ_Q)$, then there is a linear order ℓ that strengthens the transitive closure of $\succ_P \cup \succ_Q$ such that $C(\ell, \varphi)$, and (2) if $C(\ell, \varphi)$ for some linear order

ℓ. then we can extract from ℓ a dependency relation $\succ_Q$ for the controller which ensures controllability.

Theorem 5.5 *The single-step static-control problem is complete for PSPACE. The multi-step static-control problem is complete for EXP.*

The single-step *static*-control problem is in PSPACE. because we can check each linear order in PSPACE. Hardness for PSPACE follows from the fact that. given a boolean formula $(\forall x_0)(\exists y_0)\cdots(\forall x_n)(\exists y_n)\varphi$. we can encode the problem of deciding its truth value as the *static*-control problem with the control objective φ for a module P with the variables $O_P = \{x_0, \ldots, x_n\}$ and $I_P = \{y_0, \ldots, y_n\}$. the valid transition relation τ_P. and the dependency relation $\succ_P = \{(x_i, y_j) \mid 1 \leq j < i \leq n\}$.

Note that in the common special case that the dependency relation $\succ_P$ is empty. the single-step *static*-control problem amounts to evaluating the boolean Π_2^p formula $(\forall O'_P)(\exists I'_P)(\tau_P \wedge \varphi')[s]$. This case is dual to the Moore case. because here the controller can choose the next values of the input variables dependent on the next values of all output variables. The corresponding multi-step problem is again complete for EXP.

We consider now the case in which the type $\succ_Q \subseteq I_P \times X_P$ of the controller is fixed. We assume that $\succ_P \cup \succ_Q$ is acyclic: otherwise. P and Q are not *static*-composable. and the answer to the fixed-type control problems is No. Let $O_P = \{x_1, \ldots, x_m\}$ and $I_P = \{y_1, \ldots, y_k\}$. Intuitively. for $1 \leq i \leq k$. the next value for y_i can be chosen in terms of the current values of the module variables. as well as in terms of the next value of the output variables on which y_i depends. Hence. a controller with fixed static type $\succ_Q$ can be thought of as a set $\{f_1, \ldots, f_k\}$ of Skolem functions: for $1 \leq i \leq k$. the Skolem function f_i provides a next value for y_i. and has as arguments the variables in $X_P \cup \{x' \in O'_P \mid y_i \succ_Q x\}$. This set of Skolem functions corresponds to the following boolean formula with Henkin quantifiers [Hen61. Wal70]:

$$H(\succ_Q, \varphi) = \begin{pmatrix} (\forall \{x' \in O'_P \mid y_1 \succ_Q x\})(\exists y'_1) \\ \cdots \\ (\forall \{x' \in O'_P \mid y_k \succ_Q x\})(\exists y'_n) \end{pmatrix} (\tau_P \rightarrow \varphi').$$

The fixed-type single-step *static*-control problem can be solved as follows.

Lemma 5.2 *Given a statically typed module $(P, \succ_P)$. a static controller type $\succ_Q \subseteq I_P \times X_P$ which is static-composable with $\succ_P$. a control objective φ over X_P. and a state s of P. the*

state s is single-step static-controllable with respect to φ by a controller with static type $\succ_Q$ iff $H(\succ_Q, \varphi)[\![s]\!]$.

Deciding the truth value of a boolean formula with Henkin quantifiers is complete for NE, even if the formula has the restricted form shown above [GLV95].

Theorem 5.6 *The fixed-type single-step and multi-step static-control problems are complete for NE.*

Unlike Moore modules, a statically typed module may not have a most state-general controller.

Proposition 5.6 *There is a statically typed module and a control objective such that there is no most state-general single-step static-controller, nor a most state-general multi-step static-controller.*

Example 5.10 Let $(P, \succ_P)$ be a statically-typed module having the output variables $O_P = \{x_0, x_1, z\}$, the input variables $I_P = \{y_0, y_1\}$, the transition predicate $\tau_P = (z' = z)$, and the static type $\succ_P = \{x_0 \succ y_0,\ x_1 \succ y_1\}$. The control objective is $\varphi = (z \wedge (y_1 = x_0)) \vee (\neg z \wedge (y_0 = x_1))$. For every state s of P there is a *static*-controller $(Q, \succ_Q)$ such that s is *static*-controllable by $(Q, \succ_Q)$ with respect to φ: If $z[\![s]\!]$, then $\tau_Q = (y_1' = x_0')$ and $y_1 \succ_Q x_0$: If $\neg z[\![s]\!]$, then $\tau_Q = (y_0' = x_1')$ and $y_0 \succ_Q x_1$. However, due to the acyclicity requirement for dependency relations, there is no single *static*-controller that controls all states of P. For the same reason, $(P, \succ_P)$ does not have a most state-general multi-step *static*-controller for the control objective φ. ∎

5.5.4 Dynamically typed control

The solution of control problems for dynamically typed modules closely parallels the solution for statically typed modules.

Lemma 5.3 *Given a dynamically typed module $(P, \{(w_P^i, \succ_P^i) \mid 1 \leq i \leq m\})$, a control objective φ over X_P, and a state s of P, the following assertions hold:*

- *The state s is single-step dsynt-controllable with respect to φ iff there is a linear order ℓ of X_P which is compatible with all $\succ_P^i$, for $1 \leq i \leq m$, such that $C(\ell, \varphi)[\![s]\!]$.*

- *The state s is single-step dsem-controllable with respect to φ iff there is a linear order ℓ of X_P which is compatible with $\succ_P^s$, such that $C(\ell, \varphi)[\![s]\!]$.*

Theorem 5.7 *For $\alpha \in \{dsynt, dsem\}$, the single-step α-control problem is complete for PSPACE, and the multi-step α-control problem is complete for EXP.*

Hence, the control problems for statically and dynamically typed modules have the same complexity. This applies also to the fixed-type control problems.

Lemma 5.4 *Given a dynamically typed module (P, D_P), a module class $\alpha \in \{dsynt, dsem\}$, a dynamic controller type $D_Q = \{(\psi_Q^i, \succ_Q^i) \mid 1 \leq i \leq m\}$ which is α-composable with D_P, a control objective φ over X_P, and a state s of P, the state s is single-step α-controllable with respect to φ by a controller with dynamic type D_Q iff $H(\succ_Q^s, \varphi)[\![s]\!]$.*

Theorem 5.8 *For $\alpha \in \{dsynt, dsem\}$, the fixed-type single-step and multi-step α-control problems are complete for NE.*

Dynamically typed modules with syntactic composition do not necessarily have a most state-general controller. In contrast, dynamically typed modules with semantic composition always have a most state-general controller, but they may not have a most general one.

Proposition 5.7 *The following assertions hold:*

- *There is a dynamically typed module and a control objective such that there is no most state-general single-step dsynt-controller, nor a most state-general multi-step dsynt-controller.*
- *For every dynamically typed module and every control objective, there is a most state-general single-step dsem-controller, and there is a most state-general multi-step dsem-controller. However, there is a dynamically typed module and a control objective such that there is no most general single-step dsem-controller, nor a most general multi-step dsynt-controller.*

Example 5.11 The statically-typed module $(P, \succ_P)$ of Example 5.10 can be viewed as a dynamically typed module (P, D_P) whose dependency relation is the same for every state, i.e., $D_P = \{(\top, \succ_P)\}$. The control objective is $\varphi = ((y_1 = x_0) \vee (y_0 = x_1))$. There exists at least two single-step *dsem*-controllers which control every state of P: the first

controller (Q_1, D_{Q_1}) has the transition predicate $\tau_{Q_1} = (y_1' = x_0')$ and dependency relation $\{\mathsf{T}.y_1 \succ x_0\}$: the second controller Q_2 has the transition predicate $\tau_{Q_2} = (y_0' = x_1')$ and dependency relation $\{\mathsf{T}.y_0 \succ x_1\}$: However, there is no most general single-step *dsem*-controller. To control P with respect to φ in a most general way, a controller Q with the transition predicate $\tau_Q = ((y_0' = x_1') \vee (y_1' = x_0'))$ would be required. Such a controller can be typed only if dynamic types are generalized to admit disjunctions of composite dependency relations. ∎

5.5.5 Dependent type control

Consider the single-step arbitrary-type control problem $((P.C_P), s, \varphi)$. It is convenient to view the control problem as a game between the dependent-type module $(P.C_P)$ and its controller. The game is played on the *reduced micro-step graph* $RMG_s(P.C_P)$, obtained from the micro-step graph $MG_s(P.C_P)$ by pruning for all *mixed* vertices α, i.e., vertices having both outgoing P- and E-edges, all E-edges outgoing from α. Intuitively, the reduced graph represents the situation in which the module P has precedence over E in updating variable values.

Note that every nonterminal vertex of the reduced micro-step graph either has only outgoing P-edges or has only outgoing E-edges. We call the vertices with only outgoing P-edges the *module vertices*, and the vertices with only outgoing E-edges the *environment vertices*. Then we solve the game by the following marking algorithm. A terminal vertex $\alpha = \langle s, t' \rangle$ is marked if $Var(t') \supseteq X_P'$ and $\varphi[\![t]\!]$. A module vertex α is marked if all successors β of α are marked. An environment vertex α is marked if some successor β of α is marked. The answer to the given single-step unknown-type control problem is Yes iff the vertex $\langle s, \emptyset \rangle$ is marked.

If the answer to the control problem is Yes, then the marking algorithm also suggests a way of synthesizing a dependent-type controller $(Q.C_Q)$ as a set of guarded commands Γ. Given a state $s \in S_P$, denote by χ_s the *characteristic formula* of s, defined by $\chi_s = \bigwedge\{x \mid (x, \mathsf{T}) \in s\} \wedge \bigwedge\{\neg x \mid (x, \mathsf{F}) \in s\}$. The controller has the output variables $O_Q = I_P$ and input variables $I_Q = O_P$. For every marked environment vertex $\alpha = \langle s, t' \rangle$ in the reduced micro-step graph, choose one marked successor $\langle s, t' \cup \{(x', b)\} \rangle$ of α, and add to Γ the guarded command $[\![\chi_s \wedge \chi_{t'} \rightarrow x' = b$. Like composability checking, the single-step unknown-type control problem for dependent-type modules is no harder than its

counterpart for dynamically typed modules.

Theorem 5.9 *The single-step unknown-type control problem for dependent-type modules is PSPACE-complete. The multi-step unknown-type control problem for dependent-type modules is EXP-complete. Moreover, if the answer is Yes, then a dependent-type controller can be synthesized.*

The fixed-type control problem is computationally harder than the unknown-type control problem, although it is no harder than its counterpart for dynamically typed modules. The additional complexity is due to the fact that we need to construct explicitly the micro-step graph.

Theorem 5.10 *The single-step fixed-type control problem for dependent-type modules is complete for NE. The mutli-step fixed-type control problem for dependent-type modules is complete for NE. Moreover, if the answer is Yes, then a dependent-type controller can be synthesized.*

Proof. To solve the fixed-type single-step control problem $((P, C_P), C_Q, s, \varphi)$, one can guess a subgraph G of the reduced micro-step graph $RMG_s(P, C_P)$, and check that (1) the vertex $\langle s, \emptyset \rangle$ is in G; (2) for all module vertices α in G, all successors of α are also in G; (3) for all environment vertices α in G, there is exactly one successor β of α in G such that (α, β) is an edge in G; (4) for all terminal vertices $\langle s, t' \rangle$, we have $\varphi[\![t]\!]$; (5) the controller respects the variable dependency relation C_Q at all environment vertices in G. All of these conditions can be checked in time polynomial in the size of G. NE-hardness comes from the fact that dependent-type modules can encode dynamically typed modules. ∎

Like dynamically-typed modules, a dependent-type module always has a most state-general controller, but may not have a most general controller.

Proposition 5.8 *For every dependent type module and every control objective, there is a most state-general single-step dep-controller, and there is a most state-general multi-step dep-controller. However, there is a dynamically typed module and a control objective such that there is no most general single-step dep-controller, nor a most general multi-step dep-controller.*

Example 5.12 The dynamically-typed module (P, D_P) in Example 5.11 can be viewed as a dependent-type module (P, C_P) such that $C_P = \{(\tau, x \succcurlyeq \bigcup\{y \mid x \succ^{\bullet} y\}) \mid x \in X_P\}$. Similarly for the controllers $(Q_i, D_{Q_i}), i \in \{1, 2\}$ can also be viewed as the dependent-type modules (Q_i, C_{Q_i}). However, there is no single dependent-type controller whose transition relation implies both $\tau_{Q_i}, i \in \{1, 2\}$. This is because even dependent-type modules do not allow disjunction of dependency relations. ∎

5.5.6 Unrestricted control

One may be inclined to define the following "unrestricted synchronous control problem": given a non-blocking module P, a state s of P, and a predicate φ over X_P, is there a module Q composable with P such that (1) the synchronous composition $P \| Q$ is non-blocking, and (2) the answer to the single-step (resp. multi-step) verification problem $(P \| Q, s, \varphi)$ is Yes. This formulation, however, makes no distinction between output and input variables, and thus permits the controller Q to arbitrarily constrain the output variables of P, as long as the compound system $P \| Q$ is non-blocking. Thus, the "unrestricted synchronous control problem" is not a control problem at all in the traditional sense, because it simply asks for the existence of a transition (in the single-step case) or run (in the multi-step case). The single-step solution amounts to evaluating the boolean Σ_1^p formula $(\exists X'_P)(\tau_P \wedge \varphi')[s]$, and like invariant verification, the multi-step problem is complete for PSPACE. Note that if the non-blocking requirement (1) is also dropped, then the appropriate single-step formula is $(\exists X'_P)(\tau_P \rightarrow \varphi')[s]$, which permits the controller to block the progress of P.

5.5.7 The relative power of controllers

Recall the module class ordering *async* < *moore* < *static* < *dsynt* < *dsem* < *dep*. The following proposition establishes that this ordering strictly orders the power of controllers.

Proposition 5.9 *For all module classes $\alpha, \beta \in \Lambda$ with $\alpha < \beta$, there is a module $P \in \mathcal{M}_\alpha$, a control objective φ over X_P, and a state s of P, such that s is not single-step (resp. multi-step) α-controllable with respect to φ, but s is single-step (resp. multi-step) β-controllable with respect to φ.*

Example 5.13 As an example showing that an *async*-controller is strictly less powerful than a *moore*-controller, consider the module P which has one output variable $O_P = \{x\}$ and one input variable $I_P = \{y\}$. The transition relation for P is $\tau_P = (x \oplus x')$. The single-step control objective is $\varphi = (x \oplus y)$. Clearly, the states $x = \mathrm{T}, y = \mathrm{F}$ and $x = \mathrm{F}, y = \mathrm{T}$ are not asynchronously controllable, since at most one module can make a transition at any step. However, these states can be output by the Moore controller Q such that $\tau_Q = (y' \equiv x)$. In fact, this controller controls every state of P.∎

Example 5.14 Consider the same control problem for module P in Example 5.13, except that the transition predicate for P is now $\tau_P = \mathrm{T}$. Clearly, no state of P can be output by a *moore*-controller. But there is a *static*-controller Q that controls every state of P: $\tau_Q = (y' \oplus x')$, with static type $y \succ x$.∎

Example 5.15 This example shows that a *static*-controller is less powerful than a *dsynt*-controller. Consider Example 5.6. Our single-step control objective to module Q is $\varphi = (x \equiv \neg z)$. Clearly no state of s can be controlled by a *static*-controller. But all states can be controlled by a *dsynt*-controller, namely P_3.∎

Example 5.16 Consider the single-step control problem in Example 5.10. Any *dsynt*-controller for module P can control either the states with $z \equiv \mathrm{F}$, or the states with $z \equiv \mathrm{T}$, but not both. However, every state can be controlled by the *dsem*-controller Q defined as $\tau_Q = (z \wedge (y_1' \equiv x_0')) \vee (\neg z \wedge (y_0' \equiv x_1'))$, and with dynamic type $D_Q = \{(z, y_1 \succ x_0), (\neg z, y_0 \succ x_1)\}$. ∎

Example 5.17 Let P be the dynamically-typed module with $I_P = \{u, v\}$ and $O_P = \{x\}$, and the following guarded commands:

$$(P, D_P) = \begin{array}{ll} [\!] & u' \rightarrow x' = \neg v' \\ [\!] & \neg u' \rightarrow x' = \mathrm{T} \\ [\!] & \neg u' \rightarrow x' = \mathrm{F} \end{array}$$

The control objective is $x = v$. There is no dynamically typed controller (at any state), because such a controller would have x dependent on v, and P can set x to be $\neg v$. But a dependent-type controller can change the dependencies, namely, having x not dependent on any variable and having v dependent on x. The following is a dependent-type controller:

$$(Q, C_Q) = \left\{ \begin{array}{lllll} [\!] \, \mathrm{T} & \rightarrow & u' & = & \mathrm{F} \\ & & & \wedge & \\ [\!] \, \mathrm{T} & \rightarrow & v' & = & x' \end{array} \right\}. ∎$$

Chapter 6

Synthesis of Uninitialized Systems

In *sequential synthesis*, we transform a temporal specification into a reactive system that is guaranteed to satisfy the specification. A *closed system* that meets the specification can be extracted from a model that satisfies the specification, that is, the synthesis of closed systems amounts to solving a satisfiability ($\exists$) problem [EC82]. However, as argued for transformational systems in [MW80], and for reactive systems in [ALW89, Dil89, PR89a], the synthesis of *open systems*, which interact with an unknown environment, requires the solution of a $\forall\exists$ problem: for all sequences of inputs, there exists a sequence of outputs that satisfies the specification. Consider, for example, a scheduler for a printer that serves two users. The scheduler is an open system. Each time unit it reads the input signals $I1$ and $I2$ (a job sent from the first or second user, respectively), and writes the output signals $P1$ and $P2$ (print a job of the first or second user, respectively). The scheduler should be designed so that jobs of the two users are not printed simultaneously, and whenever a user sends a job, the job is printed eventually. Of course, this should hold no matter how the users send jobs. We can specify the requirement for the scheduler in terms of a *linear temporal logic* (LTL) formula ψ [Pnu81], such as

$$\Box(I1 \rightarrow \bigcirc(\neg I1 \,\mathcal{U}\, P1)) \wedge \Box(I2 \rightarrow \bigcirc(\neg I2 \,\mathcal{U}\, P2)) \wedge \Box\neg(I1 \wedge I2).$$

Evidence of ψ's satisfiability (note that ψ is satisfied in a structure in which the four signals never occur) is not of much help in extracting a correct scheduler: while such evidence only suggests a scheduler that is guaranteed to satisfy ψ for *some* input sequence, we want a scheduler that satisfies ψ for *all* possible scripts of jobs sent to the printer.

We now make this intuition formal. A *stream transducer* is a function that, given

an infinite sequence of inputs, produces an infinite sequence of outputs. In particular, for the set I of inputs signals and the set O of output signals, a stream transducer is a function from $(2^I)^\omega$ to $(2^O)^\omega$. A *stream requirement* is a binary relation between input streams and output streams: that is, a stream requirement is a subset of $(2^I)^\omega \times (2^O)^\omega$ or, equivalently, a set of infinite words in $(2^{I \cup O})^\omega$. The stream transducer T *realizes* the stream requirement R iff for every input stream $\tau \in (2^I)^\omega$, we have $R(\tau, T(\tau))$. Stream requirements can be specified by LTL formulas over the set $I \cup O$ of atomic propositions, or by *automata on infinite words* over the alphabet $2^{I \cup O}$. Stream transducers can be implemented by state machines that proceed ad infinitum. The *finite-state implementation* of a stream transducer is a deterministic finite-state machine that, from a given state on a given set of input signals, generates a set of output signals and moves to a successor state. The *realizability problem* (RP) asks, given a stream requirement R, if there is a finite-state implementation of a stream transducer that realizes R. The *sequential synthesis problem*, then, is to find a finite-state implementation (if one exists). The RP was first stated by Church [Chu62] for stream requirements specified in the *sequential calculus*. Since then, several solutions for the RP have been studied: [BL69, Rab72] showed that the RP is quadratic (exponential) if the specification is a deterministic (nondeterministic) Büchi automaton; [PR89a] showed that the RP is doubly exponential if the specification is an LTL formula (researchers from control theory also studied the RP in the context of supervisory control for discrete-event systems [RW89]). The solutions to the RP can be extended, within the same complexity bounds, to construct finite-state implementations, so that a solution to the RP immediately provides a solution also to the sequential synthesis problem [Rab70, MS95, KV99].

In practice, sequential hardware is often designed to operate without prior initialization: that is, it is supposed to satisfy its input-output requirements if started from any state. *Uninitialized state machines*, which model such hardware designs, require no reset circuitry and therefore have an advantage of smaller area. A well-known example of an uninitialized state machine is the IEEE 1149.1 standard for boundary-scan test [Com90]. Uninitialized state machines are also necessary for the *safe replaceability* of sequential circuits [SP94], where a state machine is replaced by another one in such a way that the surrounding environment is unable to detect the changes. The replacing state machine may power-up in an arbitrary state, and is therefore uninitialized. The verification problem of deciding whether an uninitialized state machine safely replaces another machine, is studied in [SP94]. The optimization problem for uninitialized state machines is studied in [QBSP96].

In this dissertation, we study the synthesis problem for uninitialized state machines.

Given a stream requirement R, the *uninitialized realizability problem* (URP) asks if there is a finite-state implementation M that satisfies R no matter what the initial state of M is. The *uninitialized synthesis problem*, then, is to find such an M (if one exists). We study the URP for stream requirements that are specified by LTL formulas or Büchi automata. We consider deterministic and nondeterministic Büchi automata, as well as universal Büchi automata, which accept a word iff all runs are accepting, and alternating Büchi automata, which allow both nondeterministic and universal branching modes. The solution of the URP is quite straightforward, and is done by a reduction to the RP: if the stream requirement R is an LTL formula, then the URP for R can be reduced to the RP for the LTL formula $always(R) = \Box R$: if R is a Büchi automaton, then the URP for R can be reduced to the RP for the automaton $always(R)$, which is obtained from R by adding a universal self-loop at the initial state. It is not hard to see that an infinite word w (i.e., a pair of input and output streams) satisfies $always(R)$ iff w and all its suffixes satisfy R. This implies that R is realizable by an uninitialized implementation iff $always(R)$ is realizable. As in the initialized case, a solution to the uninitialized synthesis problem follows immediately from a solution to the URP.

While the above solution is straightforward, it may lead to upper bounds that are exponentially worse than the complexity of the RP for the corresponding formalism. For example, while for LTL specifications both RP and URP are doubly exponential, for deterministic Büchi automata, where the RP is quadratic, the presented solution of the URP is exponential. The reason is that the automaton $always(R)$ has a universal branching mode, which R may not have, and this makes the URP exponentially harder. In particular, if R is a deterministic automaton, then $always(R)$ is universal, and if R is nondeterministic, then $always(R)$ is alternating. Can the exponential blow-up be avoided by a more sophisticated solution? We answer this question in the negative by proving corresponding lower bounds for the URP of all discussed formalisms. Unlike the upper bounds, the lower-bound proofs are not immediate, and are the main technical contributions of this work. Our results imply that specification formalisms that support an easy implementation of the *always* operator, such as LTL and alternating automata, have, unlike deterministic and nondeterministic automata, already "built-in" the complexity of uninitialized synthesis.

We say that a stream requirement R is *uninitialized* if for all infinite words w, we have w satisfies R iff all suffixes of w satisfy R. This is the same as asking if the two

formulas, or automata. R and $always(R)$ are equivalent. For example, the LTL specification $\Box p$ for an output signal p is uninitialized, as w satisfies $\Box p$ iff all the suffixes of w satisfy $\Box p$. For uninitialized stream requirements, the URP coincides with the RP. As the equivalence problem is easier than the corresponding URP in all cases, it follows that for specification formalisms whose URP is harder than RP, there is an advantage to first checking if the specification is uninitialized. In the final section, we show that the *uninitialized specification problem* (USP), which asks if a given stream requirement R is uninitialized, is, for all considered formalisms, no easier than the general equivalence problem.

6.1 Preliminaries

Trees. Given a finite set D of directions, a D-tree is a set $T \subseteq D^*$ such that if $x \cdot d \in T$, where $x \in D^*$ and $d \in D$, then also $x \in T$. The elements of T are called *nodes*, and the empty word ϵ is the *root* of T. For every $x \in T$, the nodes $x \cdot d \in T$, for $d \in D$, are the *successors* of x. Each node $x \in T$ has a direction $dir(x)$ in D, namely, $dir(\epsilon) = d^0$ for some designated $d^0 \in D$, and $dir(x \cdot d) = d$. A path π of the tree T is a set $\pi \subseteq T$ such that $\epsilon \in \pi$, and for every $x \in \pi$, exactly one successor of x is in π. Given two finite sets D and Σ, a Σ-labeled D-tree is a pair (T, V) where T is a D-tree, and $V : T \to \Sigma$ maps each node of T to a letter in Σ. We extend V to paths in the straightforward way: for a path $\pi = \{\epsilon, w_0, w_0w_1, \ldots\}$, we have $V(\pi) = V(\epsilon)V(w_0)V(w_0w_1)\ldots$ We say that a $(D \times \Sigma)$-labeled D-tree (T, V) is *D-exhaustive* if $T = D^*$, and for every node $w \in D^*$, we have $V(w) = (dir(w), \sigma)$ for some $\sigma \in \Sigma$.

Alternating Büchi Automata. For a given finite set X, let $\mathcal{B}^+(X)$ be the set of positive boolean formulas over X. A subset $Y \subseteq X$ satisfies a formula $\theta \in \mathcal{B}^+(X)$ if the truth assignment that assigns *true* to the members of Y and assigns *false* to the members of $X \setminus Y$ satisfies θ. An *alternating Büchi automaton* $\mathcal{U} = (\Sigma, U, u_0, \delta, F)$ consists of an alphabet Σ, a set U of states, an initial state $u_0 \in U$, a transition function $\delta : U \times \Sigma \to \mathcal{B}^+(U)$, and a set $F \subseteq U$ of accepting states. A *run* of $\mathcal{U}$ on an infinite word $w = w_0w_1\ldots$ in Σ^ω is an infinite U-labeled D-tree (T, r), where $D = \{1, \ldots, |U|\}$, such that $r(\epsilon) = u_0$ and the following holds: for all nodes $x \in T$, if $|x| = i$ and $r(x) = u$ and $\delta(u, w_i) = \theta$, then x has k successors $x_1, \ldots, x_k$, for some $k \leq |U|$, and $\{r(x_1), \ldots, r(x_k)\}$ satisfies θ. A run (T, r) is *accepting* if every path of (T, r) visits the accepting set F infinitely often. An infinite word w is accepted by $\mathcal{U}$ if there exists a run (T, r) on w such that (T, r) is accepting. The

language $L(\mathcal{U})$ of $\mathcal{U}$ is the set of infinite words accepted by $\mathcal{U}$.

Finite State Machines. A *finite state machine* (FSM) $M = (I, O, Q, q_{in}, \rho, \lambda)$ consists of a finite set I of input signals, a finite set O of output signals, a finite state set Q, an initial state $q_{in} \in Q$, a transition function $\rho : Q \times 2^I \to Q$ and an output function $\lambda : Q \to 2^O$. We assume that there is a special output signal *init* such that *init* $\in \lambda(q)$ iff $q = q_{in}$. We also assume that there is a nonempty set $In(q_{in}) \subseteq 2^I$ such that for each $i_0 \in In(q_{in})$, there is a $q \in Q$ such that $\rho(q, i_0) = q_{in}$; that is, the state q_{in} is reachable via some input (if this is not the case, we can add a new state from which q_{in} is reachable). An FSM M interacts with its environment through its input and output signals. Initially, M is at the initial state $q_0 = q_{in}$. The environment initiates the interaction by inputting some input $i_0 \in In(q_{in})$. Then, M starts operating by outputting $\lambda(q_0)$, to which the the environment replies with some $i_1 \in 2^I$. The FSM M replies by moving to the state $q_1 = \rho(q_0, i_1)$ and outputting $\lambda(q_1)$. Interaction then continues ad infinitum.

Hence, the FSM M can be viewed as a *strategy* $S_M : (2^I)^* \to 2^O$ that maps every finite sequence of inputs to an output. To define S_M formally, we first define the function $C_M : (2^I)^* \to Q$ that maps each finite input sequence to the state visited after the sequence has been read: $C_M(\epsilon) = q_{in}$, and $C_M(i_1 \dots i_n) = \rho(C_M(i_1 \dots i_{n-1}), i_n)$. The strategy S_M induced by M is then defined for every $w \in (2^I)^*$ by $S_M(w) = \lambda(C_M(w))$. Note that the first input i_0 merely initiates the interaction and does not have any effect on the behavior of M: it is disregarded in the definition of C_M. Each infinite sequence $i_0 i_1 \dots \in In(q_{in}) \cdot (2^I)^\omega$ induces a *computation* $(i_0, S_M(\epsilon))(i_1, S_M(i_1))(i_2, S_M(i_1 i_2)) \dots \in (2^I \times 2^O)^\omega$ of M. The *language* $L(M)$ is the set of all computations of M. We refer to the language also as a set of infinite words in $(2^{I \cup O})^\omega$, where $i_0 i_1 \dots$ induces the computation $(i_0 \cup S_M(\epsilon)) \cdot (i_1 \cup S_M(i_1)) \cdot (i_2 \cup S_M(i_1 i_2)) \dots \in (2^{I \cup O})^\omega$. The strategy S_M induces, for a given first input $i_0 \in In(q_{in})$, a *computation tree* whose branches correspond to external nondeterminism caused by different inputs, namely, the 2^I-exhaustive $(2^I \times 2^O)$-labeled 2^I-tree $((2^I)^*, V)$ such that each node $w \in (2^I \times 2^O)^*$ is labeled by $V(w) = (dir(w), S_M(w))$, where $dir(\epsilon) = i_0$. Note that all computation trees of M differ only in the first input.

6.2 The Uninitialized Realizability Problem

In this section we define and solve the uninitialized realizability problem. We first start with the (initialized) realizability problem. Given a specification R over the input

signals I and output signals O the *Realizability Problem* (RP) for R asks if there is an FSM M such that for all the words $w \in L(M)$, we have $w \models R$. If so, we say that R is realizable by M. The specification R can be an LTL formula, or a finite-state Büchi automaton. If R is an LTL formula, then the atomic propositions of R are $I \cup O$, and the relation $\models$ is the usual satisfiability relation. If R is an automaton, then the alphabet of R is $2^I \times 2^O$, and the relation $\models$ is the language membership relation, that is, $w \models R$ iff $w \in L(R)$. The realizability problem is closely related to *Church's solvability problem* [Chu62], and it has been shown that the problem is solvable in quadratic (exponential) time if R is a deterministic (nondeterministic) Büchi automaton [BL69, Rab72, Saf88, PR89a], and in doubly exponential time if R is an LTL formula [PR89a, KV99].

An *uninitialized FSM* $M = (I, O, Q, \rho, \lambda)$ is similar to an FSM except that there is no initial state. The language $L(M) = \bigcup_{q \in Q} L(M_q)$ of M is simply the union of the languages $L(M_q)$, where $M_q = (I, O, Q, q, \rho, \lambda)$ is the FSM obtained from M by regarding the state $q \in Q$ as the initial state. Given a specification R over the input signals I and output signals O, the *uninitialized realizability problem* (URP) for R asks if there is an uninitialized FSM M such that $w \models R$ for all words $w \in L(M)$. If the answer is yes, we say that R is *uninitialized realizable* by M.

6.2.1 Reducing URP to RP

We solve the URP by reducing it to the RP. For that, we define, given a specification R over the input signals I and output signals O, the specification $always(R)$ over I and O such that, for all words $w \in (2^I \times 2^O)^\omega$, we have w satisfies $always(R)$ iff all the suffixes w' of w satisfy R. It is not hard to see that the URP for R can be reduced to the RP for $always(R)$, as stated in the Theorem below.

Theorem 6.1 *Let I and O be respectively finite sets of input and output signals, and let R be a specification over I and O. Then R is uninitially realizable iff $always(R)$ is realizable.*

Proof. Assume first that R is uninitially realizable. Let $M = (I, O, Q, \rho, \lambda)$ be an uninitialized FSM that uninitially realizes R. Consider an FSM $M_q = (I, O, Q, q, \rho, \lambda)$, for some $q \in Q$. We show that M_q realizes $always(R)$. Consider an input sequence $\pi = i_0, i_1, i_2 \ldots \in In(q) \cdot (2^I)^\omega$ to M_q. The sequence induces the computation $\pi = (i_0, S_{M_q}(\epsilon)), (i_1, S_{M_q}(i_1)), (i_2, S_{M_q}(i_1 i_2)), \ldots$. For each suffix $\pi^j = (i_j, S_{M_q}(i_1 \cdots i_j)), (i_{j+1}, S_{M_q}(i_1 \cdots i$

of π, the state $q_j = C_{M_q}(i_0, \ldots, i_{j-1})$ is such that π^j is a computation of M_{q_j}. Hence, since M uninitially realizes R, it must be that π^j satisfies R. It follows that all the suffixes of all the computations of M_q satisfy R, thus M_q realizes $always(R)$.

Assume now that $always(R)$ is realizable. Let $M_{q_{in}} = (I, O, Q, q_{in}, \rho, \lambda)$ be the finite state machine that realizes $always(R)$. We can assume that every state $q \in Q$ is reachable from the initial state q_{in} (otherwise, we restrict $M_{q_{in}}$ to its reachable part). It is not hard to see that the uninitialized FSM $M = (I, O, Q, \rho, \lambda)$ realizes R. Indeed, since every word $w \in L(M_q)$, for any $q \in Q$, is a suffix of some word $w' \in L(M_{q_{in}})$, then, by the hypothesis, $w \models R$. ∎

6.2.2 Constructing $always(\mathbf{R})$

Given a specification R, we construct the specification $always(R)$. First, if R is an LTL formula, it is not hard to see that $always(R) = \Box R$. Now, if R is a Büchi automaton $\mathcal{U} = (\Sigma, U, u_0, \delta, F)$, then $always(\mathcal{U}) = (\Sigma, U \cup \{u_0'\}, u_0', \delta', F \cup \{u_0'\})$, where u_0' is a new state, and for all $\sigma \in \Sigma$, we have $\delta'(u_0', \sigma) = \delta(u_0, \sigma) \wedge u_0'$; and for all $u \in U$, we have $\delta'(u, \sigma) = \delta(u, \sigma)$. Intuitively, the automaton $always(\mathcal{U})$ behaves like $\mathcal{U}$ except that $always(\mathcal{U})$ always sends a copy of itself to the suffix of the input word whenever it makes a transition. It follows that for every word w, not only w has to be accepted by $\mathcal{U}$, but so do all its suffixes. Note that one copy of $always(\mathcal{U})$ keeps visiting u_0' forever, which is why we had to duplicate the original initial state. Formally, we have the following.

Proposition 6.1 *Let $\mathcal{U} = (\Sigma, U, u_0, \delta, F)$ be an alternating Büchi automaton. For all words $w \in \Sigma^\omega$, the automaton $always(\mathcal{U})$ accepts w iff $\mathcal{U}$ accepts all the suffixes of w.*

Proof. Consider a D-tree T. For a node $x \in T$, let $level(x)$ be the level of the node, where $level(\epsilon) = 0$ and if the node y is a successor of x, then $level(y) = level(x) + 1$. A subtree (T_x, r_x) of a Σ-labeled D-tree (T, r) with root $x \in T$, is a tree such that $y \in T_x$ iff $x \cdot y \in T$, and $r_x(y) = r(x \cdot y)$.

Consider an infinite word $w = w_0 w_1 \ldots \in (2^I \times 2^O)^\omega$ in $L(always(\mathcal{U}))$. Let (T, r) be an accepting run of $\mathcal{U}$ on w. By the definition of $always(\mathcal{U})$, each level of (T, r) has one node x with $r(x) = u_0'$. Given a suffix $w' = w_i w_{i+1} \ldots$ of w, let x be such that $level(x) = i$ and $r(x) = u_0'$, and let (T_x, r_x) be the subtree of (T, r) with root x. We can obtain from (T_x, r_x) an accepting run of $\mathcal{U}$ on the suffix $w' = w_i w_{i+1} \ldots$ of w by replacing the label of

the root node with u_0, and by pruning the tree in such a way that every node has exactly all the successors y with $r_x(y) \neq u'_0$. Hence, if $always(\mathcal{U})$ accepts w, then $\mathcal{U}$ accepts all suffixes of w.

For the other direction, assume that $\mathcal{U}$ accepts all the suffixes of w. Let (T_i, r_i) be the accepting run tree of $\mathcal{U}$ on $w' = w_i w_{i+1} \ldots$, with the label at the root node replaced by u'_0, i.e., $r_i(\epsilon) = u'_0$. We successively build an accepting run (T, r) of $always(\mathcal{U})$ on w. Let $(T^0, r^0) = (T_0, r_0)$. At each node $x \in T^i$, for $i \geq 0$, if $r^i(x) = u'_0$, then we add to x as successor the subtree (T_j, r_j), for $j = level(x) + 1$. We claim that the tree $(T, R) = \bigcup_{i \geq 0}(T^i, r^i)$ is a legal accepting run of $always(\mathcal{U})$ on w. First, since each node labeled by u'_0 has a successor labeled by u'_0, the run is legal. Then, since each path of (T, R) either eventually corresponds to a path of (T_j, r_j), for some j, or visits u'_0 forever, the run is also accepting. ∎

6.2.3 URP Complexity

As described in Section 6.1, we can now solve the URP for R by solving the RP for $always(R)$. The complexity of this naive solution depends on the type of the specification $always(R)$. In Table 6.1 below we describe the type of $always(R)$ given the type of R. It

R	$always(R)$
an LTL formula	an LTL formula
a deterministic or universal Büchi automaton	a universal Büchi automaton
a nondeterministic or alternating Büchi automaton	an alternating Büchi automaton

Table 6.1: The cost of moving from R to $always(R)$.

follows that when R is a deterministic or a nondeterministic Büchi automaton, the type of $always(R)$ is richer than that of R, which in turn would imply that the URP is harder than the RP. In this section we analyze the complexity of the URP and then show that this naive solution is optimal.

Theorem 6.2 *The URP for LTL is complete for 2EXP.*

Hence for LTL specifications, the URP is no harder than RP.

Proof. The upper bound follows from the fact that the RP for LTL is in EXP [PR89a], and $always(R)$, for R in LTL, is also in LTL. For the lower bound, we show that the URP

is at least as hard as the RP, which is 2EXP-hard [Ros92]. Indeed, the RP for an LTL formula φ can be reduced to the URP for $init \rightarrow \varphi$. ∎

We now turn to consider the various types of Büchi automata. While the upper bounds are easy, the lower bounds require complicated generic reductions. To illustrate the proof ideas, we first consider the URP for *closed* FSMs, in which $I = \emptyset$. The behavior of such FSMs is independent of the environment, and the realizability and satisfiability problems coincide. In particular, for specifications given in term of deterministic Büchi automata, the realizability problem for FSMs with $I = \emptyset$ is complete for NLOGSPACE. We show that the transition to uninitialized FSM makes the problem exponentially harder.

Proposition 6.2 *The URP for deterministic Büchi automata and closed FSMs is complete for PSPACE.*

An automata-theoretic problem that is well-known to be complete for PSPACE is the universality problem for *nondeterministic* automata [HU79, Wol82]. On the other hand, the universality problem for *deterministic* automata is complete for NLOGSPACE. The standard PSPACE lower bound proof is by reduction from the membership problem for polynomial space Turing machines: given a polynomial space Turing machine T, one constructs a nondeterministic Büchi automaton $\mathcal{U}$ such that $\mathcal{U}$ accepts invalid or rejecting computations of T. The URP as stated in Proposition 6.2 also has some flavor of "universality". It comes from the requirement that the model w (i.e., an infinite word) has to be "suffix-closed", i.e., both w and all its suffixes need to satisfy the specification. However, the lower bound proof is very different. The proof is by the construction of a *deterministic* Büchi automaton which accepts an input word w as well as all its suffixes iff w is a *valid* and *accepting* computation of T. Since the universality problem of deterministic Büchi automata is easy, the "suffix-closure" property is as strong a requirement as the universality requirement of a nondeterministic Büchi automaton.

Proof of Proposition 6.2. Consider a deterministic Büchi automaton R. When $I = \emptyset$, the set 2^I is a singleton and the universal Büchi automaton $always(R)$ is realizable iff its language is not empty. Since the latter can be checked in PSPACE [MH84, VW94], the upper bound follows.

For the lower bound, we do a reduction from the membership problem of a polynomial space Turing machine. Let $T = (\Gamma, Q, \mapsto, q_0, F_{acc}, F_{rej})$ be a polynomial space Turing

machine, where Q is the set of states, q_0 is the initial state, $F_{acc} \subseteq Q$ and $F_{rej} \subseteq Q$ are respectively accepting and rejecting states, and $\mapsto: Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the transition function. Assume that T starts with the initial configuration, i.e., T at state q_0 with its reading head pointing at the leftmost cell of the empty tape. We also assume that once T reaches an accepting configuration, i.e., $q \in F_{acc}$, it "cleans" the tape content and restarts from the initial configuration. The machine T accepts the empty tape iff T has an infinite computation visiting the initial and accepting configurations infinitely often.

Assume T uses $s(n)$ tape cells to process an input of length n. We encode a configuration of T by a string $\sharp\gamma_1\gamma_2 \ldots (q, \gamma_i) \ldots \gamma_{s(n)}$ in $(2^O)^*$, where subsets of the output signals O are selected to encode the the alphabets $\Gamma \cup (Q \times \Gamma) \cup \{\sharp\}$, i.e., $2^O = \Gamma \cup (Q \times \Gamma) \cup \{\sharp\}$. That is, a configuration starts with the letter $\sharp$, followed by a string of letters $\gamma_j \in \Gamma$, except for one in $Q \times \Gamma$. The meaning of the string is that γ_j is the letter on the j-th tape cell, while the letter (q, γ_i) indicates in addition that T is at state q with its reading head pointing at the i-th tape cell. Let $c = \sharp\sigma_1\sigma_2 \ldots \sigma_{s(n)}$, and $c' = \sharp\sigma'_1\sigma'_2 \ldots \sigma'_{s(n)}$ be two configurations. If c' is the successor of c, then we know by the transition function of T what σ'_i for $1 \leq i \leq s(n)$ should be. Let $next(\sigma_{i-1}, \sigma_i, \sigma_{i+1})$ denote our expectations for σ'_i. That is,

- $next(\gamma_{i-1}, \gamma_i, \gamma_{i+1}) = next(\sharp, \gamma_i, \gamma_{i+1}) = next(\gamma_{i-1}\gamma_i, \sharp) = \gamma_i$.
- $next((q, \gamma_{i-1}), \gamma_i, \gamma_{i+1}) = next((q, \gamma_{i-1}), \gamma_i, \sharp) = \begin{cases} \gamma_i & \text{if } (q, \gamma_{i-1}) \mapsto (q', \gamma'_{i-1}, L) \\ (q, \gamma_i) & \text{if } (q, \gamma_{i-1}) \mapsto (q', \gamma'_{i-1}, R). \end{cases}$
- $next(\gamma_{i-1}, (q, \gamma_i), \gamma_{i+1}) = next(\sharp, (q, \gamma_i), \gamma_{i+1}) = next(\gamma_{i-1}, (q, \gamma_i), \sharp) = \gamma'_i$, where $(q, \gamma_i) \mapsto (q', \gamma_{i-1}, \Delta)$.
- $next(\gamma_{i-1}, \gamma_i, (q, \gamma_{i+1})) = next(\sharp, \gamma_i, (q, \gamma_{i+1})) = \begin{cases} \gamma_i & \text{if } (q, \gamma_{i+1}) \mapsto (q', \gamma'_{i-1}, L) \\ (q, \gamma_i) & \text{if } (q, \gamma_{i+1}) \mapsto (q', \gamma'_{i-1}, R). \end{cases}$
- $next(\sigma_{s(n)}, \sharp, \sigma'_1) = \sharp$.

We define a deterministic Büchi automaton $\mathcal{U} = (U, 2^O, u_0, \delta, F)$ such that $\mathcal{U}$ accepts an input word $w = w_0 w_1 \ldots$ iff w satisfies the following conditions: (1) the *next* relation of T is satisfied for the first three letters in w, i.e., $w_{s(n)+2} = next(w_0, w_1, w_2)$; and (2) the initial and the accepting configurations are eventually reached. It follows that $\mathcal{U}$ accepts an infinite word w as well as all its suffixes iff T has an infinite computation visiting the initial and accepting configuration infinitely often. Both conditions can be specified by a deterministic Büchi automaton of size polynomial in T.

Thus, if there exists a word w such that w and all its suffixes are accepted by $\mathcal{U}$, then there exists a suffix w' of w such that w' encodes an accepting run of T. On the other hand, if T has an accepting run, then it can be encoded as an infinite string $w \in \Sigma_0^*$ all of whose suffixes (including w itself) are accepted by $\mathcal{U}$. ∎

We now consider the URP for open FSMs, i.e., when $I \neq \emptyset$. While the RP for deterministic Büchi automata is quadratic, the following theorem states that the URP for open FSMs is harder than that for closed FSMs and the RP for open FSMs.

Theorem 6.3 *The URP for deterministic or universal Büchi automata is complete for EXP.*

Proof. Consider a deterministic or a universal Büchi automaton $\mathcal{U}$. The automaton $always(\mathcal{U})$ is a universal automaton, whose RP can be solved in EXP [Rab72, MS95].

For the lower bound, we use the input signals in I in order to encode branches and extend the proof of Theorem 6.2 to apply to alternating Turing machines. Consider an alternating linear-space Turing machine $T = (\Gamma, Q_u, Q_e, \mapsto, q_0, F_{acc}, F_{rej})$, where the disjoint sets of states Q_u and Q_e are respectively the universal and existential states, while the disjoint sets of states $F_{acc} \subseteq Q_e$ and $F_{rej} \subseteq Q_e$ are respectively the accepting and rejecting states. Their union is denoted by Q. Our model of alternation prescribes that $\mapsto \subseteq Q \times \Gamma \times Q \times \Gamma \times \{L, R\}$ has a binary branching degree. When a universal or an existential state of T branches into two states, we distinguish between the left and the right branches. Accordingly, we use $(q, a) \mapsto^l (q_l, b_l, \Delta_l)$ and $(q, a) \mapsto^r (q_r, b_r, \Delta_r)$ to indicate that when T is in state $q \in Q_u \cup Q_e$ reading input symbol a, it branches to the left with (q_l, b_l, Δ_l) and to the right with (q_r, b_r, Δ_r). We also assume that once T reaches an accepting configuration, it "cleans" the tape content and restarts from the initial configuration (i.e., empty tape and initial state at the left end of the tape).

Assume T uses $s(n)$ cells in its working tape in order to process an input of length n. A configuration of T is encoded in a similar way to how a configuration of a polynomial space Turing machine is encoded, except that a configuration starts with either $\sharp_l$ or $\sharp_r$. The letter $\sharp \in \{\sharp_l, \sharp_r\}$ marks the beginning of a configuration; moreover, since T has an existential mode, i.e., when the state q of T is in Q_e, the letter $\sharp$ also indicates a guess (left or right) for the accepting successor. A computation of T can then be encoded by a computation tree whose branches describe sequences of configurations of T. Note that the computation tree is unique if we ignore the distinction between $\sharp_l$ and $\sharp_r$. A run of T is a

pruning of a computation tree in which all the universal configurations have both successors and all the existential configurations $c = \sharp\gamma_1\gamma_2\ldots(q,\gamma_i)\ldots\gamma_{s(n)}$ have only the left (resp. right) successor if $\sharp = \sharp_l$ (resp. $\sharp_r$). The run is accepting if all branches in the pruned tree visit the initial and accepting configuration infinitely often.

Given an alternating linear-space Turing machine T as above, we construct a deterministic Buchi word automaton $\mathcal{U}$ such that $\mathcal{U}$ is uninitially realizable iff T has an accepting run on the empty tape (clearly, proving a lower bound for deterministic automata, implies a bound also for universal ones). The automaton $\mathcal{U}$ has input signals I such that the subsets of I encode the set $\{l,r\}$, i.e., $2^I = \{l,r\}$. It also has output signals O such that $2^O = \{\sharp_l,\sharp_r\}\cup\Gamma\cup(Q\times\Gamma)$. Let $c = \sharp\sigma_1\sigma_2\ldots\sigma_{s(n)}$ and $c' = \sharp'\sigma'_1\sigma'_2\ldots\sigma'_{s(n)}$ be two configurations, and let $(d_0,\sharp)(d_1,\sigma_1)\ldots(d_{s(n)},\sigma_{s(n)})(d'_0,\sharp')(d'_1,\sigma'_1)\ldots(d'_{s(n)},\sigma'_{s(n)})$ be a word in $(2^I\times 2^O)^*$. The letter d'_0 indicates the direction of c' with respect to c: if $d'_0 = l$, then c' is the left successor of c, and if $d'_0 = r$, then c' is the right successor of c. Note that the direction of c' is given by d'_0, not by the letter $\sharp$ or $\sharp'$: the letter $\sharp$ is only the guess that T makes at c if c is an existential configuration. If $\sharp = \sharp_l$ (resp. $\sharp_r$), then T guesses that the left (resp. right) successor of c leads to an accepting run: if the guess of T is different from the successor information given by the input, we say that there is a mismatch between the input and the guess of T at the configuration. That is, a mismatch happens at c with $\sharp = \sharp_r$ and $d'_0 = l$, as well as with $\sharp = \sharp_l$ and $d'_0 = r$. Recall that we require every path of the computation tree of T to be legal and accepting. On the other hand, since T is alternating, only the paths in the computation tree that are guessed in existential configurations need to be accepting. We use $\sharp_l$ and $\sharp_r$ in order to detect mismatches, where paths that contain a mismatch are considered accepting.

If the configuration c' is a successor of the configuration c, we know by the transition relation of T what the "next" relation is. Now we have two "next" relations, one for left branching and one for right branching. Let $next^l$ and $next^r$ be the "next" relations for the left branch and right branch respectively. The definition of $next^l$ (resp. $next^r$)is similar to that of the *next* relation in the polynomial space Turing machine case, except that only the transition function $\mapsto^l$ (resp. $\mapsto^r$) is considered, the letter $\sharp$ is in $\{\sharp_l,\sharp_r\}$, and $next^l(\sigma_{s(n)},\sharp,\sigma'_1)\in\{\sharp_l,\sharp_r\}$.

The automaton $\mathcal{U}$ can be constructed as follows. On input of a word $w = (d_0,\sigma_0)(d_1,\sigma_1)\ldots$, $\mathcal{U}$ checks the following:

1. The "next" transition relations of T are satisfied, i.e., $\sigma_{s(n)+2} = next^l(\sigma_0, \sigma_1, \sigma_2)$ if $d'_0 = l$, and $\sigma_{s(n)+2} = next^r(\sigma_0, \sigma_1, \sigma_2)$ if $d'_0 = r$; and

2. either of the following is true:

   (a) Eventually there is a mismatch in the direction specified by the input and T at an existential configuration, i.e., w contains the string $(d_0, \natural_0) \ldots (d_j, (q, \gamma_j)) \ldots (d_{s(n)}, \sigma_{s(n)})(d'_0, \natural'_0)$, where $q \in Q_e$ and either $\natural_0 = \natural_r$ and $d'_0 = l$, or $\natural_0 = \natural_l$ and $d'_0 = r$.

   (b) The initial configuration is eventually reached, and thereafter the accepting configuration is also eventually reached.

All the above conditions can be specified by a deterministic Büchi word automaton linear in the size of T.

Given a path w of a (2^I)-exhaustive $(2^I \times 2^O)$-labeled 2^I-tree $((2^I)^*, \tau)$, if $\mathcal{U}$ accepts w and all the suffixes of w, then by condition (2), w is a valid branch of the computation tree of T; moreover, by condition (3), if w is a branch guessed by T, then infinitely often the initial and accepting configurations are reached. Thus, $\mathcal{U}$ accepts all suffixes of all branches of $((2^I)^*, \tau)$ iff T has an accepting run. ∎

The RP for nondeterministic Büchi automata can be solved in exponential time, while the RP for alternating Büchi automata requires doubly exponential time. The following theorem shows that the URP for nondeterministic Büchi automata is exponentially harder than the RP, while that for alternating Büchi automata, there is no additional cost.

Theorem 6.4 *The URP for nondeterministic or alternating Büchi automata is complete for 2EXP.*

Proof. Consider a nondeterministic or an alternating Büchi automaton $\mathcal{U}$. The automaton $always(\mathcal{U})$ is an alternating automaton, whose RP can be solved in 2EXP [Rab72, MS95].

The proof of the lower bound is similar to that for the URP for deterministic Büchi automata in Theorem 6.3, except that now the reduction is from the membership problem of an alternating exponential space Turing machine.

Consider a Turing machine $T = (\Gamma, Q_u, Q_e, \mapsto, q_0, F_{acc}, F_{rej})$ that uses $s(n) = 2^{n^k} - 1$ tape cells, for some $k \geq 1$. We construct a nondeterministic Büchi automaton $\mathcal{V}$ such that $\mathcal{V}$ is uninitially realizable iff T has an accepting run. We first define the encoding

of a configuration. A configuration of T consists of a *block* $p_1 \dots p_{n^k}\sigma$ of the form $\{0,1\}^{n^k} \cdot \Sigma$, for $\Sigma = \Gamma \cup (Q \times \Gamma) \cup \{\sharp_l, \sharp_r\}$. The meaning of a block is that the j-th tape cell, where j is represented in binary as $p_1 \dots p_{n^k}$ with the bit p_{n^k} being the least significant bit, contains the letter σ. For example, the block 011γ encodes the fact that the third tape cell contains the letter γ. A configuration is therefore a sequence $B_0 B_1 \dots B_{s(n)}$ of 2^{n^k} blocks, where B_i encodes the content of the i-th tape cells. The first block B_0 of a configuration is either $0 \dots 0\sharp_l$ or $0 \dots 0\sharp_r$, indicating the beginning of a configuration as well as a guess of the accepting successor, similar to the proof in Theorem 6.3.

Let $2^O = \{0,1\} \cup \Sigma$. Given a string $w \in (2^O)^*$, we construct an automaton $\mathcal{V}_1$ such that $\mathcal{V}_1$ accepts w and all its suffixes iff w is a suffix of a valid sequence of configurations. This amounts to checking that

1. w starts with a suffix of a block and is followed by a sequence of blocks;
2. the blocks that encode position 0 of the tape, and only those blocks, encode the beginning of a configuration. That is, for every block $B = p_1 \dots p_{n^k}\sigma$, we have $p_i = 0$ for all i iff $\sigma \in \{\sharp_l, \sharp_r\}$; and
3. within a configuration, consecutive blocks encode the contents of consecutive tape positions.

Conditions (1) and (2) are easily checked by a linear size deterministic Büchi automaton. To check condition (3), note that given any binary numbers $p = p_0 \dots p_n$ and $p' = p + 1 = p'_0 \dots p'_n$, the bit $p_i \neq p'_i$ iff $p_{i+1} = \dots = p_n = 1$. Therefore, the automaton $\mathcal{V}_1$ can check that if $w_0 \neq w_{n^k+1}$ iff there exists $i, 0 \leq i \leq n^k$ such that $w_i \in \Sigma$ and $w_1 = \dots = w_{i-1} = 1$.

Now we construct an automaton $\mathcal{V}_2$ to check that successive configurations are valid transitions of the Turing machine T. For discussion, consider only the left branching. Consider three consecutive blocks $B_i = p_1^i \dots p_{n^k}^i \sigma_i$, $0 \leq i \leq 2$, as well as a block $B = q_1 \dots q_{n^k}\sigma$ in the next configuration. The block B encodes the same tape cell as B_1 iff $p_i^1 = q_i$ for all $1 \leq i \leq n^k$. In this case, it should be that $\sigma_1 = next^l(\sigma_0, \sigma_1, \sigma_2)$, where $next^l$ is the "next" relation defined in the proof of Theorem 6.3. We can use nondeterminism to do this check: the automaton $\mathcal{V}_2$ guesses a bit p_i^1 in block B_1, and check that for each block B in the next configuration, either $q_i \neq p_i^1$ or $\sigma_i = next^l(\sigma_0, \sigma_1, \sigma_2)$. Note that $\mathcal{V}_2$ has size polynomial in n.

Now we can complete the proof in analogy to the proof of Theorem 6.3. Let $2^I = \{l, r\}$, and let $w = (d_0, w_0)(d_1, w_1)\ldots$ be a string in $(2^I \times 2^O)^\omega$. The automaton $\mathcal{V}$ checks the following:

1. The string $w_0 w_1 \ldots$ is accepted by the automata $\mathcal{V}_1$ and $\mathcal{V}_2$; and
2. either of the following is true:
   (a) Eventually there is a mismatch in the direction specified by the input and T at an existential configuration, i.e., w contains the string $(d_0, \sharp)\ldots(d_l, \sigma)(d'_0, \sharp')\ldots(d'_l, \sigma')$ where $d_i \in 2^I$, $c = \sharp\ldots\sigma$ and $c' = \sharp'\ldots\sigma'$ are two consecutive configurations, and c is an existential configuration, such that either $\sharp = \sharp_r$ and $d' = l$, or $\sharp = \sharp_l$ and $d' = r$.
   (b) The initial configuration is eventually reached, and thereafter the accepting configuration is also eventually reached.

If w and its suffixes are accepted by $\mathcal{V}$, then by condition (1), w is a valid sequence of configurations; by condition (2), if w is the sequence of configurations guessed by the Turing machine T, then it visits the initial and accepting configurations infinitely often. It follows that the suffixes of all branches of a $2^I \times 2^O$-labeled 2^I-tree $((2^I)^*, \tau)$ are accepted by $\mathcal{V}$ iff T has an accepting run. ∎

6.3 Uninitialized Specifications

For a specification R, if the RP and URP for R coincides, then we say that R is *uninitialized*. In other words, R is uninitialized iff for every computation w, we have that w satisfies R iff all the suffixes of w satisfy R. Given a specification R over the input signals I and output signals O. The *uninitialized specification problem (USP)* for R asks whether R is uninitialized. If R is an uninitialized specification, then every FSM M that realizes R induces an uninitialized FSM M' (obtained from M by dropping its initial state) that realizes R. Hence for formalisms for which the URP is harder than the RP, the URP becomes easier if the specification is uninitialized.

Solving the USP for the specification R amounts to checking if R is equivalent to $always(R)$. Clearly, $always(R)$ implies R, thus we only need to check whether

R implies $always(R)$. For LTL formulas, this can be done by checking the validity of $R \rightarrow always(R)$, and for Büchi automata we need to solve the language-containment problem $L(R) \subseteq L(always(R))$. We show that this simple approach, like the naive solution for URP, is also optimal. The lower bounds can be obtained by a reduction from either the satisfiability or the validity problems for the corresponding formalisms (for a Büchi automaton R, we say that R is satisfiable iff it is nonempty and we say that R is valid iff it is universal). We describe both reductions below.

In the following, we let R be a specification over the finite sets I and O of input and output signals, respectively. For a computation $w = w_0 w_1 \ldots$ in $(2^{I \cup O})^\omega$ and a fresh signal $p \notin I \cup O$, we denote by $w \uplus p = (w_0 \cup \{p\})w_1 \ldots$ the computation obtained from w by adding p to w_0 and leaving the other states unchanged.

Lemma 6.1 *The USP for LTL, deterministic, nondeterministic, universal, or alternating Büchi automata, is at least as hard as the corresponding satisfiability problem.*

Proof. Consider a specification R over I and O. Let p be a signal not in I or O, and let R' be a specification over I and $O' = O \cup \{p\}$ such that for all $w \in (2^{I \cup O'})^\omega$, we have $w \models R'$ iff $w \models R$ and the first state of w contains p. Thus, if R is an LTL formula, then $R' = R \wedge p$ and if R is a Büchi automaton, then R' is obtained from R by removing from the initial states transitions that are labeled by letters that do not contain p (if the initial state is reachable, we also have to duplicate it). We claim that R is satisfiable iff R' is not uninitialized.

Assume first that R' is not uninitialized. Then, there is a computation w that satisfies R' and some of its suffixes do not satisfy R'. The computation w also satisfies R, implying that R is satisfiable. Assume now that R is satisfiable. Let $w \in (2^{I \cup O})^\omega$ be a computation that satisfies R. Then the computation $w' = w \uplus p$ satisfies R', but (since p holds only in the initial state of w') no proper suffix of w' satisfies R'. It follows that R is not uninitialized. ∎

Lemma 6.2 *The USP for LTL, deterministic, nondeterministic, universal, or alternating Büchi automata, is at least as hard as the corresponding validity problem.*

Proof. Consider a specification R over I and O. Let p be a signal not in I or O, and let R' be a specification over I and $O' = O \cup \{p\}$ such that for all $w \in (2^{I \cup O'})^\omega$, we have $w \models R'$

iff $w \models R$ or the first state of w contains p. Thus, if R is in LTL, then $R' = R \vee p$ and if R is a Büchi automaton, then R' is obtained from R by adding a transition labeled by letters that contain p, from the initial state to an accepting sink (if the initial state is reachable, we also have to duplicate it). We claim that R is valid iff R' is uninitialized.

Assume first that R is valid. Then, R' is valid too, so it must be uninitialized. Assume now that R is not valid. Let $w \in (2^{I \cup O})^\omega$ be a computation that does not satisfy R. Consider the computation $w' = (\{p\}) \cdot w$ that is obtained from w by prepending it with the state $\{p\}$ (any state in which p holds will do the job). Clearly, w' satisfies R', whereas its suffix w does not satisfy R'. It follows that R' is not uninitialized. ∎

We can now obtain tight bounds for the USP for the different formalisms we study in this chapter.

Theorem 6.5 *The USP for LTL, universal, nondeterministic or alternating Büchi automata is complete for PSPACE.*

Proof. For the upper bound, recall that R is uninitialized iff R implies $always(R)$. If the specification R is an LTL formula, then checking validity of $R \rightarrow always(R)$ is in PSPACE [SC85]. If R is an alternating (or universal) automaton, we have to check the language-containment problem $L(R) \subseteq L(always(R))$. For that, we can first construct a nondeterministic Büchi automaton R_n such that the size of R_n is exponential in the size of R and $L(R_n) = L(R)$ [MH84], and we construct a nondeterministic Rabin automaton R_c such that the size of R_c is exponential in the size of $always(R)$ and R_c complements $always(R)$ (that is, $L(R_c) = \Sigma^\omega \setminus L(always(R))$.) [MS95]. Now, the product of R_n and R_c is a nondeterministic Rabin automaton whose emptiness can be checked in nondeterministic logarithmic space, implying a PSPACE upper bound for the USP.

The lower bound follows from Lemmas 6.1 and 6.2, and from the fact that the satisfiability problem for LTL and universal (or alternating) Büchi automata, as well as the validity problem for nondeterministic Büchi automata are PSPACE-hard [SC85, HU79, Wol82]. ∎

Theorem 6.6 *The USP for deterministic Büchi automata is complete for NLOGSPACE.*

Proof. For a deterministic Büchi automaton R, the automaton $always(\mathcal{U})$ is universal, thus its complement R_c is a nondeterministic co-Büchi automaton. The product of R and

R_c can be defined as a nondeterministic Rabin automaton, whose emptiness problem can be solved in nondeterministic logarithmic space, implying an NLOGSPACE upper bound for the USP.

The lower bound follows from Lemma 6.1, and from the fact that the satisfiability problems for deterministic Büchi automata is NLOGSPACE-hard. ∎

	URP	RP	USP = Equivalence
LTL formulas	2EXP	2EXP	PSPACE
deterministic Büchi automata	EXP	Quadratic	NLOGSPACE
nondeterministic Büchi automata	2EXP	EXP	PSPACE
universal Büchi automata	EXP	EXP	PSPACE
alternating Büchi automata	2EXP	2EXP	PSPACE

Table 6.2: The complexity of the RP, URP, and USP.

Table 6.2 summarizes our results. Note that the URP for finite-state specifications that do not allow universal branching (e.g., deterministic and nondeterministic Büchi automata) are harder than their initialized counterpart; on the other hand, the URP for specifications that allow universal branching are not harder than their initialized counterpart. This is because the requirement that the FSM should work from any state is a universal requirement.

Bibliography

[AdAHM99] R. Alur, L. de Alfaro, T.A. Henzinger, and F.Y.C. Mang. Automating modular verification. In J.C.M. Baeten and S. Mauw, editors, *CONCUR 99: Concurrency Theory*, Lecture Notes in Computer Science 1664, pages 82–97. Springer-Verlag, 1999.

[AH98] R. Alur and T.A. Henzinger. Computer-aided verification. An introduction to model building and model checking for concurrent systems. http://www.eecs.berkeley.edu/~tah/CavBook, 1998, draft.

[AH99] R. Alur and T.A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15(1):7–48, 1999.

[AHK97] R. Alur, T.A. Henzinger, and O. Kupferman. Alternating-time temporal logic. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science*, pages 100–109. IEEE Computer Society Press, 1997.

[AHM+98] R. Alur, T.A. Henzinger, F.Y.C. Mang, S. Qadeer, S.K. Rajamani, and S. Tasiran. Mocha: modularity in model checking. In A.J. Hu and M.Y. Vardi, editors, *CAV 98: Computer Aided Verification*, Lecture Notes in Computer Science 1427, pages 521–525. Springer-Verlag, 1998.

[AL93] M. Abadi and L. Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, 1993.

[AL95] M. Abadi and L. Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, 1995.

[ALW89] M. Abadi, L. Lamport, and P. Wolper. Realizable and unrealizable concurrent program specifications. In G. Ausiello, M. Dezani-Ciancaglini, and S. Ronchi

Della Rocca, editors. *ICALP 89: Automata, Languages, and Programming*. Lecture Notes in Computer Science 372, pages 1-17. Springer-Verlag, 1989.

[ASSSV94] A. Aziz, T.R. Shiple, V. Singhal, and A.L. Sangiovanni-Vincentelli. Formula-dependent equivalence for ctl model checking. In D.L. Dill, editor. *CAV 94: Computer-Aided Verification*. Lecture Notes in Computer Science 818, pages 324-337. Springer-Verlag, 1994.

[BCCZ99] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In R. Cleaveland, editor. *TACAS 99: Tools and Algorithms for the Construction and Analysis of Systems*. Lecture Notes in Computer Science 1579, pages 193-207. Springer-Verlag, 1999.

[BCG+97] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B. Tabbara. *Hardware-Software Co-Design of Embedded Systems: The Polis Approach*. Kluwer Academic Press, 1997.

[Bee80] C. Beeri. On the membership problem for functional and multivalued dependencies in relational databases. *ACM Transactions on Database Systems*, 5:241-259, 1980.

[Ber98] G. Berry. The foundations of Esterel. In G. Plotkin, C. Stirling, and M. Tofte, editors. *Proof, Language and Interaction: Essays in Honour of Robin Milner*. Foundations of Computing Series. MIT Press, 1998.

[Ber99] G. Berry. The constructive semantics of Esterel. Draft book, current version 3.0, 1999.

[BG86] A. Blass and Y. Gurevich. Henkin quantifiers and complete problems. *Annal of Pure and Applied Logic*, 32:1-16, 1986.

[BG88] G. Berry and G. Gonthier. The synchronous programming language ESTEREL: Design, semantics, implementation. Technical Report 842, INRIA, 1988.

[BGM92] D. Barbara and H. Garcia-Molina. The demarcation protocol: a technique for maintaining linear arithmetic constraints in distributed database systems.

In Alain Pirotte, Claude Delobel, and Georg Gottlob, editors. *EDBT 92: Extending Database Technology*, Lecture Notes in Computer Science 580, pages 373–388. Springer-Verlag, 1992.

[BHSV+96] R. Brayton, G. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. Ranjan, S. Sarwary, T. Shiple, G. Swamy, and T. Villa. VIS: A system for verification and synthesis. In D.L. Dill, editor. *CAV 94: Computer Aided Verification 1102*, pages 428–432. Springer-Verlag, 1996.

[BL69] J.R. Büchi and L.H. Landweber. Solving sequential conditions by finite-state strategies. *Transactions of the American Mathematical Society*, 138:295–311, 1969.

[Bry86] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8), 1986.

[CC99] P. Cousot and R. Cousot. Refining model checking by abstract interpretation. *Automated Software Engineering Journal*, 6(1):69–95, 1999.

[CdAHM02] A. Chakrabarti, L. de Alfaro, T.A. Henzinger, and F.Y.C. Mang. Synchronous and bidirectional component interfaces. To appear in CAV 02: Computer-Aided Verification, 2002.

[CDFV88] R. Cieslak, C. Desclaux, A. Fawaz, and P. Varaiya. Supervisory control of discrete–event processes with partial observations. *IEEE Transactions on Automatic Control*, 33(3):249–260, 1988.

[CE81] E.M. Clarke and E.A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In Dexter Kozen, editor. *Logic of Programs: Workshop*. Lecture Notes in Computer Science 131, pages 52–71. Springer-Verlag, 1981.

[Cen] Berkeley Wireless Research Center. http://bwrc.eecs.berkeley.edu.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[Chu62] A. Church. Logic, arithmetic, and automata. In *Proceedings of International Congress of Mathematicians*, pages 23–35. Institut Mittag-Leffler, 1962.

[CKS81] A.K. Chandra, D.C. Kozen, and L.J. Stockmeyer. Alternation. *Journal of the ACM*, 28:114–133, 1981.

[CL90] K.L. Calvert and S.S. Lam. Formal methods for protocol conversion. *IEEE Journal of Selected Areas in Communications*, 8(1):127–142, 1990.

[CLM89] E.M. Clarke, D.E. Long, and K.L. McMillan. Compositional model checking. In *Proceedings of Fourth Annual Symposium on Logic in Computer Science*, pages 464–475. IEEE Computer Society Press, 1989.

[Coe95] T. Coe. Inside the Pentium FDIV bug. *Dr. Dobb's Journal of Software Tools*, 20(4):129–135, 1995.

[Com90] IEEE Standards Committee. *IEEE Standard Test Access Port and Boundary Scan Architecture*. IEEE, 345 East 47th Street, New York, NY10017-2394, July 1990. IEEE Standard 1149.1-1990.

[Con92] A. Condon. The complexity of stochastic games. *Information and Computation*, 96(2):203–224, 1992.

[CRS00] F. Cassez, M.D. Ryan, and P.-Y. Schobbens. Proving feature non-interaction with alternating-time temporal logic. In *Language Constructs for Describing Features*. Springer-Verlag, 2000.

[dAH01a] L. de Alfaro and T.A. Henzinger. Interface automata. In *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering*. ACM Press, 2001.

[dAH01b] L. de Alfaro and T.A. Henzinger. Interface theories for component-based design. In T.A. Henzinger and C.M. Kirsch, editors, *EMSOFT 01: Embedded Software*. Lecture Notes in Computer Science. Springer-Verlag, 2001.

[dAHM00a] L. de Alfaro, T.A. Henzinger, and F.Y.C. Mang. The control of synchronous systems. In C. Palamidessi, editor, *CONCUR 00: Concurrency Theory*. Lecture Notes in Computer Science 1877, pages 458–473. Springer-Verlag, 2000.

[dAHM00b] L. de Alfaro. T.A. Henzinger. and F.Y.C. Mang. Detecting errors before reaching them. In E.A. Emerson and A.P. Sistla. editors. *CAV 00: Computer-aided Verification*. Lecture Notes in Computer Science 1855. pages 186–201. Springer-Verlag. 2000.

[dAHM01] L. de Alfaro. T.A. Henzinger. and F.Y.C. Mang. The control of synchronous systems. part II. In K. G. Larsen and M. Nielsen. editors. *CONCUR 01: Concurrency Theory*. Lecture Notes in Computer Science 2154. pages 566–581. Springer-Verlag. 2001.

[Dam96] D. Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. PhD thesis. Technical University of Eindhoven. 1996.

[DFH+93] G. Dowek. A. Felty. H. Herbelin. G. Huet. C. Murthy. C. Parent. C. Paulin-Mohring. and B. Werner. The Coq proof assistant user's guide. Technical Report 154. INRIA-Rocquencourt. France. 1993.

[Dil89] D.L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-independent Circuits*. The MIT Press. 1989.

[DJ89] N. Dershowitz and J.P. Jouannaud. Rewrite systems. In J. van Leeuwen. editor. *Handbook of Theoretical Computer Science: Formal Methods and Semantics*. volume B. North-Holland. 1989.

[dM94] G. de Micheli. *Synthesis and Optimization of Digital Circuits*. New York: McGraw Hill. 1994.

[dSJSB+00] J. L. da Silva Jr.. M. Sgroi. F. De Bernardinis. S.F. Li. A. Sangiovanni-Vincentelli. and J. Rabaey. Wireless protocols design: Challenges and opportunities. In *CODES 00: Hardware/Software Codesign*. 2000.

[EC82] E.A. Emerson and E.M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*. 2:241–266. 1982.

[EJ91] E.A. Emerson and C.S. Jutla. Tree automata. mu-calculus and determinacy (extended abstract). In *Proceedings of the 32th Annual Symposium on Foun-*

dations of Computer Science. pages 368–377. IEEE Computer Society Press. 1991.

[EJS93] E.A. Emerson. C.S. Jutla. and A.P. Sistla. On model-checking of fragments on μ-calculus. In C. Courcoubetis. editor. *CAV 93: Computer-aided Verification*. Lecture Notes in Computer Science 697. pages 385–396. Springer-Verlag. 1993.

[EL86] E.A. Emerson and C.-L. Lei. Efficient model checking in fragments of the propositional mu-calculus. In *Proceedings of the First Annual Symposium on Logic in Computer Science*. pages 267–278. IEEE Computer Society Press. 1986.

[GH82] Y. Gurevich and L. Harrington. Trees. automata. and games. In *Proceedings of the 14th Annual Symposium on Theory of Computing*. pages 60–65. 1982.

[GLV95] G. Gottlob. N. Leone. and H. Veith. Second order logic and the weak exponential hierarchies. In J. Wiedermann and P. Hjek. editors. *MFCS 95: Mathematical Foundations of Compuer Science*. Lecture Notes in Computer Science 969. pages 66–81. Springer-Verlag. 1995.

[Gor88] M.J.C. Gordon. HOL: A proof generating system for higher-order logic. In G.M. Birtwistle and P.A. Subrahmanyam. editors. *VLSI Specification. Verification and Synthesis*. pages 73–128. Kluwer Academic Publishers. 1988.

[GS97] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In O. Grumberg. editor. *CAV 97: Computer-Aided Verification*. Lecture Notes in Computer Science 1254. pages 72–83. Springer-Verlag. 1997.

[GSSAL98] G. Gawlick. R. Segala. J. Sogaard-Andersen. and N. Lynch. Liveness in timed and untimed systems. *Information and Computation*. 141(2):119–171. 1998.

[Hal93] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers. 1993.

[Hen61] L. Henkin. Some remarks on infinitely long formulas. In *Infinitistic methods. Proceedings of the Sympopsium on Foundations of Mathematics*. pages 167–183. Wydawnictwo. Naukowe and Pergamon Press. 1961.

[HK97] T.A. Henzinger and P.W. Kopke. Discrete-time control for rectangular hybrid automata. In P. Degano, R. Gorrieri, and A. Marchetti-Spaccamela, editors. *ICALP 97: Automata, Languages, and Programming*. Lecture Notes in Computer Science 1256, pages 582–593. Springer-Verlag, 1997.

[HKKM02] T.A. Henzinger, S.C. Krishnan, O. Kupferman, and F.Y.C. Mang. Synthesis of uninitialized systems. To appear in ICALP 02: Automata, Languages, and Programming, 2002.

[HKQ98] T.A. Henzinger, O. Kupferman, and S. Qadeer. From *pre*historic to *post*modern symbolic model checking. In A.J. Hu and M.Y. Vardi, editors. *CAV 98: Computer-aided Verification*. Lecture Notes in Computer Science 1427, pages 195–206. Springer-Verlag, 1998.

[Hol91] G.J. Holzman. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[HQR98] T.A. Henzinger, S. Qadeer, and S.K. Rajamani. You assume, we guarantee: methodology and case studies. In A.J. Hu and M.Y. Vardi, editors. *CAV 98: Computer-aided Verification*. Lecture Notes in Computer Science 1427, pages 440–451. Springer-Verlag, 1998.

[HQR00] T.A. Henzinger, S. Qadeer, and S.K. Rajamani. Decomposing refinement proofs using assume-guarantee reasoning. In *Proceedings of the International Conference on Computer-aided Design*, pages 245–252. IEEE Computer Society Press, 2000.

[HR72] R. Hossley and C. Rackoff. The emptiness problem for automata on infinite trees. In *IEEE Symposium on 13th Annual Symposium on Foundations of Computer Science* pages 121–124. IEEE Computer Society Press, 1972.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[Jur00] M. Jurdzinski. *Games for Verification: Algorithmic Issues*. PhD thesis, University of Aarhus, 2000.

[Klo91] J.W. Klop. Term rewriting systems. In S. Abramsky, D. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 1, pages 1–116. Oxford University Press, 1991.

[KMP98] M. Kaufmann, A. Martin, and C. Pixley. Design constraints in symbolic model checking. In A.J. Hu and M.Y. Vardi, editors, *CAV 98: Computer-aided Verification*, Lecture Notes in Computer Science 1427, pages 477–487. Springer-Verlag, 1998.

[KMTV00] O. Kupferman, P. Madhusudan, P.S. Thiagarajan, and M.Y. Vardi. Open systems in reactive environments: Control and synthesis. In C. Palamidessi, editor, *CONCUR 00: Concurrency Theory*, Lecture Notes in Computer Science 1877, pages 92–107. Springer-Verlag, 2000.

[Koz83] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27(2):333–354, 1983.

[KR00] S. Kremer and J.-F. Raskin. Formal verification of non-repudiation protocols – a game approach. In *FMCS 2000: Formal Methods and Computer Security*, 2000.

[KR01] S. Kremer and J.-F. Raskin. A game-based verification of non-repudiation and fair exchange protocols. In K. G. Larsen and M. Nielsen, editors, *CONCUR 01: Concurrency Theory*, Lecture Notes in Computer Science 2154, pages 551–565. Springer-Verlag, 2001.

[KS97] R. Kumar and M.A. Shayman. Centralized and decentralized supervisory control of nondeterministic systems under partial observation. *SIAM Journal of Control and Optimization*, 35(2):363–383, 1997.

[Kur94] R.P. Kurshan. *Computer-aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, 1994.

[KV96] O. Kupferman and M.Y. Vardi. Module checking. In R. Alur and T.A. Henzinger, editors, *CAV 96: Computer Aided Verification*, Lecture Notes in Computer Science 1102, pages 75–86. Springer-Verlag, 1996.

[KV99] O. Kupferman and M.Y. Vardi. Church's problem revisited. *The Bulletin of Symbolic Logic*. 5(2):245-263. 1999.

[KV00] O. Kupferman and M.Y. Vardi. μ-calculus synthesis. In M. Nielsen and B. Rovan, editors. *MFCS 00: Mathematical Foundations of Computer Science*. Lecture Notes in Computer Science 1893. pages 497-507. Springer-Verlag. 2000.

[LW88a] F. Lin and W. M. Wonham. Decentralized supervisory control of discrete-event systems. *Information Science*. 44(3):199-224. 1988.

[LW88b] F. Lin and W. M. Wonham. On observability of discrete-event systmes. *Information Science*. 44(3):173-198. 1988.

[LW90] F. Lin and W. M Wonham. Decentralized control and coordination of discrete-event systems with partial observation. *IEEE Transactions on Automatic Control*. 35(12):1330 1337. 1990.

[Lyn96] N.A. Lynch. *Distributed Algorithms*. Morgan-Kaufmann. 1996.

[Mal94] S. Malik. Analysis of cyclic combinational circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. 13(7):950-956. 1994.

[McM93] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers. 1993.

[McM97] K.L. McMillan. A compositional rule for hardware design refinement. In O. Grumberg, editor. *CAV 97: Computer-Aided Verification*. Lecture Notes in Computer Science 1254. pages 24-35. Springer-Verlag. 1997.

[McN93] R. McNaughton. Infinite games played on finite graphs. *Annal of Pure and Applied Logic*. 65:149-184. 1993.

[MH84] S. Miyano and T. Hayashi. Alternating finite automata on ω-words. *Theoretical Computer Science*. 32:321-230. 1984.

[Mil71] R. Milner. An algebraic definition of simulation between programs. In *Second International Joint Conference on Artificial Intelligence*. pages 481-489. The British Computer Society. 1971.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice-Hall. 1989.

[MS84] D.E. Muller and P. E. Schupp. Alternating automata on infinite objects. determinacy and Rabin's theorem. In M. Nivat and D. Perrin. editors. *Automata on Infinite Words.* Lecture Notes in Computer Science 192. pages 100–107. Springer-Verlag. 1984.

[MS95] D.E. Muller and P.E. Schupp. Simulating aternating tree automata by nondeterministic automata: New results and new proofs of theorems of Rabin. McNaughton and Safra. *Theoretical Computer Science.* 141:69–107. 1995.

[Muc84] A.A. Muchnik. Games on infinite trees and automata with dead-ends. *Semiotics and Information.* 24:17–40. 1984. (in Russian).

[MW80] Z. Manna and R. Waldinger. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems.* 2(1):90–121. 1980.

[Oku86] K. Okumura. A formal protocol conversion method. In *Proceedings ACM SIGCOMM.* pages 30–37. ACM Press. 1986.

[Pap94] C.H. Papadimitriou. *Computational Complexity.* Addison-Wesley. 1994.

[Pau94] L.C. Paulson. *Isabelle: a generic theorem prover.* Lecture Notes in Computer Science 828. Springer-Verlag. 1994.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science.* pages 46–57. IEEE Computer Society Press. 1977.

[Pnu81] A. Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science.* 13:45–60. 1981.

[PR89a] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proceedings of the 16th Annual Symposium on Principles of Programming Languages.* pages 179–190. ACM Press. 1989.

[PR89b] A. Pnueli and R. Rosner. On the synthesis of an asynchronous reactive module. In G. Ausiello. M. Dezani-Ciancaglini. and S. Ronchi Della Rocca. editors.

ICALP 89: Automata, Languages, and Programs. Lecture Notes in Computer Science 372, pages 652 - 671. Springer-Verlag, 1989.

[PR90] A. Pneuli and R. Rosner. Distributed reactive systems are hard to synthesis. In *Proceedings of the 31th Annual Symposium on Foundations of Computer Science*, pages 746–757. IEEE Computer Society Press, 1990.

[PRSV98] R. Passerone, J.A. Rowson, and A.L. Sangiovanni-Vincentelli. Automatic synthesis of interfaces between incompatible protocols. In *Proceedings of the 35th Annual Conference on Design Automation Conference*, pages 8–13, 1998.

[Pur95] A. Puri. *Theory of Hybrid Systems and Discrete Event Systems*. PhD thesis, University of California at Berkeley, 1995.

[QBSP96] S. Qadeer, R. K. Brayton, V. Singhal, and C. Pixley. Latch redundancy removal without global reset. In *Proceedings of the International Conference on Computer Design*, pages 432 439. IEEE Computer Society Press, 1996.

[QS81] J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in Cesar. In *Proceedings of the 5th International Symposium on Programming*, Lecture Notes in Computer Science 137, pages 337 -351. Springer-Verlag, 1981.

[Rab69] M.O. Rabin. Decidability of second order theories and automata on infinite trees. *Transactions of the American Mathematical Society*, 141:1-35, 1969.

[Rab70] M.O. Rabin. Weakly definable relations and special automata. In Y. Bar-Hillel, editor, *Mathematical Logic and Foundations of Set theory*. North-Holland, 1970.

[Rab72] M.O. Rabin. Automata on infinite objects and Church's problem. In *Number 13 in Regional Conference Series in Mathematics*. American Mathematical Society, 1972.

[Raj99] S.K. Rajamani. *New Directions in Refinement Checking*. PhD thesis, University of California at Berkeley, 1999.

[Rei84] J.H. Reif. The compexity of two-player games of incomplete information. *Journal of Computer and System Sciences*, 29:274-301, 1984.

[RG95] R.Kumar and V.K. Garg. *Modeling and control of logical discrete event systems.* Kluwer Academic Publishers. 1995.

[Ros92] R. Rosner. *Modular Synthesis of Reactive Systems.* PhD thesis. Weizmann Institute of Science. Rehovot. Israel. 1992.

[RW87] P.J. Ramadge and W.M. Wonham. Supervisory control of a class of discrete-event processes. *SIAM Journal of Control and Optimization.* 25:206–230. 1987.

[RW89] P.J.G. Ramadge and W.M. Wonham. The control of discrete event systems. *IEEE Transactions on Control Theory.* 77:81–98. 1989.

[RW92] K. Rudie and W. Wonham. Think globally. act locally: Decentralized supervisory control. *IEEE Transactions on Automatic Control.* 37(11):1692–1708. 1992.

[Saf88] S. Safra. On the complexity of omega-automata. In *Proceedings of the 29th Annual Symposium on Foundations of Computer Science.* pages 319–327. IEEE Computer Society Press. 1988.

[SBT96] T. Shiple. G. Berry. and H. Touati. Constructive analysis of cyclic circuits. In *EDTC 96: European Design and Test Conference.* pages 328–333. 1996.

[SC85] A.P. Sistla and E.M. Clarke. The complexity of propositional linear temporal logic. *Journal of the ACM.* 32:733–749. 1985.

[SdSJB+00] M. Sgroi. J. da Silva Jr.. F. De Bernardinis. F. Burghardt. A. Sangiovanni-Vincentelli. and J. Rabaey. Designing wireless protocols: Methodology and applications. In *ICASSP 00: Acoustics. Speech. and Signal Processing.* 2000.

[SP94] V. Singhal and C. Pixley. The verification problem for safe replaceability. In D.L. Dill. editor. *CAV 94: Computer-Aided Verification.* Lecture Notes in Computer Science 818. pages 311–323. Springer-Verlag. 1994.

[Sta85] E.W. Stark. A proof technique for rely/guarantee properties. In *Proceedings of the 5th Conference on Foundations of Software Technology and Theoretical Computer Science.* Lecture Notes in Computer Science 206. pages 369–391. Springer-Verlag. 1985.

[Sti95] C. Stirling. Local model checking games. In Insup Lee and Scott A. Smolka, editors. *CONCUR 95: Concurrency Theory*. Lecture Notes in Computer Science 962, pages 1–11. Springer-Verlag, 1995.

[Sti97] C. Stirling. Bisimulation, model checking and other games. Available at http://www.dcs.ed.ac.uk/home/cps/.. 1997.

[Tho93] W. Thomas. The Ehrenfeucht-Frasse game in theoretical computer science (extended abstract). In M.-C. Gaudel and J.-P. Jouannaud, editors. *TAPSOFT 93: Theory and Practice of Software Development*. Lecture Notes in Computer Science 668, pages 559–568. Springer-Verlag, 1993.

[Tho95] W. Thomas. On the synthesis of strategies in infinite games. In E. W. Mayr and C. Puech, editors. *STACS 95: Theoretical Aspects of Computer Science*. Lecture Notes in Computer Science 900, pages 1–13. Springer-Verlag, 1995.

[TW68] J.W. Thatcher and J.B. Wright. Generalized finite-automata theory with an application to a decision problem of second-order logic. *Mathematical Systems Theory*, 2:57–81, 1968.

[TW94] J.G. Thistle and W.M. Wonham. Control of infinite behavior of finite automata. *SIAM J. on Control and Optimization*, 32(4):1075–1097, 1994.

[Var95] M.Y. Vardi. An automata-theoretic approach to fair realizability and synthesis. In P. Wolper, editor. *CAV 95: Computer-Aided Verification*. Lecture Notes in Computer Science 939, pages 267–292. Springer-Verlag, 1995.

[VW94] M.Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, 1994.

[Wal70] W. Walkoe. Finite partially-ordered quantification. *Journal of Symbolic Logic*, 35:535–555, 1970.

[Wol82] P. Wolper. *Synthesis of Communicating Processes from Temporal Logic Specifications*. PhD thesis, Stanford University, 1982.

[Wol95] E.S. Wolf. *Hierarchical models of synchronous circuits for formal verification and substitution*. PhD thesis, Stanford University, 1995.

[WW96] K. Wong and W. Wonham. Hierarchical control of discrete-event systems. *Discrete Event Dynamic Systems*, 6:241-306, 1996.

[YD98] C.H. Yang and D.L. Dill. Validation with guided search of the state space. In *Proceedings of the 35th Annual Conference on Design Automation Conference*, pages 599–604, 1998.

[YSAA97] J. Yuan, J. Shen, J.A. Abraham, and A. Aziz. On combining formal and informal verification. In O. Grumberg, editor, *CAV 97: Computer Aided Verification*, Lecture Notes in Computer Science 1254, pages 376–387. Springer-Verlag, 1997.

[ZP96] U. Zwick and M. Paterson. The complexity of mean payoff games on graphs. *Theoretical Computer Science*, 158(1-2):343-359, 1996.