Compositional and Hierarchical Techniques for the Formal Verification of Real-Time Systems

by

Serdar Taşıran

B.S. (Bilkent University, Ankara, Turkey) 1991
M.S. (University of California, Berkeley) 1995

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

in

Engineering - Electrical Engineering and Computer Sciences

in the

GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA at BERKELEY

Committee in charge:

Professor Robert K. Brayton, Chair
Professor Alberto L. Sangiovanni-Vincentelli
Professor Jack H. Silver

Fall 1998

The dissertation of Serdar Taşıran is approved:

Chair Date

Date

Date

University of California at Berkeley

1998

Compositional and Hierarchical Techniques for the Formal Verification of Real-Time Systems

by

Serdar Taşıran

Abstract

Compositional and Hierarchical Techniques for the Formal Verification of Real-Time Systems

by

Serdar Taşıran
Doctor of Philosophy in Engineering - Electrical Engineering and Computer Sciences
University of California at Berkeley
Professor Robert K. Brayton, Chair

The focus of this dissertation is the formal verification of real-time systems: systems with discrete control structures operating over a continuous time domain. Validation of functionality and timing constitutes a major portion of any electronic system design effort. The prevalent approach based on software simulation is no longer satisfactory, not only because it does not provide formal assurance, but also because it requires an inordinate amount of computational and human resources. The scale of modern electronic systems make formal verification techniques indispensable.

Algorithmic verification techniques suffer from "compositional complexity": the fact that the size of a system's state-space grows exponentially in the number and sizes of its components. For real-time systems, this difficulty is compounded by the representation of the timing information. Recent research has produced formal verification algorithms with significantly improved performance; however, system sizes multiply every few years, and it is unlikely that these algorithms will be made exponentially more efficient over time. To counter the computational complexity of verifying real-time systems, a scheme for decomposing the problem is required. Toward this end, we develop a compositional and hierarchical verification methodology.

As our modeling formalism, we use a variant of timed automata and languages. We explore several notions of refinement for real-time systems, present complexity results and algorithms. An assume-guarantee style verification rule is proposed for dividing the hierarchical verification task into subtasks, each one corresponding to one module in the system.

The soundness of this rule used in conjunction with the refinement preorders is proven. The algorithms developed were implemented on the COSPAN and MOCHA verification platforms.

We demonstrate the efficacy of our framework on two practical circuit designs: the Seitz queue and the STARI communication chip. We also specialize our techniques to the delay modeling and computation problem for combinational circuits. Modified modeling assumptions brought about by shrinking circuit feature sizes has revived interest in this problem. We report results that improve on previous timed-automaton-based methods significantly.

Professor Robert K. Brayton
Dissertation Committee Chair

To my mother, Gülen, my father, Sadettin,
and my brother, Selim

Annem Gülen, babam Sadettin
ve kardeşim Selim'e

Contents

List of Figures

List of Tables

Acknowledgements

I am very fortunate to have had Prof. Robert Brayton as my advisor. I benefited greatly from his unique advising style that combines guidance, support and encouragement with the freedom to explore and develop a perspective. I learned how to conduct and evaluate research from him. Bob's candor, curiosity and keenness, his positive approach to research, life and people are an inspiration. I am grateful to him for his contribution to my development as a researcher and a person.

I thank Prof. Alberto Sangiovanni-Vincentelli for serving on my thesis and qualifying examination committees. Alberto has helped me put my work in perspective on numerous occasions. I look up to his energy, his ability to define new research directions and develop key insights.

A great deal of what I know about doing research I learned from Rajeev Alur and Robert Kurshan. The line of research presented in this dissertation was shaped while I worked with them at Bell Labs. I appreciate their contribution and continued support.

I am grateful to Prof. Henzinger for his contributions to my work during the last year of my graduate career, and for serving on my qualifying examination committee. My thanks go to him also for his guidance and support during my job search.

It was refreshing to interact with Prof. Kurt Keutzer and to be exposed to his perspective and approach to research during my last semester as a graduate student. I appreciate the time and thought he took to help me shape my future research career.

Professors Brayton, Henzinger, Keutzer, Newton and Sangiovanni-Vincentelli have built the much envied Berkeley CAD Group reputation. I am grateful to them for the environment they have created. Here, I have had the opportunity to grow and interact with some of the most talented and accomplished researchers in the field. Brad Krebs, Flora Oviedo, Kia Cooper, and many others have contributed to a pleasant and supportive working experience. It will be hard to find an environment that can match CAD Group.

I thank Prof. Jack Silver for teaching me logic and serving on my thesis committee. I would also like to express my appreciation for Professors Jack Potter (anthropology) and Michael Watts (geography) for their influence on my view of technology and the world.

I have had the privilege of collaborating with Adnan Aziz, Ramin Hojati, Yuji Kukimoto, Sunil Khatri, Shaz Qadeer and Sriram Rajamani on publications that have been incorporated into this thesis. I gratefully acknowledge their contribution to my research. I

learned about verification together with the "Berkeley verifiers", which, in addition to the above, includes Felice Balarin, Sriram Krishnan, Vigyan Singhal and Tom Shiple. I must also mention valuable collaboration with Sergio Yovine and Oded Maler.

Yuji Kukimoto, Rajeev Ranjan, Harry Hsieh and I practiced for the prelims together; attended each others' quals practice talks; in short, we found our way through graduate school together. I appreciate their friendship.

Many of the ideas in this thesis were conceived and written at cafes around the Berkeley campus. To everyone who worked at Nefeli, Odyssia, Brewed Awakenings (formerly Coffee Connection) and the I-House cafe, thank you.

I would like to acknowledge the financial support provided by the Scientific and Technical Research Council of Turkey and the Semiconductor Research Corporation.

Selim Ünlü and Pınar Zengingönül have been my big brother and sister. During my first year in the U.S., their home was a refuge to me. They helped me make important career decisions. Selim and Pınar, Cingöz thanks you for your support.

Alper Demir and I grappled through life together through our undergraduate and graduate years. Alper, Serkan Hoşten, Reha Tütüncü, and And Türken are not only great friends but also have greatly influenced my intellectual development. I thank them.

The memories of my life at Berkeley are inseparable from Kristen Sortais ("my old carpet") and Annette Roman. I thank them for their love and support. Special thanks go to my friends Serdar Aksu, Kerem Bülbül, and Önder Çetin, who were there for me during some of the most trying phases of my life at Berkeley.

My extended family (the Özbeks and the Taşırans), with its Turkish, American and Japanese branches, has always been a source of strength and happiness for me. Among them, my uncle Akif Özbek deserves special mention. He has always been an uplifting influence on me. I admire his optimism and enthusiasm and hope that I inherited some of it.

My deepest gratitude goes to my family. This thesis is dedicated to them for their love and sacrifices.

Chapter 1

Introduction

1.1 Formal Verification

The rapid advance of technology has allowed the realization of ever more ambitious design goals in the form of integrated systems. Most notably, the electronics industry and research community have made, and are making, a concerted effort to double the size of what is considered a single system every few years. This has brought about significant challenges for developers of tools for electronic systems design. Even more formidable is the task faced by verification engineers and verification tool designers. The validation of a design, even today, takes as much computing and human resources as the actual implementation of the design.

A typical design process involves several iterations of the following cycle: First the design intent is specified in some form, and then designers build an "implementation" of this specification. The implementation process typically consists of expressing the design intent at a more refined level, making more concrete what may be left unspecified initially. This refined description then serves as the specification for the next iteration of the design cycle. Within this process, two sorts of verification problems arise: implementation verification, and design verification. Implementation verification is the process of ensuring that the implementation built by the designers is consistent with its specification. This is often achieved by showing that the implementation is equivalent to or is a refinement of the system specification. Design verification, by contrast, involves the validation of the initial specification, that is, increasing confidence that a system implemented according to the specification will behave as desired. Since there is no correct "golden" specification to start from, the design verification problem can not be solved by equivalence checking tools. It

is the more challenging of the two verification problems, and formalizing it is often more difficult.

The prevalent approach to design verification is software simulation. By simulating the response of the system to a set of stimuli, designers exercise the design in various ways in order to uncover undesired behavior that the system may exhibit. Simulation has intuitive appeal, and is valuable in discovering simple errors quickly and building familiarity with the design. However, as a verification paradigm, it is becoming increasingly unsatisfactory. The amount of simulation required for recent microprocessors has proved prohibitively costly. Moreover, simulation using commercial tools yields no more information than the waveforms observed at selected circuit nodes. Even after a large number of simulation runs, there is no formal measure of how large a portion of the system's behavior is "covered". It seems infeasible to build the even larger integrated systems of the future in a reasonable time without resorting to more mathematical, formal techniques.

1.2 Computer-Aided Formal Verification

Formal design verification is the process of proving within a mathematical framework either that a system satisfies a property, or that all of the possible behaviors of a system are consistent with a specification. The system and its specification are described using (possibly different) formal languages. A formal semantics ascribes meanings to these descriptions and also defines formally what it means for the system to satisfy the specification (the *satisfaction relation*). A mechanism for checking this relation is the last component of a formal verification framework. The work presented in this dissertation falls into the category of algorithmic techniques for computer-aided formal verification. This is in contrast to theorem proving techniques (see [Gup91] for a review), which provide computer assistance for manually proving the satisfaction relation.

In most algorithmic formal verification techniques, the semantic model for the system is some form of a *state-transition graph* which consists of a *state-space*, i.e., the set of possible configurations that the system can be at a given point in time, and a *transition relation* describing the evolution of the system over time or in response to input stimuli. A variety of languages exist for specifying systems. For specifying digital hardware, VHDL and Verilog are the most popular choices. A plethora of computer programming languages exist for specifying software systems. It is more convenient specify certain systems, such as digital

circuits described at the register transfer level, or digital signal processing systems described as block diagrams, using graphical interfaces. Many system description languages do not have a uniquely defined formal semantics, but are interpreted by the tools that operate on them as state-transition systems according to some concurrency model. Roughly speaking, the concurrency model specifies which variables define the state-space and how they are updated over time. Many tools convert the system description to an internal format that more closely reflects the concurrency model. BLIF-MV [BCH+91], the intermediate format for HSIS [ABB+94] and VIS [BHSV+96] and S/R, the intermediate format for COSPAN [HHK96] are some examples. A majority of these intermediate formats resemble automata, and the verification methods used on them are based on automata theory [HU79]. Other formalisms, such as Petri net based ones [Rei85] and process algebra based ones (Calculus of communicating systems (CCS) [Mil80] and communicating sequential processes (CSP) [Hoa78] being the most prominent ones) are geared towards more specialized uses such as communications protocols, asynchronous circuits and data-flow networks.

A key feature of system specification languages and the underlying concurrency models is modularity. Systems are described in the form of coordinating components, each of them realizing a part of the functionality. This is an essential feature for complexity management, and ease of design. It is common practice to carry out large designs by first dividing them into smaller and less complex ones.

An even wider spectrum of formalisms exist for expressing specifications. They can be broadly divided into two categories: logic-based, and automaton-based specifications. In the logic-based approach, first introduced by [Pnu77], properties are stated as formulae in a temporal logic (See [Eme90] for a review). Popular choices are linear temporal logic (LTL) [Pnu77, OL82] and computational tree logic (CTL) [BMP81, EC82]. Then, it is checked if the system satisfies the formula. *Model checking* methods, pioneered by [CES86], [LP85], [GW91], and [BCM+92] among others, have been successful in practice and have been integrated into many verification tools, SMV [McM93], HSIS [ABB+94] and VIS [BHSV+96] to name a few. By exploring the state-space of the design, model checkers determine if the system is a "model" of the formula, i.e., whether in all behaviors of the system, the property is satisfied.

In automaton-based approaches, the same formalism is used for describing both the systems and the specifications. The specification is a more abstract and smaller state-transition system. It is then desired to show that all behaviors of the system are part

of the specification also. This is often achieved by showing an "implementation relation" between the system and its specification such as *language containment* [Tho81], *simulation* [DHWT91] or a *homomorphism* [Kur94]. This paradigm has two advantages: First, most of the time, it is more convenient to express specifications in the same language as the system itself. Second, the same algorithms can be used for implementation verification also, because one can view the specification and the implementation as descriptions of the same design at two different levels of detail. The verification tools HSIS [ABB+94, Tas95, THB95] and COSPAN [HHK96] support specifications described as automata and implementation verification using them. All of these approaches work by exploring the state-spaces of the system in conjunction with that of the specification and determining whether all behaviors in the system's state-space can be matched by the specification's state-space.

By explicitly exploring or mathematically reasoning about all possible behaviors of the system, algorithmic techniques are able to either guarantee the correctness of properties and specifications, or produce *error traces*: counterexamples along with input stimuli, which indicate how it is possible for the system to violate a specification. When it is possible to perform algorithmic formal verification of a system, it is invaluable both for debugging and for ensuring properties. The use of implicit representations (such as the binary decision diagrams of [Bry86]) for sets of states and transitions, and symbolic techniques for graph exploration ([TSL+90], [BCM+92], [DH93], [McM93] among many others) enabled a major leap in the sizes of the systems that could be analyzed [McM93, BCM+92]. The size of an implicit representation is not proportional to the size of the set it represents, and thus, exploration of very large state-spaces was possible. The initial success of verification tools developed at universities, combined with the intensified need for formal tools, has fueled industrial interest in formal verification. Internal tools have been developed (by Motorola, IBM, and Intel, to name a few) and commercial tools (such as Lucent Technologies' FormalCheck, IBM's RuleBase, Chrysalis' Design INSIGHT) have started appearing on the market.

1.3 Verification of Timing Properties

Most of the earlier work on algorithmic formal verification focused on verifying the Boolean functionality of synchronous digital designs or computer software. System models were either completely synchronous (hardware) or completely asynchronous (asynchronous

hardware and software). Time factored into specifications only as a qualitative notion; properties asserted that certain conditions always or eventually hold. While qualitative modeling of time facilitates efficient verification of certain "sanity properties"[1] , it is not a satisfactory means of validating the correctness and performance of many systems which depend crucially on timing. Combinational circuits must meet the clock speed requirements; embedded controllers must be able to perform routine tasks and respond to interrupts within a given bound; asynchronous interfaces must ensure correct communication over channels with varying delays; to verify properties about the performance of a computer the response time of each subsystem must be modeled accurately. This requires a quantitative notion of time to be incorporated both into the system models and into the specification formalism. Among several frameworks that attempt this [Ram74, CR83, LA90, Ost90], the timed automata framework pioneered by [AD94] has proved to be the most natural and operational way of incorporating quantitative time into state transition structures. Timed automata have, over time, become the canonical representation for real-time systems, and several timing verification tools (COSPAN [AK96], KRONOS [DOY94], UPPAAL [LPY95], and MOCHA [AHM+98]), including the algorithms that will be presented in this dissertation, have taken this formalism as their basis.

Timed automata model time as a real-valued ($\mathbb{R}$) quantity. This facilitates more precise modeling of analog or asynchronous systems as well as systems with components that run on different clocks. If time is modeled as a discrete-valued quantity, a time unit must be chosen in advance for each system. When two systems are connected to form a larger system, the automata modeling them need to be composed. For this composition to be well-defined, the two automata must have the same time unit. If not, one of them needs to be modified in order to express the same behavior using a different time unit [Alu91]. Because of this, if a discrete time domain is used, the semantics of a timed automaton can not be defined independently of the other systems that it will be used in conjunction with. Therefore, from a formal analysis point of view, it is more desirable to operate on the real time domain. One possible drawback of using $\mathbb{R}$ as the time domain is added complexity. However, with proper analysis techniques ([AIKY92], [HMP92], [GPV94], [ABH+97], [TKY+98]), the dense time domain does not increase the complexity of analysis unless the added expressiveness is indeed necessary. With $\mathbb{R}$ as the time domain, timed automata make convenient and

[1] Such as "the system eventually responds to a request".

efficient models for heterogeneous systems, e.g., systems with independently-clocked digital components, or digital systems interfacing with a continuous-time environment. This is what is referred to by the term "real-time system".

All specification formalisms and verification frameworks mentioned in the previous sections have their "timed" counterparts. However, the density of the time domain makes the underlying theory significantly more involved and furthermore requires specialized verification methods [Alu91, Eme92]. Some existing methods are outlined in Chapter 2. The focus of this thesis is the algorithmic formal verification of real-time systems.

1.4 The Complexity Problem

The major obstacle in the way of practical applicability of algorithmic formal verification is the size of the representation of the state-space of the system. The size of this state-space is roughly exponential in the number of state variables. This number is doubled every few years, and, as a result, the sizes of the state-spaces to be analyzed increases super-exponentially over time. This problem, notoriously known as "state space explosion" has been at the center of most research on formal verification. Implicit representations and symbolic techniques for state-space traversal have been researched intensely, and there has been considerable improvement in their capacities. Partial-order reduction methods have also brought about significant improvements [GW90, GW91, Rok93, BM98]. Nevertheless, excepting certain designs with particular structures, at best, systems with several hundred state variables can be handled in their entirety. This falls very short of the sizes of current integrated circuit designs: modern microprocessors can have tens of thousands of memory elements. The problem is exacerbated for real-time systems because timing information compounds the state-space.

Super-exponential improvements in the performance of verification tools are unlikely. This leaves abstraction- and decomposition-based techniques as the only ones with prospects for future applicability. The fact that these techniques parallel the way humans cope with the information associated with a large design lends further promise.

In the *hierarchical verification* paradigm, a system is described at several different levels of detail. The goal is to verify properties using the most abstract, therefore smallest description. The hierarchy of descriptions may have come about either because the design process is carried out in a top-down fashion through a series of refinements or because the

implementation level description is too complex for automatic verification algorithms and simpler, more abstract models have been constructed for validation purposes. If for each pair of adjacent levels the more refined one is a correct implementation of the more abstract one, the properties proved are satisfied by the actual implementation of the system[2].

While performing hierarchical verification, we must avoid constructing monolithic representations for the state-spaces of refined descriptions. Doing otherwise would defeat the purpose of hierarchical verification. The goal in *compositional verification* is to decompose the problem of verifying refinement into subproblems in order to achieve this. The form of the decomposition often follows the physical or logical structure of the system. The refinement proofs for the individual modules put together imply correct refinement for the entire system, and property verification can then be performed in the manner described in the preceding paragraph.

The main contributions presented in this dissertation are theory and tools for, and experiments with hierarchical and compositional verification of real-time systems.

1.5 Outline of The Dissertation

The mathematical model used in this dissertation to represent real-time systems is a variant of timed automata model of Alur and Dill [AD94]. Timed automata can be viewed as finite automata augmented with real-valued stopwatch variables, called clocks. Clocks are used to keep track of the time that elapses between events. This degree of abstraction in modeling is especially useful if the essential characteristics of the continuous behavior of the system can be captured by constraints on the time elapsed between distinct states. Timed automata and our modeling and verification formalism based on them are introduced in Chapter 2.

As indicated earlier, formal verification with timed automata is in practice much more complex than verification with finite automata. Iterative techniques [AIKY92, LB93, Bal95b] verify properties of timed automata by disregarding the timing information at first and then gradually bringing in timing constraints as needed. Chapter 3 presents our work on improving iterative techniques for timing verification. While we found that this abstraction technique is helpful if much of the timing information in the system description

[2]Clearly, for this to hold, the "implementation relation" must preserve the properties expressible in the specification formalism.

is redundant, in general this form of complexity relief is not sufficient for handling systems with large control structures.

Chapters 4 and 5 address in greater depth the issues of abstraction and complexity management for real-time systems. In Chapter 4 we investigate several notions of abstraction for real-time systems. We prove that checking whether one timed automaton "simulates" another is decidable, and present an algorithm for performing this check. This algorithm can be computationally expensive, so, as a more efficient alternative, we study homomorphisms. Here, a mapping between the discrete control structures of two timed systems is given, and one checks whether, under this mapping, the timed behavior of one automaton conforms to that of the other. We present an algorithm for performing this check.

Chapter 5 presents our research on modular verification of abstractions. While proving that the abstract model for a component is correct, typically, assumptions need to be made about the environment that it operates in. Assume-guarantee style reasoning is a way of making a (possibly circular) set of assumptions about the environment of each component and discharging them in a sound manner. In Chapter 5, we develop a theory for assume-guarantee style reasoning using timed automata, and demonstrate their use on a small asynchronous circuit.

Chapter 6 is dedicated to the verification of STARI, a high-bandwidth communication chip, using the techniques presented in Chapters 4 and 5. STARI has now become a benchmark for timing verification techniques. The code describing the verification of STARI is provided in the Appendix.

While verification of real-time behavior has been an intensely studied topic in recent years, current practice continues to be to design systems so that the logical functionality is correct regardless of timing. For high-speed combinational circuits, however, over-designing for all worst-case circuit effects is not feasible. With deep sub-micron effects such as cross-talk and coupling, and with new circuit design styles, the timing and the logical behavior of a circuit have become very tightly coupled. In Chapter 7, we propose the use of timed automata for modeling and analyzing deep sub-micron combinational circuits. Verification with timed automata is currently restricted to small systems, but the structure of the delay computation problem lends itself to novel approaches. This, to the best of our knowledge, is the first mainstream application of timed-automaton-based techniques.

Chapter 8 concludes the dissertation and indicates future research directions.

Chapter 2

Timed Automata

When modeling digital real-time systems, the primary concern regarding the continuous dynamics of the system is the time elapsed between distinct events. Thus, it is appropriate for a mathematical model of such a system to have a discrete control structure with discrete valued inputs and outputs, and a mechanism to express timing constraints. This is achieved most conveniently and efficiently by augmenting finite automata with real-valued stopwatch variables. In the timed automaton model, these variables, called clocks, are used to measure and express constraints on the time elapsed between selected transitions. Timed automata have been introduced by Alur and Dill [AD94] and have been used widely afterwards. The timed automaton formalism is a natural extension of the finite automaton model to the continuous time domain and, as argued in [AD94][1], is at least as expressive as other formalisms for modeling real-time systems. The model used in this dissertation is a variant of timed automata, and is presented in the following sections.

2.1 Notation

Let X be a finite set of real-valued variables. An X-valuation Φ assigns a nonnegative real value $\Phi(x)$ to each variable $x \in X$. Let Φ be an X-valuation. For a real number $\delta \geq 0$, $\Phi + \delta$ denotes the X-valuation that assigns the value $\Phi(x) + \delta$ to each variable x, and $\mathbf{0}$ denotes the X-valuation that assigns the value 0 to all $x \in X$. For a subset $R \subseteq X$, $\Phi[R := 0]$ denotes the X-valuation that assigns the value 0 to each $x \in R$ and the value $\Phi(x)$ to each $x \notin R$. An X-predicate φ is a positive Boolean combination of constraints of

[1] An overview of other models for real-time systems can be found in the same article

the form $x \diamond k$, where k is a nonnegative integer constant, $x \in X$ is a variable, and $\diamond$ is one of the binary comparison relations: $\leq$, $\geq$, $=$. We write $\Phi \models \varphi$ if the valuation Φ satisfies the formula φ. Note that the set of X-valuations satisfying the X-predicate φ is closed.

Let P be a finite set of variables, each ranging over a finite type. A P-valuation $\mathbf{f}$ is an assignment of values to variables in P. For a P-valuation $\mathbf{f}$ and a subset $Q \subseteq P$, $\mathbf{f}(Q)$ denotes the Q-valuation obtained by the restriction of $\mathbf{f}$ to the variables in Q. This is called the *projection* of $\mathbf{f}$ onto Q. A P-event γ is a pair $\langle \mathbf{f}, \mathbf{f}' \rangle$ consisting of P-valuations, $\mathbf{f}$ and $\mathbf{f}'$ denoting the old and the new values of the variables in P. The projection of the event γ onto a set of variables $Q \subseteq P$ is given by $\langle \mathbf{f}(Q), \mathbf{f}'(Q) \rangle$. A P-predicate χ specifies a subset of P-events. While writing P-predicates as formulas, we use primed variables to refer to the updated values. For instance, the P-predicate $p' \neq p$ is satisfied by set of all P-events $\langle \mathbf{f}, \mathbf{f}' \rangle$ such that $\mathbf{f}'(p) \neq \mathbf{f}(p)$. We use $stutter(P)$ as an abbreviation for the predicate $\wedge_{p \in P}\, p' = p$.

2.2 Finite Automata and Languages

A *finite automaton* A is a tuple $\langle S, S_0, O, I, \alpha, \mu, E \rangle$, where

- S is the finite (nonempty) set of locations.
- $S_0 \subseteq S$ is the nonempty set of initial locations.
- O is the finite set of output variables, each ranging over a finite type. An *output* of A is a O-valuation.
- I is the finite set of input variables, each ranging over a finite type. It is required that I and O are disjoint. $I \cup O$ is called the *support* of the automaton. An *input* of A is an I-valuation, an *input-event* is an I-event, and an *input-predicate* is a I-predicate. An *observation* of A is a $(I \cup O)$-valuation, and an *observation-event* of A is a $(I \cup O)$-event.
- μ is the output function that assigns the output $\mu(s)$ to each location $s \in S$.
- E is the finite set of edges. Each edge e is a tuple $\langle s, t, \chi \rangle$ consisting of the source location s, the target location t, and the input-predicate χ.

An *(untimed) event sequence* $\overline{\gamma} = \gamma_0, \gamma_1, ..., \gamma_{k-1}$ is a finite sequence of observation events $\gamma_i = \langle \mathbf{f}_i, \mathbf{f}'_i \rangle$ such that $\mathbf{f}_{i+1} = \mathbf{f}'_i$ for $0 \leq i < k-1$. The projection of $\overline{\gamma}$ onto a set

of variables Q is obtained by replacing γ_i by $\langle \delta_i, \mathbf{f}_i(Q), \mathbf{f}'_i(Q) \rangle$ for all i. A *run* of the finite automaton A on an event sequence $\overline{\gamma}$ is a sequence of locations $s_0, s_1, s_2, ..., s_k$ such that $s_0 \stackrel{\gamma_0}{\rightarrow} s_1 \stackrel{\gamma_1}{\rightarrow} s_2 \stackrel{\gamma_2}{\rightarrow} ... \stackrel{\gamma_{k-1}}{\rightarrow} s_k$ in A. Such a run is said to be *initialized* if s_0 is an initial location. A location s is said to be *reachable* if there exists an initialized run leading to s. The event sequence $\overline{\gamma}$ is called a *trace* of A if there exists an initialized run in A on $\overline{\gamma}$. The *(untimed) language* of an automaton A, denoted $\mathcal{L}(A)$, is the set of traces of A.

The finite automaton A is *comparable* to B iff $O^B \subseteq O^A$ and $I^B \subseteq I^A$. Let A be comparable to B. A is said to *implement* B iff $\mathcal{L}(A)(I^B \cup O^B) \subseteq \mathcal{L}(B)$, where $\mathcal{L}(A)(I^B \cup O^B)$ denotes the projection of the language of A onto the input and output variables of A.stateThis relation is also referred to as *language inclusion.* A and B are said to be *language equivalent* (denoted $A \cong_L B$ iff $\mathcal{L}(A) = \mathcal{L}(B)$.

Composition (Product) of Finite Automata

The operator for building coordinating systems from their components is parallel composition. Consider two finite automata $A = \langle S^A, S_0^A, O^A, I^A, \mu^A, E^A \rangle$ and $B = \langle S^B, S_0^B, O^B, I^B, \mu^B, E^B \rangle$. The two automata A and B are *composable* iff their output variables O^A and O^B are disjoint. For two such composable finite automata, their parallel composition $A \parallel B$ is the automaton with the following components:

- The set S of locations is given by $S^A \times S^B$.
- The set S_0 of initial locations is $S_0^A \times S_0^B$.
- The set O of output variables is given by $O^A \cup O^B$.
- The set I of input variables is $(I^A \cup I^B) \setminus O$.
- For every $s \in S^A$ and $t \in S^B$, the output $\mu(\langle s, t \rangle)$ equals $\mu^A(s) \cup \mu^B(t)$.
- For every edge $e^A = \langle s^A, t^A, \chi^A \rangle$ in E^A and $e^B = \langle s^B, t^B, \chi^B \rangle$ in E^B, the set E contains the edge $e^{A\|B} = \langle \langle s^A, s^B \rangle, \langle t^A, t^B \rangle, \chi^{A\|B} \rangle$, where the I-event $\langle \mathbf{f}, \mathbf{f}' \rangle$ is in $\chi^{A\|B}$ iff $\langle \mathbf{f} \cup \mu^B(s^B), \mathbf{f}' \cup \mu^B(t^B) \rangle \models \chi^A$ and $\langle \mathbf{f} \cup \mu^A(s^A), \mathbf{f}' \cup \mu^A(t^A) \rangle \models \chi^B$. The edges e^A and e^B are the *projections* of $e^{A\|B}$ onto A and B respectively.

$A \parallel B$ is also referred to as the *(synchronous) product* of A and B. The parallel composition operator is commutative and associative with respect to language equivalence:

Proposition 2.2.1 *$A \parallel B \cong_L B \parallel A$, and $A \parallel (B \parallel C) \cong_L (A \parallel B) \parallel C$.*

2.3 Timed Automata

A *timed automaton* A is a tuple $\langle S, S_0, O, I, X, \alpha, \mu, E \rangle$, where

- S is the finite (nonempty) set of locations.
- $S_0 \subseteq S$ is the nonempty set of initial locations.
- X is the finite set of real-valued variables, called *clocks*.
- O is the finite set of output variables, each ranging over a finite type. An *output* of A is a O-valuation.
- I is the finite set of input variables, each ranging over a finite type. It is required that I and O are disjoint. $I \cup O$ is referred to as the *support* of A. An *input* of A is an I-valuation, an *input-event* is an I-event, and an *input-predicate* is a I-predicate. An *observation* of A is a $(I \cup O)$-valuation, and an *observation-event* of A is a $(I \cup O)$-event.
- α is the invariant function that assigns the X-predicate $\alpha(s)$ to each location $s \in S$.
- μ is the output function that assigns the output $\mu(s)$ to each location $s \in S$.
- E is the finite set of edges. Each edge e is a tuple $\langle s, t, \varphi, \chi, R \rangle$ consisting of the source location s, the target location t, the X-predicate φ, the input-predicate χ, and the set $R \subseteq X$ of clocks to be reset. It is required that (1) E contains the edge $\langle s, s, \mathit{true}, \mathit{stutter}(I), \emptyset \rangle$ for each location s, and (2) for given source and target locations, there is at most one edge between them[2].

2.4 Semantics of Timed Automata

2.4.1 States and Transitions

A *state* σ of a timed automaton A is a pair $\langle s, \Phi \rangle$ containing the location $s \in S$ and the X-valuation $\Phi \models \alpha(s)$. The set of all states is denoted Σ_A. The state $\langle s, \Phi \rangle$ is initial if $s \in S_0$ and $\Phi(x) = 0$ for all $x \in X$.

[2]This restriction simplifies the presentation of iterative abstraction methods and compositional proof rules, which will be described in Chapters 3-5.

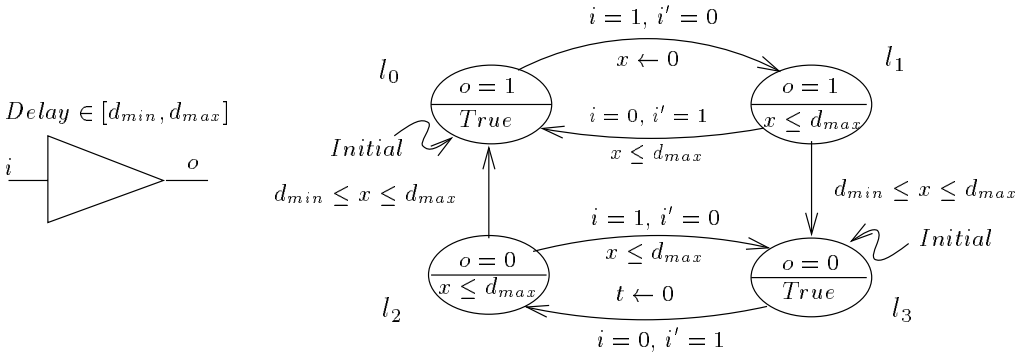

Figure 2.1: The timed automaton corresponding to an inertial delay buffer with non-deterministic delay between d_{min} and d_{max}. The self loops on locations corresponding to the stuttering edges have not been shown.

Consider a state $\sigma = \langle s, \Phi \rangle$ of the timed automaton A and a positive time increment δ. The automaton A can *wait* for δ in state σ, written $wait(\sigma, \delta)$, iff for all $0 \leq \delta' < \delta$, $(\Phi + \delta') \models \alpha(s)$. A *timed event* γ of the timed automaton A is a tuple $\langle \delta, \mathbf{f}, \mathbf{f}' \rangle$ consisting of a positive real-valued increment δ and the observation-event $\langle \mathbf{f}, \mathbf{f}' \rangle$. Such an event means that the automaton can wait for the time period δ and then update its output variables from $\mathbf{f}(O)$ to $\mathbf{f}'(O)$ while its environment is updating the input variables from $\mathbf{f}(I)$ to $\mathbf{f}'(I)$. The set of all timed events of A is denoted Γ_A. The projection of a timed event γ onto $Q \subseteq I^A \cup O^A$ is given by $\langle \delta, \mathbf{f}(Q), \mathbf{f}'(Q) \rangle$.

The timed automaton A defines a labeled transition system over the state-space Σ_A with the labels Γ_A. For states $\sigma = \langle s, \Phi \rangle$ and $\tau = \langle t, \Theta \rangle$ in Σ_A, and a timed event $\gamma = \langle \delta, \mathbf{f}, \mathbf{f}' \rangle$ in Γ_A, define $\sigma \xrightarrow{\gamma} \tau$ iff $\mathbf{f}(O) = \mu(s)$, $\mathbf{f}'(O) = \mu(t)$, $wait(\sigma, \delta)$, and there exists an edge $\langle s, t, \varphi, \chi, R \rangle$ such that $(\Phi + \delta) \models \varphi$, $\langle \mathbf{f}, \mathbf{f}' \rangle \models \chi$, and $\Theta = (\Phi + \delta)[R := 0]$. We write $\sigma \xrightarrow{\gamma}$ if $\sigma \xrightarrow{\gamma} \tau$ for some τ.

Proposition 2.4.1 (Closure under stuttering) *For $\gamma = \langle \delta, \mathbf{f}, \mathbf{f}' \rangle$, if $\sigma \xrightarrow{\gamma} \tau$ then for every $0 < \delta' < \delta$, there exists σ' with $\sigma \xrightarrow{\gamma'} \sigma'$ and $\sigma' \xrightarrow{\gamma''} \tau$ where $\gamma' = \langle \delta', \mathbf{f}, \mathbf{f} \rangle$ and $\gamma'' = \langle \delta - \delta', \mathbf{f}, \mathbf{f}' \rangle$.*

Example 2.4.2 Figure 2.1 depicts a timed automaton representing an inertial delay buffer with non-deterministic delay between d_{min} and d_{max}. The buffer is part of a combinational digital circuit, and has a Boolean input and output. According to the inertial delay model, an input pulse must have duration at least d_{min} to be reflected at the output, and any input pulse of duration more than d_{max} has to create a corresponding pulse at the output.

Suppose that the buffer starts at location l_0 representing the case where the input and output of the buffer are both 1 at power-up. When the input i goes low, the automaton responds by moving to l_1 and resetting clock x. x records the time that elapsed since i went low. In l_1 the output o is still high, but a change is pending. The automaton can move to l_3 when x is between d_{min} and d_{max} and at l_3 the output goes low. The use of x as a guard on the transition to l_3 ensures that the output follows the input within time $[d_{min}, d_{max}]$. If the input goes high again before the automaton moves to l_3, the automaton may move back to l_0. This represents the case when an input pulse lasting between d_{min} and d_{max} gets dropped by the buffer, i.e., it is not reflected at the output. If i goes high while $x < d_{min}$, the automaton moves to l_0, which models the fact that pulses lasting less than this minimum length are always dropped. ■

Observe that, in our model of timed automata, an automaton spends non-zero time in each location and transitions are instantaneous. The enabling condition on a transition can refer to the old (unprimed) as well as the new (primed) values of the inputs; this makes the use of transient (zero-time) states unnecessary.

2.4.2 Timed Languages

A *timed event sequence* $\overline{\gamma} = \gamma_0, \gamma_1, ..., \gamma_{k-1}$ is a finite sequence of timed events $\gamma_i = \langle \delta_i, \mathbf{f}_i, \mathbf{f}_i' \rangle$ such that $\mathbf{f}_{i+1} = \mathbf{f}_i'$ for $0 \leq i < k-1$. The projection of $\overline{\gamma}$ onto Q is obtained by projecting each timed event onto Q, i.e., replacing each γ_i by $\langle \delta_i, \mathbf{f}_i(Q), \mathbf{f}_i'(Q) \rangle$ for all i. For the timed event sequence $\overline{\gamma}$, define $\Delta_0 = 0$ and $\Delta_i = \sum_{j=0}^{i-1} \delta_j$ for $1 \leq i \leq k$. Each such $\overline{\gamma}$ uniquely defines a function $F_{\overline{\gamma}}$ from the interval $[0, \Delta_k)$ to the observations given by $F_{\overline{\gamma}}(t) = \mathbf{f}_i$ for $t \in [\Delta_i, \Delta_{i+1})$.

A *run* of A on a timed event sequence $\overline{\gamma}$ is a sequence of states $\sigma_0, \sigma_1, \sigma_2, ..., \sigma_k$ such that $\sigma_0 \stackrel{\gamma_0}{\rightarrow} \sigma_1 \stackrel{\gamma_1}{\rightarrow} \sigma_2 \stackrel{\gamma_2}{\rightarrow} ... \stackrel{\gamma_{k-1}}{\rightarrow} \sigma_k$ in A. Such a run is said to be *initialized* if σ_0 is an initial state. The timed event sequence $\overline{\gamma}$ is called a *trace* of A if there exists an initialized run in A on $\overline{\gamma}$. The *timed language* of an automaton A, denoted $\mathcal{L}(A)$, is the set of traces of A.

2.4.3 Language Inclusion

Consider two timed automata $A = \langle S^A, S_0^A, O^A, I^A, X^A, \alpha^A, \mu^A, E^A \rangle$ and $B = \langle S^B, S_0^B, O^B, I^B, X^B, \alpha^B, \mu^B, E^B \rangle$. The timed automaton A is *comparable* to B iff $O^B \subseteq O^A$ and $I^B \subseteq I^A$. Let A be comparable to B. A is said to *timed-implement* B iff for every

trace $\overline{\gamma}^A$ of A, there exists $\overline{\gamma}^B$ in $\mathcal{L}(B)$ assigning the same values to the input and output variables of B at all times, i.e., $F_{\overline{\gamma}^A}(I^B \cup O^B) = F_{\overline{\gamma}^B}$, where $F_{\overline{\gamma}}$ denotes the observation waveform as defined in Section 2.4.2. Timed implementation is denoted by $A \preceq_L B$ and is also referred to as the *language inclusion* relation. The two timed automata are said to be *timed equivalent*, written $A \cong_L B$, iff both $A \preceq_L B$ and $B \preceq_L A$.

Proposition 2.4.3 (Preorder) *The relation $\preceq_L$ is reflexive and transitive.*

Proposition 2.4.4 (Equivalence) *The relation $\cong_L$ is an equivalence relation.*

A *property* $\mathcal{P}$ is a set of timed event sequences. The timed automaton A is said to satisfy the property $\mathcal{P}$ if $\mathcal{L}(A) \subseteq \mathcal{P}$. Often, $\mathcal{P}$ is described over a subset of the observation variables $I^A \cup O^A$. Let $V^{\mathcal{P}}$ be the set of variables that $\mathcal{P}$ specifies the values of. Then, A satisfies $\mathcal{P}$ if $\mathcal{L}(A)(V^{\mathcal{P}}) \subseteq \mathcal{P}$, where $\mathcal{L}(A)(V^{\mathcal{P}})$ consists of the projections of the timed event sequences in $\mathcal{P}$ onto $V^{\mathcal{P}}$. Observe that, for a timed automaton B, if $B \preceq_L A$, then all properties that are satisfied by A are also satisfied by B.

2.5 Operations on Timed Automata

2.5.1 Composition (Product)

The operator for building complex automata from simpler components is parallel composition. Consider two timed automata $A = \langle S^A, S_0^A, O^A, I^A, X^A, \alpha^A, \mu^A, E^A \rangle$ and $B = \langle S^B, S_0^B, O^B, I^B, X^B, \alpha^B, \mu^B, E^B \rangle$. We assume that the clocks X^A and X^B are disjoint [3]. The two timed automata A and B are *composable* iff their output variables O^A and O^B are disjoint. For two such composable timed automata, their parallel composition $A \parallel B$ is the timed automaton with the following components:

- The set S of locations is given by $S^A \times S^B$.
- The set S_0 of initial locations is $S_0^A \times S_0^B$.
- The set O of output variables is given by $O^A \cup O^B$.
- The set I of input variables is $(I^A \cup I^B) \setminus O$.
- The set X of clock variables equals $X^A \cup X^B$.

[3] This can be achieved by renaming some clocks

- For every $s \in S^A$ and $t \in S^B$, the invariant $\alpha(\langle s,t\rangle)$ is given by the conjunction $\alpha^A(s) \wedge \alpha^B(t)$.
- For every $s \in S^A$ and $t \in S^B$, the output $\mu(\langle s,t\rangle)$ equals $\mu^A(s) \cup \mu^B(t)$.
- For every edge $e^A = \langle s^A, t^A, \varphi^A, \chi^A, R^A\rangle$ in E^A and $e^B = \langle s^B, t^B, \varphi^B, \chi^B, R^B\rangle$ in E^B, the set E contains the edge $e^{A\|B} = \langle\langle s^A, s^B\rangle, \langle t^A, t^B\rangle, \varphi^A \wedge \varphi^B, \chi^{A\|B}, R^A \cup R^B\rangle$, where the I-event $\langle \mathbf{f}, \mathbf{f}'\rangle$ is in $\chi^{A\|B}$ iff $\langle \mathbf{f} \cup \mu^B(s^B), \mathbf{f}' \cup \mu^B(t^B)\rangle \models \chi^A$ and $\langle \mathbf{f} \cup \mu^A(s^A), \mathbf{f}' \cup \mu^A(t^A)\rangle \models \chi^B$. The edges e^A and e^B are the *projections* of $e^{A\|B}$ onto A and B respectively.

$A \parallel B$ is also referred to as the *(synchronous) product* of A and B.

Observe that the parallel composition operator is commutative and associative:

Proposition 2.5.1 *$A \parallel B \cong_L B \parallel A$, and $A \parallel (B \parallel C) \cong_L (A \parallel B) \parallel C$.*

2.5.2 Variable Hiding

Variable hiding, also called existential quantification, is useful when it is desired to describe the behavior of a timed automaton over a subset of the input or output variables. This is typically the case when a number of components are combined to form a larger system, and the signals internal to the system are not observable from outside, or are not of interest.

Consider a timed automaton $A = \langle S^A, S_0^A, O^A, I^A, X^A, \alpha^A, \mu^A, E^A\rangle$. For an input variable $i \in I^A$, the automaton obtained by hiding i is denoted by $(\exists i)\ A$ and corresponds to existentially quantifying the i in all input-predicates on the edges. More precisely,

$$(\exists i)\ A = \langle S^A, S_0^A, O^A, I^A \setminus \{i\}, X^A, \alpha^A, \mu^A, (\exists i)\ E^A\rangle$$

where

$$(\exists i)\ E^A = \left\{ \langle s, t, \varphi, (\exists i, i')\ \chi, R\rangle \;\middle|\; \langle s, t, \varphi, \chi, R\rangle \in E^A \right\}$$

The existentially quantified predicate $(\exists i, i')\ \chi$ is formally defined as

$$(\exists i, i')\ \chi = \left\{ \langle \mathbf{f}(I^A \setminus \{i\}), \mathbf{f}'(I^A \setminus \{i\})\rangle \;\middle|\; \langle \mathbf{f}, \mathbf{f}'\rangle \in \chi \right\}$$

The hiding of an output variable $o \in O^A$ is achieved simply by restricting the output function μ to output variables other than o. Formally,

$$(\exists o)\ A = \langle S^A, S_0^A, O^A \setminus \{o\}, I^A, X^A, \alpha^A, \mu^A(O^A \setminus \{o\}), E^A\rangle$$

2.5.3 Untiming

Intuitively, the untiming operation corresponds to removing the timing information and constraints, and interpreting a timed automaton as a finite automaton. Formally, for a timed automaton $A = \langle S^A, S_0^A, O^A, I^A, X^A, \alpha^A, \mu^A, E^A \rangle$, $Untime(A)$ is the finite automaton $\langle S^A, S_0^A, O^A, I^A, \mu^A, E^{Untime(A)} \rangle$ where

$$E^{Untime(A)} = \{\langle s, t, \chi \rangle \mid \langle s, t, \varphi, \chi, R \rangle \in E^A \text{ for some } \varphi \text{ and } R\}$$

2.6 Verification with Timed Automata

2.6.1 Safety Properties

A major portion of interesting properties to be proven about timed automata fall into the *safety properties* category. Formally, a safety property is a language $\mathcal{P}$ such that if a timed event sequence $\overline{\gamma} \notin \mathcal{P}$, then all timed event sequences that are extensions $\overline{\gamma\eta}$ of $\overline{\gamma}$ also lie outside $\mathcal{P}$. Intuitively, once a safety property is violated at a certain point in time, it is not possible for the system to satisfy the property later, no matter what behavior it exhibits. A majority of interesting timing properties can be stated as safety properties, and in this dissertation we restrict our attention to safety properties only.

A convenient way of verifying a safety property of a system described as a timed automaton is to construct a monitor, also described as a timed automaton, which observes the inputs and outputs of the system and moves to a distinguished ("bad") location if undesired behavior occurs. The verification problem then reduces to checking whether in the composition of the system and the property automata it is possible for the monitor automaton to enter the "bad" location. This is a particular case of the *(location) reachability problem*, which is formally defined below.

Definition 2.6.1 (Reachability) *An instance of the reachability problem consists of a timed automaton* $A = \langle S^A, S_0^A, O^A, I^A, X^A, \alpha^A, \mu^A, E^A \rangle$ *and a set of locations* $T \subseteq S^A$. *T is said to be* reachable *in A if there exists a trace γ of A that leads to a state $\langle s, \Phi \rangle$ where $s \in T$.*

Example 2.6.2 Consider the train gate controller depicted in Figure 2.2 [Wu94, AIKY92]. This example consists of three automata representing a train, a gate and a controller. When

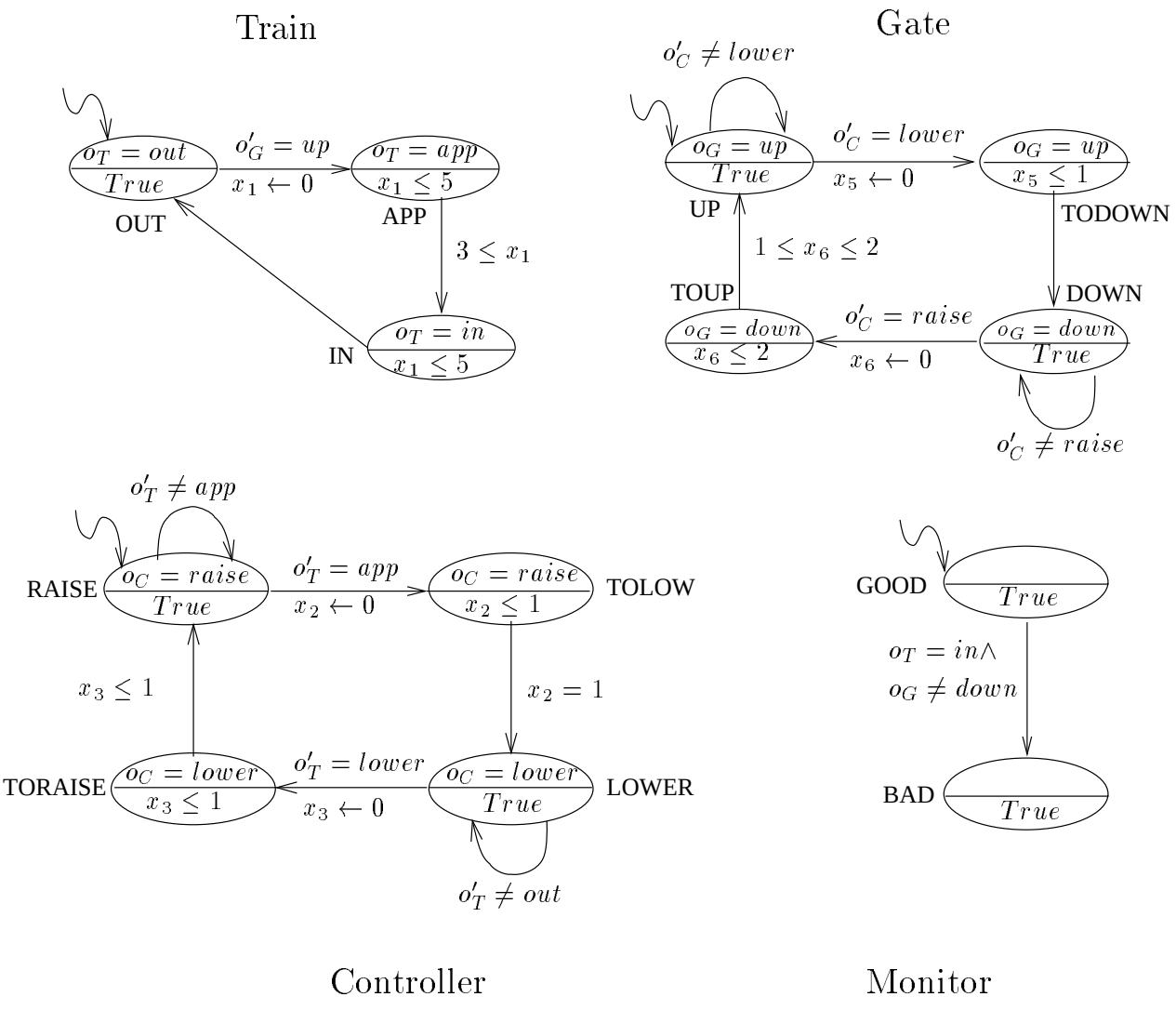

Figure 2.2: Train gate controller example. o_T, o_G and o_C denote the output variables of the train, the gate and the controller respectively. The input variables i_T, i_G and i_C are equal to o_G, o_C, and o_T respectively.

the train approaches the gate, the controller lowers the gate, and raises it after the train has exited. The property to be verified is that the gate is always down when the train is crossing the gate. This is achieved by creating a monitor that moves to location "BAD" when the train's output is *in* and the gate's output is different from "down". It must then be verified that the product locations where the monitor is at location "BAD" are not reachable because of timing constraints. ■

2.6.2 Region Equivalence

Recall that a state of a timed automaton $\langle s, \Phi \rangle$ involves a real-valued assignment to clock variables, Φ. The set of states Σ_A is therefore a dense, uncountably infinite set. Alur and Dill [AD90, Alu91, AD94] have shown that verification on this state space is tractable by defining an equivalence relation that has a finite quotient, called *region equivalence*, on this state space. Many interesting verification problems have then been converted (see, for instance, [Alu91], [Eme92], [Čerāns92], [TY96]) to computations on finite structures based on the equivalence classes of this relation. All algorithms to be presented in this dissertation employ some variant of this construction, and hence it is presented below.

Let A be a timed automaton and for each $x \in X$ let K_x denote the largest integer such that $x \diamond K_x$ appears in an X-predicate on some edge of A. Also let $Fr(x) = x - \lfloor x \rfloor$, the fractional part of x. The region equivalence [AD94] relation is defined as follows: clock valuations Φ and Θ are equivalent iff for all $x \in X$, either both $\Phi(x)$ and $\Theta(x)$ are larger than K_x, or the following hold

- $\lfloor \Phi(x) \rfloor = \lfloor \Theta(x) \rfloor$, and
- For all $x' \in X$ such that $\Phi(x') \leq K_{x'}$, $Fr(\Phi(x)) \geq Fr(\Phi(x'))$ iff $Fr(\Theta(x)) \geq Fr(\Theta(x'))$, and
- $Fr(\Phi(x)) = 0 \Leftrightarrow Fr(\Theta(x)) = 0$.

We write $\langle s, \Phi \rangle \equiv \langle t, \Theta \rangle$ iff $s = t$ and Φ and Θ are region equivalent.

Proposition 2.6.3 (Alur, Dill 1991) *$\equiv$ is an equivalence relation. If Φ and Θ belong to the same equivalence class, then for all clock predicates φ occuring on the edges of A, $\Phi \models \varphi$ if and only if $\Theta \models \varphi$.*

$[\Phi]_A$ denotes the equivalence class that Φ belongs to. The number of equivalence classes is exponential in the number of clocks, $|X|$, and the clock upper bounds K_x for $x \in X$ [AD94].

2.6.3 The Region Automaton

The region automaton $Region(A)$ of a timed automaton A is a finite automaton that encapsulates the location reachability information. For $A = \langle S^A, S_0^A, O^A, I^A, X^A, \alpha^A, \mu^A, E^A \rangle$, $\mathcal{R} = Region(A)$ is given as follows

- The set of locations $S^{\mathcal{R}} = \{\langle s, [\Phi]_A \rangle \mid \langle s, \Phi \rangle \in \Sigma_A\}$
- The set of initial locations $S_0^{\mathcal{R}} = \{\langle s_0, [\mathbf{0}]_A \rangle \mid s_0 \in S_0^A\}$
- The input and output variables and the output function remain unchanged, i.e., $I^{\mathcal{R}} = I^A$, $O^{\mathcal{R}} = O^A$, $\mu^{\mathcal{R}}(\langle s, [\Phi]_A \rangle) = \mu^A(s)$.
- If for some timed event γ, $\langle s, \Phi \rangle \xrightarrow{\gamma} \langle t, \Theta \rangle$ in A by taking an edge $\langle s, t, \varphi, \chi, R \rangle$, then $\langle \langle s, [\Phi]_A \rangle, \langle t, [\Theta]_A \rangle, \chi \rangle$ is in $E^{\mathcal{R}}$.

The soundness of the region automaton construction, as well as other properties that make it useful for verification, are proved in [Alu91]. The following fact is the one we will utilize in the rest of this dissertation.

Theorem 2.6.4 (Alur, 1991) *A location s is reachable in A iff it is reachable in the region automaton $Region(A)$.*

The verification tools ORBITS [Rok93], COSPAN [AK96], KRONOS [DOY94] and MOCHA [AHM+98] include timing verification algorithms based on the region construction. A BDD-based symbolic traversal or a depth-first search based on explicit state enumeration of the region automaton forms the core of these algorithms.

2.6.4 Difference Bound Matrices

The number of locations of the region automaton is exponential in the number of clocks in the timed automaton, as well as the integers used in clock predicates [Alu91]. Often the region automaton is quite large and it may not be feasible to explore a direct representation

of the region automaton, even with improved BDD-based or explicit techniques [ABH+97]. This is often the case if the upper bounds on the timers are large integers. A number of tools, including COSPAN [AK96], KRONOS [DOY94], and UPPAAL [LPY95], incorporate verification algorithms based on a particular symbolic representation, referred to as *difference bound matrix*. A difference bound matrix represents a convex union of regions in the following way: Let $x_1, ..., x_n$ be the clocks of a timed automaton A and let us adopt the convention that x_0 is always zero. A difference bound matrix D is an $(n+1) \times (n+1)$ matrix with integer entries, where the entry at the ith row and the jth column, $d_{i,j}$ represents the fact that $x_i - x_j \leq d_{i,j}$. The set of constraints that the matrix D implies define a convex region in $\mathbb{R}^n$ which is a union of regions. Traversal algorithms then use difference bound matrices as the data structure to represent a convex set of regions. While the the location reachability problem of timed automata is PSPACE-complete, the difference bound matrix representation may result in big efficiency gains if a large number of regions are equivalent with respect to the computation being performed.

Known algorithms for solving the location reachability problem have an exponential worst-case dependency on the number of clocks and the sizes of the constants appearing in clock predicates. This remains true even if the constants are encoded in unary [Alu91]. In practice, this problem exhibits itself as a more severe dependency of verification complexity on the timing constraints than the size of the control structure. Unless a system has a special structure that makes it more amenable for analysis using a particular data structure or algorithm, only tens of clocks can be handled simultaneously by state-of-the-art methods [DOTY96, BLL+96, ABH+97, BM98]. If modeled in complete detail at the implementation level, real-time systems encountered in practice often require many more clocks. Therefore, to be able to benefit from the complete coverage offered by timed automaton based techniques, they must be applied to abstracted models of systems. The aim of the techniques studied in this dissertation is to make the formal verification of practical real-time systems viable through the use of abstractions and hierarchical techniques.

Chapter 3

Iterative Abstraction Methods

Much of the time, not all of the timing information in a system description is necessary for proving a given property. Because of the exponential dependency of verification complexity on the number of clocks, it is desirable to include in a timed automaton only the clocks and clock predicates relevant to the property at hand. Iterative techniques [AIKY92, LB93, BSV95, Bal95b] achieve this by disregarding the timing information at first and then gradually including timing constraints as needed. At each iteration, only part of the timing information in the system is included in the system model, and thus, these models can be viewed abstractions of the original system. In this chapter, we give an overview of iterative abstraction methods and present our improvements to them.

Section 3.1 gives an overview of iterative verification methods. Sections 3.2 focuses on iterative methods based on the "simple path property": a structural property of timed automata that makes iterative verification less complex. Sections 3.2.1 and 3.2.2 present previous work, and Section 3.2.3 describes our method for exploiting the simple path property. In Section 3.3 we propose a method for making more efficient use of the information computed in each iteration of a timing verification algorithm. Section 3.4 presents our conclusions and indicates future research directions.

3.1 Timing Verification by Iteration

3.1.1 The Intuition

Let a real-time system be described by a timed automaton $S = \langle S^S, S_0^S, O^S, I^S, X^S, \alpha^S, \mu^S, E^S \rangle$ and suppose that a safety property $\mathcal{P}$ is expressed using a monitor $M = \langle S^M, S_0^M, O^M, I^M, X^M, \alpha^M, \mu^M, E^M \rangle$ as described in Section 2.6.1, such that M moves to location s_{bad}^M if the system S violates the safety property. Let S_{target} denote the locations in the product automaton $P = S \parallel M$ where $\mathcal{P}$ is violated, i.e., $S_{target} = \left\{ \langle s^S, s_{bad}^M \rangle \,\middle|\, s^S \in S^S \right\}$ We must determine whether S_{target} is reachable in P. The aim of the iterative approach to timing verification is to achieve this using a small subset of the timing information in P. Initially, we construct the finite automaton $Untime(P)$ and check if S_{target} is reachable in this automaton. This can be performed using any verification tool that operates on finite automata. If no location in S_{target} is reachable in $Untime(P)$, then S_{target} is clearly not reachable in P. If S_{target} is reachable in $Untime(P)$, then the verification tool returns an *error trace*: an initialized run $s_0, s_1, s_2, ..., s_k$ ending in a location $s_k \in S_{target}$ with an (untimed) event sequence $\overline{\gamma} = \gamma_0, \gamma_1, \gamma_2, ...$ such that $s_0 \stackrel{\gamma_0}{\rightarrow} s_1 \stackrel{\gamma_1}{\rightarrow} s_2 \stackrel{\gamma_2}{\rightarrow} ... \stackrel{\gamma_{k-1}}{\rightarrow} s_k$ in $Untime(P)$. Using this information, we would like to see if it is possible to construct a timed event sequence $\overline{\gamma^t} = \gamma_0^t, \gamma_1^t, \gamma_2^t, ...$ that takes P through the same sequence of locations while the input and output variables make the same sequence of transitions in $\overline{\gamma^t}$ as they do in $\overline{\gamma}$. Formally stated, each γ_i^t associates a duration with γ_i: if $\gamma_i = \langle \mathbf{f}_i, \mathbf{f}_i' \rangle$ then $\gamma_i^t = \langle \delta_i, \mathbf{f}_i, \mathbf{f}_i' \rangle$ for some $\delta_i \in \mathbb{R}^+$ and we require $\sigma_0 \stackrel{\gamma_0^t}{\rightarrow} \sigma_1 \stackrel{\gamma_1^t}{\rightarrow} \sigma_2 \stackrel{\gamma_2^t}{\rightarrow} ... \stackrel{\gamma_{k-1}^t}{\rightarrow} \sigma_k$ where $\sigma_i = \langle t_i, \Phi_i \rangle$ for some clock valuation Φ_i. Different iterative algorithms resolve the question of whether such an $\overline{\gamma^t}$ exists using different methods. If $\overline{\gamma^t}$ exists, then this is proof that $t_k \in S_{target}$ is reachable. If not, then we would like to remove the run $t_0 \stackrel{\gamma_0}{\rightarrow} t_1 \stackrel{\gamma_1}{\rightarrow} t_2 \stackrel{\gamma_2}{\rightarrow} ... \stackrel{\gamma_{k-1}}{\rightarrow} t_k$ from $Untime(P)$, since it does not correspond to a run in P. This is accomplished by composing $Untime(P)$ with a "constraining automaton" that observes its transitions. If we view $Untime(P)$ as our current abstraction for P, the constraining automaton in effect incorporates a portion of the timing information in P into $Untime(P)$ and thus refines it. Then, the procedure presented up to now is repeated until either S_{target} becomes unreachable in the finite automaton or a timed run leading to S_{target} is found. This style of verification will be presented more formally in the upcoming sections.

3.1.2 Preliminaries

Using iterative methods, it is sometimes possible to perform verification on a larger class of timed automata than those defined in Section 2.3. In the rest of this chapter, we will not place any restriction on the form of the clock predicates on the edges of a timed automaton. We allow a clock predicate φ to be any formula that evaluates to a Boolean value given a clock evaluation Φ, i.e., $\Phi \models \varphi$ can be interpreted in any decidable mathematical theory.

We proceed to define some terminology to aid in the presentation of the algorithms.

Definition 3.1.1 *We will use the shorthand $R(x_i)$ for a clock x_i being reset on an edge or a transition. For a set of clocks $x_{i_1}, x_{i_2}, ..., x_{i_k}$, $Q(x_{i_1}, x_{i_2}, ..., x_{i_k})$ denotes a condition (a* **query***) that depends on $x_{i_1}, x_{i_2}, ..., x_{i_k}$ that must be satisfied for a transition to be taken ($x_{i_1}, x_{i_2}, ..., x_{i_k}$ are said to be* **queried***). A* **symbol** *is a reset or a query.*

Example 3.1.2 We will use the train gate controller example of Section 2.6.1 for illustration in this chapter also. The example is repeated in Figure 3.1 for convenience. In the automaton for the train, on the edge from location OUT to APP, clock x_1 is reset, i.e. $R(x_1)$. On the edge from location APP to IN, x_1 is queried. Its value must satisfy $3 \leq x_1 \leq 5$, i.e., $Q(x_1) = \{3 \leq x_1 \leq 5\}$. Note that the query corresponding to this edge is the conjunction of the clock predicate on the edge taken and the invariant on the source location, since both of these must be satisfied. ■

Definition 3.1.3 *For a timed automaton A, a sequence of locations $\pi = s_0, s_1, ..., s_n$ connected by edges is called a* **path** *in A. The path π is said to be* **traversable** *iff there exists a timed event sequence $\overline{\gamma^t}$ such that $\sigma_0, \sigma_1, \sigma_2, ..., \sigma_n$ is a run of A on $\overline{\gamma^t}$, where for all i, $\sigma_i = \langle s_i, \Phi_i \rangle$ for some clock valuation Φ_i.*

An ordered sequence of distinct *locations $\sigma = t_0, t_1, ..., t_k$ is said to be* **traversed** *by a path $\pi = s_0, s_1, ..., s_n$ if σ is a subsequence of π, i.e., if it is possible to choose $0 \leq i_0 < i_2 < ... < i_k \leq n$ such that $\sigma = s_{i_0}, s_{i_1}, ..., s_{i_k}$. The sequence σ is said to be traversable if there exists a traversable path π that traverses σ.*

Example 3.1.4 In Figure 3.1, the path $\begin{pmatrix}\text{Train}\\ \text{Controller}\\ \text{Gate}\end{pmatrix} = \begin{pmatrix}\text{OUT}\\ \text{RAISE}\\ \text{UP}\end{pmatrix}, \begin{pmatrix}\text{APP}\\ \text{TOLOW}\\ \text{UP}\end{pmatrix}, \begin{pmatrix}\text{APP}\\ \text{LOWER}\\ \text{TODOWN}\end{pmatrix},$-

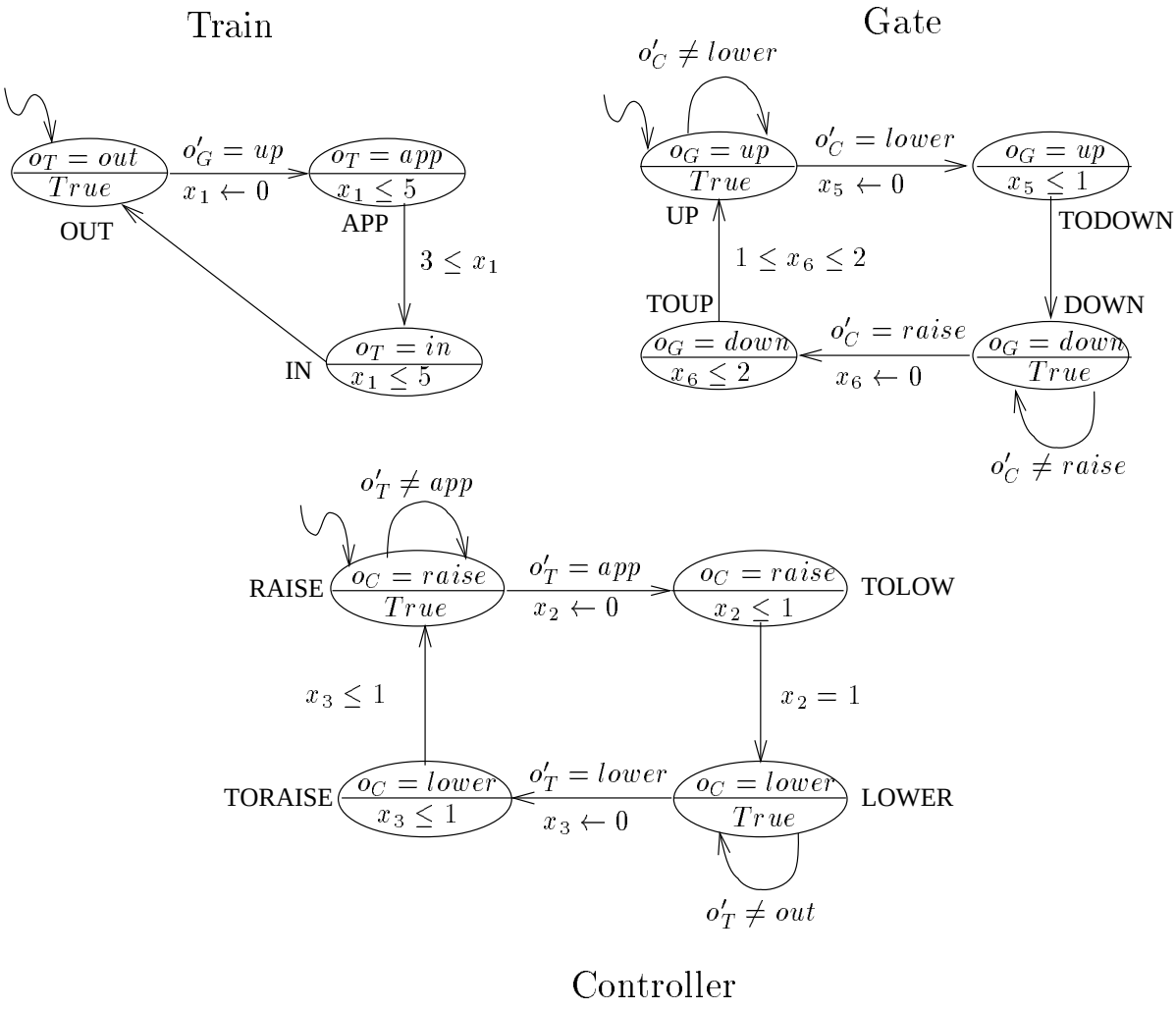

Figure 3.1: Train gate controller example.

$\begin{pmatrix} \text{IN} \\ \text{LOWER} \\ \text{TODOWN} \end{pmatrix}$ is not traversable, because in the first transition the clocks x_1 and x_2 are both reset, and $x_2 = 1$ for the second transition and $x_5 \leq 1$ for the third transition imply that less than 2 time units pass between the first and third transitions, whereas the condition $3 \leq x_1 \leq 5$ on the last transition contradicts this. ■

Definition 3.1.5 *Let $\pi = s_0 s_1 s_2 ... s_n$ be a path. The* **reset-query (RQ) sequence** *of π is the sequence $\Theta = \theta_0, \theta_1, ..., \theta_k$ of resets and queries along π such that*

- *Each θ_i is a non-empty set of symbols,*

- *Each θ_i consists of the resets and queries on one edge $(t_{j_i} \rightarrow t_{j_i+1})$ along π,*
- *For each transition with resets or queries on it, there is a corresponding θ_i, and*
- *Θ preserves the order of π, i.e., if θ_r and θ_s correspond to transitions $(t_{j_r} \rightarrow t_{j_r+1})$ and $(t_{j_s} \rightarrow t_{j_s+1})$ respectively, then $j_r < j_s$ iff $r < s$.*

$\theta_i \prec \theta_j$ denotes the fact that θ_i corresponds to an earlier edge along π than θ_j.

For the path in the train example, the RQ sequence is $\Theta = \theta_0, \theta_1, \theta_2$ where $\theta_0 = \{R(x_1), R(x_2)\}$, $\theta_1 = \{x_2 = 1, R(x_5)\}$ and $\theta_2 = \{3 \leq x_1 \leq 5, x_5 \leq 1\}$.

Definition 3.1.6
- *In an RQ sequence Θ, a reset of $R(x_i)$ of a clock x_i is said to be* **effective** *iff it is the last reset of x_i before a subsequent query involving x_i.*
- *The* **effective RQ sequence** *of a path π is obtained from the RQ sequence of π by removing the ineffective resets from the θ_i's, and then removing the empty θ_i's from the sequence.*

Let π be a path and $\Theta = \theta_0, ..., \theta_n$ its effective RQ sequence. Let $t_{i,j}$ be the time elapsed from the transitions corresponding to θ_i to the one corresponding to θ_j. Θ implies two sets of inequalities on the variables $t_{i,j}$:

(i) Those associated with queries: Suppose that clocks $x_{i_1}, ..., x_{i_k}$ are queried on θ_m, and let $last(x_i, m)$ denote the latest effective reset of x_i before θ_m. The clock valuation that assigns to each x_{i_j} the value $t_{last(x_{i_j},m),m}$ must satisfy the query θ_m.

(ii) Those indicating the ordering of the θ's and the fact that at each location a non-zero amount of time needs to be spent: $t_{i,j} > 0$ iff $\theta_i \prec \theta_j$.

Theorem 3.1.7 *Let $Ineq(\pi)$ denote the set of inequalities (i) and (ii) above. Suppose that a path π is traversable in $Untime(A)$. Then π is traversable in A iff it is possible to choose $t_{i,j}$'s that satisfy $Ineq(\pi)$.*

If $t_{i,j}$ satisfying $Ineq(\pi)$ exist, they are referred to as a *consistent timing* for π and π is said to admit a consistent timing.

$U_0 \leftarrow Untime(A)$, $i \leftarrow 0$, $S^{(i)}_{target} \leftarrow S_{target}$
repeat
    $IsReachable \leftarrow Check\_Reachable(U_i, S^{(i)}_{target})$
        if ($IsReachable$ = FALSE)
        Property passes, exit.
    else
        $\pi \leftarrow Obtain\_Error\_Trace(U_i, S^{(i)}_{target})$
        $IsTraversable \leftarrow Check\_Traversable(\pi)$
        if ($IsTraversable$ = TRUE)
            Property fails.
            return $Construct\_Consistent\_Timing(\pi)$ and exit.
        else
            $C_i \leftarrow Construct\_Constraining\_Automaton(U_i, \pi)$
            $U_{i+1} \leftarrow (U_i \parallel C_i)$
            $S^{(i+1)}_{target} \leftarrow (S^{(i)}_{target} \times S^{C_i})$
            $i \leftarrow i + 1$

Figure 3.2: The structure of an iterative timing verification algorithm

3.1.3 A Generic Iterative Algorithm

Using the facts presented in the previous two sections, we lay out the structure of an iterative timing verification algorithm in Figure 3.2. $Check\_Reachable(U_i, S^{(i)}_{target})$ checks whether the set of locations $S^{(i)}_{target}$ is reachable in the finite automaton U_i. Any reachability algorithm that operates on finite automata can be used to implement $Check\_Reachable()$. Most finite automaton verification tools also provide the $Obtain\_Error\_Trace()$ function. The algorithms $Check\_Traversable()$ and $Construct\_Consistent\_Timing()$ depend on the types of inequalities in $Ineq(\pi)$. For instance, if $Ineq(\pi)$ consists of linear inequalities, $Check\_Traversable(\pi)$ must determine if this linear program has a feasible solution and, if one exists, $Construct\_Consistent\_Timing()$ constructs one. The $Construct\_Constraining\_Automaton()$ procedure has the largest effect on the overall efficiency of the iterative algorithm.

In the following section, we define a structural property on timed automata that makes the $Check\_Reachable()$ routine more efficient and that enables $Construct\_Constraining\_Automaton()$ to eliminate more paths from U_i than π alone.

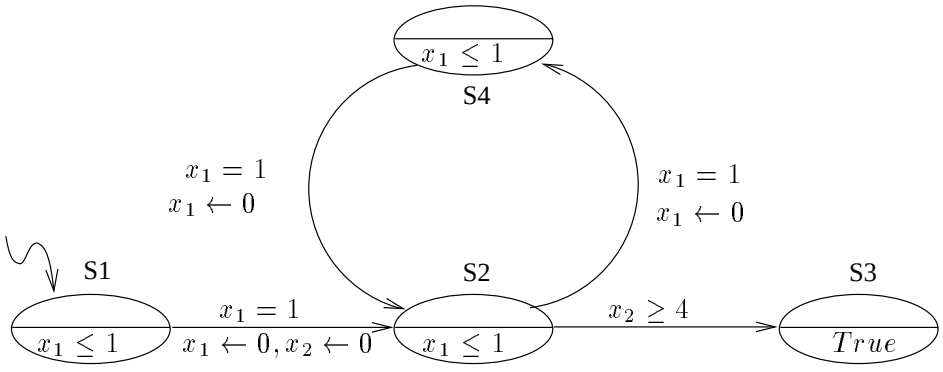

Figure 3.3: A timed automaton that does not have the simple path property.

3.2 Iterative Verification and The Simple Path Property

3.2.1 The Simple Path Property

Definition 3.2.1 *A path $\pi = s_0, s_1, ..., s_n$ is* **simple** *if π visits each location no more than once, i.e., $s_i = s_j$ implies $i = j$.*

A **cycle** *is a path $\pi = s_0, s_1, ..., s_n$ that starts and ends at the same location: $s_n = s_0$. The cycle π is said to be* **simple** *if for non-zero $i \neq j$, $s_i \neq s_j$.*

Definition 3.2.2 *A timed automaton is said to have the* **simple path property** *(SPP) if the following holds for every ordered sequence of distinct locations $\sigma = t_0, ..., t_n$: σ is traversable by a path if and only if it is traversable by a simple path.*

Example 3.2.3 The timed automaton in Figure 3.3[1] does not have the simple path property: The path S1 $\rightarrow$ S2 $\rightarrow$ S3 is not traversable, because when the automaton moves to S2, $x_2 = 1$ and $x_1 = 0$, and the first time S2 is $x_1 = 1$ and $x_2 = 2$. Thus, the clock predicate $x_2 \geq 4$ is not satisfied. But each time around the loop S2 $\rightarrow$ S4 $\rightarrow$ S2, 2 time units elapse. Therefore, the path S1 $\rightarrow$ S2 $\rightarrow$ S4 $\rightarrow$ S2 $\rightarrow$ S4 $\rightarrow$ S2 $\rightarrow$ S3, which is obtained from the initial path by inserting a cycle, is traversable. If the clock predicate $x_2 \geq 4$ were removed, then this automaton would have the SPP. ■

The SPP can be viewed as a restriction on the topology of the timed automaton and where the timing constraints can be placed. This restriction makes it simpler to perform

[1] This example is modified from an example in [LB94a].

iterative timing verification [LB94a] For automata with the SPP, the procedures *Check_Reachable*() and *Obtain_Error_Trace*() in the iterative algorithm of Section 3.1.3 need to consider only simple paths that lead to S_{target}. Note that, since there are only a finite number of simple paths in an automaton, the problem of reachability for timed automata possessing the SPP is decidable regardless of the form of the timing constraints. The only requirement is that the set of inequalities $Ineq(\pi)$ be decidable for all simple paths π. For instance, linear inequalities among multiple clocks can be allowed as clock predicates. This, in general, is not true for timed automata[Alu91].

We now describe how the SPP can be taken advantage of while implementing *Construct_Constraining_Automaton*() in order to eliminate an untraversable path π.

Definition 3.2.4

- *A path π' is a* **cycle expansion** *of a path π if it is obtained by replacing some locations t_i along π by (possibly non-simple) cycles starting and ending on t_i.*

- *An RQ sequence Θ is said to* **dominate** *Θ' iff Θ can be obtained from Θ' through the following sequence of transformations*

  - *Remove all symbols in Θ' that are not in Θ.*
  - *Remove some queries from Θ'.*
  - *Remove all non-effective resets.*
  - *Remove all empty θ_i''s.*

  The sequence of transformations can be viewed as a subsequence of the projection of Θ' on the resets and queries in Θ, so that the ordering of these symbols implied by Θ' is consistent with Θ. The dominance relation is transitive.

- *Let the effective RQ sequence of a path π be Θ and let π' be obtained from π by cycle expansion or removal, and let Θ' be the effective RQ sequence of π'. The RQ order of π is said to be* **preserved** *by this expansion or removal iff Θ dominates Θ'.*

In [LB94a] the following characterization for the SPP is given.

Theorem 3.2.5 (Lam, Brayton 1994) *If the queries in a timed automaton can refer to an arbitrary subset of the clocks, then the automaton has the SPP iff the RQ order of each simple path is preserved under cycle expansion.*

Certain classes of timed automata restrict the clocks that the clock predicates can depend on. For instance, the timed temporal structures of [KR97] allow predicates to specify upper and lower bounds on one clock only. The purpose of the phrase "arbitrary subset of the clocks" in the theorem statement is to rule out such classes of automata. To see how it is possible for the RQ order of a simple path not to be preserved under cycle expansion, consider to the following example.

Example 3.2.6 In the timed automaton of Example 3.2.3, the path $\pi_1 = \text{S1} \rightarrow \text{S2} \rightarrow \text{S4} \rightarrow \text{S2} \rightarrow \text{S4} \rightarrow \text{S2} \rightarrow \text{S3}$ is a cycle expansion of the simple path $\pi_2 = \text{S1} \rightarrow \text{S2} \rightarrow \text{S3}$. The RQ sequence for π_1 is $\Theta_1 = \{R(x_1), R(x_2), x_1 = 1\}$, $\{R(x_1), x_1 = 1\}$, $\{R(x_1), x_1 = 1\}$, $\{R(x_1), x_1 = 1\}$, $\{R(x_1), x_1 = 1\}$, $\{R(x_1), x_1 \leq 1, x_2 \geq 4\}$ and the RQ sequence for π_2 is given by $\Theta_2 = \{R(x_1), R(x_2), x_1 = 1\}$, $\{R(x_1), x_1 \leq 1, x_2 \geq 4\}$. For Θ_1 the effective reset for the query $x_1 \geq 1$ on the last transition is on the same transition where x_2 is reset, which is not the case for Θ_2. Therefore, the RQ sequence of π_2 is not preserved under cycle expansion. Recall that this automaton does not have the SPP. ■

In the rest of this chapter, we will assume that the queries in a timed automaton can refer to an arbitrary subset of the clocks, and make implicit use of Theorem 3.2.5.

3.2.2 Previous Approaches

In this section, we describe how the SPP has been exploited in the iterative timing verification algorithm of [LB94a]. In [LB94a] the following theorem was proven:

Theorem 3.2.7 (Lam, Brayton 1994) *Let Θ_1 and Θ_2 be effective RQ sequences such that Θ_1 dominates Θ_2. Then the set of inequalities I_{π_2} induced by π_2 is a superset of I_{π_1}. Therefore, if I_{π_1} is unsatisfiable, then so is I_{π_2}.*

Example 3.2.8 Let $\pi = t_0 t_1 t_2 t_3$ denote the following path in the train gate controller example of Section 3.2.1:

$$\begin{pmatrix} \text{Train} \\ \text{Controller} \\ \text{Gate} \end{pmatrix} = \begin{pmatrix} \text{OUT} \\ \text{RAISE} \\ \text{UP} \end{pmatrix}, \begin{pmatrix} \text{APP} \\ \text{TOLOW} \\ \text{UP} \end{pmatrix}, \begin{pmatrix} \text{APP} \\ \text{LOWER} \\ \text{TODOWN} \end{pmatrix}, \begin{pmatrix} \text{IN} \\ \text{LOWER} \\ \text{TODOWN} \end{pmatrix}$$

Consider $\pi' = t_0 t_1 t_0 t_1 t_2 t_3$, the following cycle expansion of π:

$$\begin{pmatrix}\text{Train}\\ \text{Controller}\\ \text{Gate}\end{pmatrix} = \begin{pmatrix}\text{OUT}\\ \text{RAISE}\\ \text{UP}\end{pmatrix}, \begin{pmatrix}\text{APP}\\ \text{TOLOW}\\ \text{UP}\end{pmatrix}, \begin{pmatrix}\text{OUT}\\ \text{RAISE}\\ \text{UP}\end{pmatrix}, \begin{pmatrix}\text{APP}\\ \text{TOLOW}\\ \text{UP}\end{pmatrix}, \begin{pmatrix}\text{APP}\\ \text{LOWER}\\ \text{TODOWN}\end{pmatrix}, \begin{pmatrix}\text{IN}\\ \text{LOWER}\\ \text{TODOWN}\end{pmatrix}$$

Recall that the RQ sequence for π is given by $\Theta = \theta_0, \theta_1, \theta_2 = \{R(x_1), R(x_2)\}$, $\{x_2 = 1, R(x_5)\}$, $\{3 \leq x_1 \leq 5, x_5 \leq 1\}$. The RQ sequence of π' is $\Theta' = \theta_0, \theta_0, \theta_1, \theta_2$. Observe that the effective RQ sequence of π' is the same as that of π. Thus, Θ' is dominated by Θ. This, together with the fact that π is not traversable implies that π' is not traversable either, by Theorem 3.2.7. ■

Theorems 3.2.5 and 3.2.7 imply that, given an automaton with the SPP and a path in the automaton known to be non-traversable because of timing constraints, all cycle expansions of that path are also not traversable. Using this fact, *Obtain_Error_Trace*() can be restricted to return a path whose projection onto A is a simple path, and *Construct_Constraining_Automaton*(U_i, π) can eliminate π along with all of its cycle expansions from U_i.

[LB94a] and [Wu94] have implemented an iterative timing verification algorithm in the Berkeley verification tool HSIS based on these observations. The version implemented in HSIS allows arbitrary linear inequalities between clocks. Moreover, the exponential dependency of verification complexity on the timing constants is eliminated[2]. However, this algorithm also suffers from the following shortcomings:

- A priori knowledge that the automaton has the SPP is required. It is complex to determine automatically whether an automaton has the SPP. The known conditions that are sufficient for the SPP are either too restrictive or require a complicated proof or an exponential algorithm, which must be repeated for each example [Wu94, LB93, LB94a].

- The SPP may not be preserved under composition of automata. Further restrictions need to be imposed on the placement of timing constraints on the automaton to ensure the SPP upon composition [Wu94]. Although with some additional effort it is possible to express most real-time systems in accordance to these restrictions, in practice they are often too prohibitive and make it inconvenient to specify systems.

[2]Recall that the size of the region automaton is exponential in the sizes of the constants that clocks get compared with (Section 2.6.2).

In the next section, we present a new algorithm that addresses these difficulties.

3.2.3 A Novel Method for Exploiting the SPP

The following observation formed the basis of a method that overcomes the problems with having to check for the SPP. Suppose that we would like to perform iterative verification on a timed automaton that does not necessarily have the SPP and we have an error trace π that admits no consistent timing, which we would therefore like to eliminate. Let Θ be the effective RQ sequence of π. Assume that we have a way of eliminating all paths π' whose effective RQ sequences are dominated by Θ. By Theorem 3.2.7, regardless of whether the automaton has the SPP, all such π' are untraversable, therefore can be eliminated. If the automaton does have the SPP, this will include all cycle expansions of π by Theorem 3.2.5. Therefore, eliminating all such π' will make full use of the SPP.

Note that this method may enable the elimination of more than one path even if the automaton does not have the SPP. For example, it could be the case that some cycle expansions of a certain path π are dominated by π's effective RQ sequence although the automaton in general does not have the SPP. The approach described in the preceding paragraph will eliminate all these cycle expansions.

Since an automaton has a finite number of simple paths, this approach is a complete decision procedure for automata with the SPP. For iterative algorithms that operate by eliminating one path a time, such as [AIKY92], replacing the elimination step by the algorithm above only improves the algorithm. Note that this algorithm can be applied to timed automata that can not be handled by existing iterative algorithms also. However, in general, there is no termination guarantee, and the algorithm becomes a semi-decision procedure. This is unavoidable, since the verification problem becomes undecidable unless either the topology or the form of the timing constraints are restricted [Alu91].

The next section formalizes these ideas, and presents a construction for eliminating all paths whose effective RQ sequences are dominated by a given Θ.

3.2.4 Elimination of Dominated RQ Sequences

Suppose that we are a given a finite automaton A and we would like to eliminate paths whose effective RQ sequences are dominated by the RQ sequence $\Theta = \theta_0 ... \theta_n$. We achieve this by constructing a constraining finite automaton M_Θ which, when composed with A,

disallows all such paths. M_Θ's input variable ranges over the set of resets and queries of A and it has one location for each $i \in \{0, 1, ..., n\}$, denoted by p_i.

The intuition underlying the construction is the following. M_Θ observes the resets and queries performed by A, and keeps track of the longest prefix of Θ that dominates the current effective RQ sequence, Θ_{cur}. If M_Θ is at location p_i, this means that Θ_{cur} is dominated by $\theta_0...\theta_{i-1}$. At each transition, M_Θ checks whether a longer prefix of Θ dominates Θ_{cur}, if not, it checks if $\theta_0...\theta_{i-1}$ continues to dominate Θ_{cur}, otherwise moves to an earlier location p_i corresponding to the longest prefix of Θ dominating Θ_{cur}.

The construction of M_Θ is rather technical. The following definitions will be useful in the presentation.

Definition 3.2.9 *A reset $R(x_k) \in \theta_j$ is said to be* **effective on** *θ_l iff $j \leq l$ and θ_j includes the effective reset of $R(x_k)$ for some query on θ_{l+1} or later.*

In the following, we say that a reset $R(x_i)$ is *observed* on an edge if x_i gets reset on that edge. A query $\psi()$ is said to be *observed* on an edge if it is the query associated with it.

Definition 3.2.10 *For each θ_i in Θ, let γ_i denote the following boolean conditions about a given edge e in the timed automaton:*

$\alpha_i =_{def}$ *All the resets and queries in θ_i are observed on e,* and
$\beta_i =_{def}$ *None of the $R(x_k)$'s effective on θ_i but not in θ_i are observed on e.*

Note that this definition allows for some extra queries and irrelevant resets to be observed on e.
M_Θ has the following five groups of edges

(I) From each location p_i with $i > 0$, there is an edge to itself on the condition γ_{i-1}.

(II) M_Θ moves from p_i to p_{i+1} if γ_i is true.

(III) From each location p_i, there is an edge to itself on the following condition, denoted by ρ_i

$\overline{\alpha_i}$: Some of the resets and queries in θ_i are not observed, *and*
β_i: None of the $R(x_k)$'s not in θ_i are observed if $R(x_k)$ is effective on θ_i.

This edge represents the fact that γ_i has not been observed, and all of the previous resets that are currently effective remain effective.

(IV) Suppose that while at location p_i the symbols observed make γ_{k-1} true for some $k < i$, where γ_{k-1} includes a reset $R(x_j)$ that is still effective on θ_i.

Claim 3.2.11 *There can be only one such k.*

Proof: Assume that k_1 and k_2 are as mentioned above and $k_1 < k_2$. Since γ_{k_1-1} and γ_{k_2-1} are both observed, γ_{k_1-1} must have all the effective resets of γ_{k_2-1}, but may differ from it in the queries that it includes. But then the resets in γ_{k_1-1} are not effective, since γ_{k_2-1} comes after it while the resets in γ_{k_1-1} are still effective. This contradicts the fact that γ_{k_1-1} includes a reset that is still effective on θ_i. ■

This means that $\theta_0...\theta_{i-1}$ no longer dominates the current effective RQ sequence and we must go back to the earlier location p_k. M_Θ makes a transition to p_k in this case.

(V) Suppose that none of the conditions in cases **I - IV** holds and while at location p_i some reset $R(x_k)$ is observed while some previous $R(x_k) \in \theta_{l_k}$ is still effective on θ_i. Note that such x_k does not have to be unique. Let k be such that l_k is minimized, i.e., let us consider the earliest θ_{l_k} that is rendered ineffective. M_Θ makes a transition to p_{l_k}.

M_Θ has no edges out of p_n, which means no run exists in $A \times M_\Theta$ if the effective RQ sequence of the path traversed in A is dominated by $\Theta = \theta_0...\theta_n$, assuming the construction above is correct. An example of the construction of M_Θ can be found in Figure 3.4.

The following theorem formalizes the argument above and proves the correctness of the construction.

Theorem 3.2.12 *If M_Θ is at location p_j after the kth transition of a run of $A \times M_\Theta$, then Ω_k, the effective RQ sequence of the run in A so far, is dominated by $\theta_0...\theta_{j-1}$, where j is the maximum such index.*

Proof: The proof is by induction on k.

- $k = 0$. Since no transitions have been taken, M_Θ is in location p_0 and the empty sequence is the longest prefix of Θ that dominates $\Omega = \emptyset$.

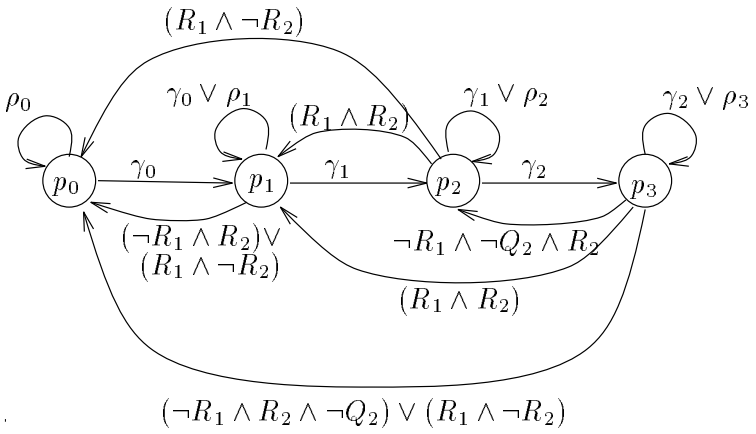

Figure 3.4: Construction of M_Θ for $\Theta = (\Theta_0, \Theta_1, \Theta_2, \Theta_3) = (\{R_1, R_2\}, \{Q_1(x_1)\}, \{Q_2(x_1, x_2), R_2\}, \{Q_3(x_1, x_2)\})$, where R_i denotes a reset of clock x_i and $Q_j(x_i)$ denotes a query depending on x_i.

- Assume that M_Θ being at location p_j after m transitions, where $m \leq k$, implies that $\theta_0 ... \theta_{j-1}$ is the longest prefix of Θ that dominates Ω_m.

- Let M_Θ be at locations p_j and p_r after the kth and $(k+1)$st transitions respectively. By the induction hypothesis, $\theta_0 ... \theta_{j-1}$ is the longest prefix of Θ that dominates Ω_k.

  Lemma 3.2.13 *If $\theta_0 ... \theta_j$ and $\theta_0 ... \theta_l$ are the longest prefixes of Θ that dominate Ω_k and Ω_{k+1} respectively, l can be at most $j+1$.*

  Proof: Assume towards a contradiction that $l \geq j+2$. Since $l > j$, the symbols observed during the $(k+1)$st transition must satisfy γ_l. Then $\theta_0 ... \theta_{l-1}$ is also a prefix of Θ that dominates Ω_k, which contradicts the maximality of j. ■

  Armed with this lemma, let us proceed with the induction.

Case 1. $p_r = p_j$.

Then the last transition taken was a self-loop. This transition is implied by either **I** or **III** above.

Case a. The $(k+1)$st transition was implied by **I**.

Then γ_{j-1} held during the last transition. Thus $\theta_0 ... \theta_{j-1}$ still satisfies the requirements of the theorem, since, the θ_{j-1} observed merely replaces the previous one.

Case b. The $(k+1)$st transition was implied by **III**.

Since some of the resets and queries on θ_j were not observed, Ω_{k+1} can not

be dominated by $\theta_0...\theta_j$. However, since none of the resets that were effective on θ_j have been overridden by the ones observed on transition $k+1$, they remain effective, and thus, $\theta_0...\theta_{j-1}$ continues to dominate Ω_{k+1}. Therefore, $\theta_0...\theta_{j-1}$ satisfies the requirements in the theorem.

Case 2. $r = j+1$.

Then the $(k+1)$st transition was implied by **II**, i.e., γ_j held during the $(k+1)$st transition. Since none of the resets that are effective on θ_j that are not on θ_j were observed, $\theta_0...\theta_{j-1}$ dominates Ω_{k+1}. Moreover, since all the resets and queries on θ_j were observed, $\theta_0...\theta_j$ dominates Ω_{k+1}. The maximality of j follows from Lemma 3.2.13.

Case 3. $r < j$.

Case a. The $(k+1)$st transition was implied by **IV**.

This means γ_{r-1} was observed on the $(k+1)$st transition, and it includes a reset that is still effective on γ_j.

- **Fact:** $\theta_0...\theta_{r-1}$ *dominates* Ω_{k+1}.

  To see this, consider the last time the automaton was at location p_r. All the transitions taken after that point, except for the $(k+1)$st one, do not change the fact that $\theta_0...\theta_{r-1}$ dominates the current Ω, by the induction hypothesis. Since the $(k+1)$st transition merely repeats γ_{r-1}, $\theta_0...\theta_{r-1}$ continues to dominate Ω_{k+1}.

- **Fact:** $\theta_0...\theta_{r-1}$ *is the maximum such prefix.*

  Assume otherwise, i.e., let $\theta_0...\theta_{p-1}$ be the longest such prefix, with $p > r$. Since γ_{r-1} was true during the $(k+1)$st transition, the resets observed on the previous θ_{r-1} are no longer effective, therefore, $\theta_0...\theta_{p-1}$ contains ineffective resets. This is a contradiction.

Case b. The $(k+1)$st transition was implied by **V**.

Let x_k be the clock minimizing l_k where $R(x_k) \in \theta_{l_k}$ is observed on the $(k+1)$st transition while still effective. Then $r = l_k$.

- **Fact:** $\theta_0...\theta_{r-1}$ *dominates* Ω_{k+1}.

  To see this, consider the last time the automaton was at location p_r. By the induction hypothesis, all the transitions taken after that point until the $(k+1)$st one do not alter the fact that $\theta_0...\theta_{r-1}$ dominates the

current Ω, by the induction hypothesis. The $(k+1)$st transition does not include any resets that are on $\theta_0 ... \theta_{r-1}$ that are still effective, by the minimality of l_k. Therefore, $\theta_0 ... \theta_{r-1}$ dominates Ω_{k+1}.

- **Fact:** $\theta_0 ... \theta_{r-1}$ *is the maximum such prefix.*
  Assume otherwise, i.e., let $\theta_0 ... \theta_{p-1}$ be the longest such prefix, with $p > r$. Since $R(x_k)$ was observed on $(k+1)$st transition, $\theta_r = \theta_{l_k}$ contains a reset that is not effective. Thus, $\theta_0 ... \theta_{p-1}$ contains an ineffective reset, a contradiction.

Cases 1,2 and 3 cover all edges of M_Θ. Thus, the statement of the theorem holds for $k+1$. Therefore, the theorem holds for all k. ■

Compute Θ_π the effective RQ sequence of π.
Compute a minimal RQ sequence $\Theta_{\pi,min}$ such that
- $\Theta_{\pi,min}$ dominates Θ_π, and
- $Ineq(\Theta_{\pi,min})$ is not satisfiable.

return $M_{\Theta_{\pi,min}}$.

Figure 3.5: Algorithm for $Construct\_Constraining\_Automaton(U_i, \pi)$.

The construction given, along with Theorem 3.2.12 proves that $Construct\_Constraining\_Automaton(U_i, \pi)$ can be correctly implemented as shown in Figure 3.5. A procedure for obtaining $\Theta_{\pi,min}$ from Θ_π was presented in [Wu94]. The iterative verification algorithm of Figure 3.2 in conjunction with the subroutine in Figure 3.5 is a complete decision procedure for automata with the SPP. This is because each time an untraversable simple path is found, all of its cycle expansions are eliminated by $M_{\Theta_{\pi,min}}$ as well, which means it suffices to explore the finitely many simple paths. Note that the construction presented provides an alternative, and possibly more powerful, way to eliminate a given error trace. Therefore, for any iterative algorithm that eliminates one path at a time, replacing the elimination step by the algorithm described in this section will not affect correctness. Thus, the algorithm above is a complete decision procedure for timed automata for which such iterative algorithms are guaranteed to converge. As argued previously, for more general constraints, no termination guarantee can be proven, since the problems themselves may be undecidable [Alu91]. It may be possible, however, to prove termination for certain other special cases.

We have implemented this procedure in the verification tool HSIS. The performances

Example	[LB93]		[Wu94]		[BSV95]		[TB95]	
	Time	Iter.	Time	Iter.	Time	Iter.	Time	Iter.
Railroad (safety)	2.6	3	1.7	N/A	1.4	2	1.4	2
Railroad (bounded liveness)	175.6	11	25.6	N/A	8.7	11	12.3	9
Plane	1.5	3	2.8	N/A	0.9	3	1.0	3

Table 3.1: Run times on a DECstation 5000/260 and number of iterations for four iterative algorithms implemented in HSIS.

of the several algorithms are compared in Table 3.1 on the example systems presented in [Wu94]. The first two examples are a safety and a bounded liveness property [3] of a train gate controller; the last example is a plane landing system, all described in detail in [Wu94]. All algorithms were implemented on the HSIS verification tool. The first algorithm (the column marked [LB93]) is an iterative timing verification algorithm implemented in HSIS that operates on "alternating R/Q timed automata":a restricted class of timed automata for which the SPP is guaranteed by construction [LB93]. All three examples are expressible in the form of alternating R/Q timed automata, and therefore it was possible to apply this algorithm. The second algorithm (the column marked [Wu94]) assumes a certain structural property that guarantees the SPP. The third algorithm (marked [BSV95]) does not make use of the SPP, but uses other heuristics to make the iterations efficient. The fourth algorithm (marked [TB95]) is the one described in Section 3.2.3. On the basis of the examples in Table 3.1, the performance of our implementation is better than those of [LB93] and [Wu94], and is comparable to that of [BSV95].

3.3 Elimination of Multiple Traces in One Iteration

The iterative timing verification algorithms in the literature use an untimed reachability computation tool to obtain an error trace, and proceed by restricting the system to eliminate that trace if it is found to be inconsistent with the timing constraints. Typically, the

[3] Intuitively, a liveness property is a property that the system must eventually satisfy. Thus, any timed event sequence can be extended to satisfy a liveness property. When a quantitative bound is placed on the time within which the property must be satisfied, (a "bounded liveness property") the property formally becomes a safety property [BSV95]. This is because, after the bound is exceeded, later behavior of the system can not satisfy the property.

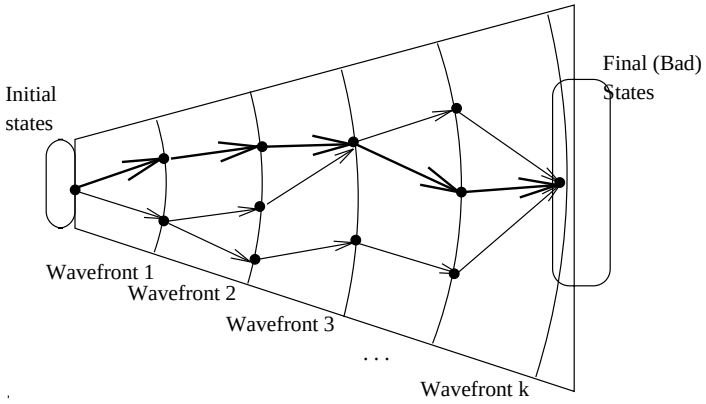

Figure 3.6: HSIS error trace computation.

reachability tool computes enough information to obtain more than one error trace each time it is run. We propose a method that makes use of all of these traces. This offers a new flexibility: the time spent in each iteration is increased, but, in return, the number of iterations is decreased. Since reachability computation is usually the most time consuming part of the verification process, fewer iterations can reduce run time significantly. In this section, we address the problem of making use of all these traces within the iterative timing verification algorithms in HSIS [ABB+94].

Starting from the initial locations, HSIS computes the wavefronts depicted in Figure 3.6, where the nth wavefront corresponds to the set of locations reachable by traversing n edges. When the latest wavefront contains a "bad" location $s_{bad} \in S_{target}$, HSIS computes a path leading to s_{bad} by moving backwards from s_{bad} until an initial location is reached. Suppose that s_{bad} is on wavefront k. The algorithm for computing a path π leading to s_{bad} from an initial location is given in Figure 3.7.

Instead of choosing $p \in S$, we propose keeping all of the information about S, i.e., computing and recording the transitions that reach some location in S from wavefront $(j-1)$. This way, the part of the transition structure that leads to s_{bad} from the initial locations is retained. Then, the paths in this reduced transition structure are checked for traversability. This timed reachability analysis is simpler since the graph involves no loops and is smaller than the original automaton.

Suppose that the paths in this reduced graph have been shown to be untraversable because of timing constraints. Let us generalize the method of the previous section to eliminate all paths in this graph. Note that if any path is found to admit a consistent timing,

$s \leftarrow s_{bad}$, $j \leftarrow k$, $\pi = s$
While s is not an initial location do
    $S \leftarrow$ the set of locations on wavefront $(j-1)$ that can reach s
    If S contains an initial location s_i
        $\pi \leftarrow s_i \circ \pi$: Add s_i to the beginning of π.
    else
        Choose $p \in S$
        $\pi \leftarrow p \circ \pi$
        $s \leftarrow p$, $j \leftarrow j-1$

Figure 3.7: Algorithm in HSIS for finding a path leading to s_{bad} from an initial location

then language emptiness does not hold and a timed error trace can be returned. This is one improvement over considering one trace at a time: if there is a timing consistent path that leads to s_{bad}, it is discovered at this stage without needing to eliminate timing inconsistent paths leading to s_{bad} and run the reachability routine after each such elimination.

As argued in the previous section, what determines the timed traversability of a path is its effective RQ sequence. Therefore, in the above graph (henceforth referred to as G_{err}), locations that are connected by an edge with no resets or queries can be merged. Then all resets which are not the effective resets for a query along some path are deleted. After this, if two edges originating from the same location have the same resets and queries, these edges (i.e., the locations that they lead to) are merged by a similar reasoning.

At this point, all RQ sequences represented by G_{err} are known to admit no consistent timing, therefore, they must all be eliminated. By removing these paths all at once, we eliminate the computational cost of running the reachability algorithm after eliminating each of them. We propose the following method to eliminate a group of RQ sequences in one pass: For each RQ sequence Θ, we find a minimal Θ_{min} that dominates Θ and admits no consistent timing. We then construct $M_{\Theta_{min}}$ for each distinct Θ_{min} and take the product of the system with all such $M_{\Theta_{min}}$.

One possible difficulty with this approach is that there may be a large number of RQ sequences all dominated by a few Θ_{min}. To alleviate this, after calculating each Θ_{min}, we remove from G_{err} those RQ sequences that are dominated by Θ_{min}. With this optional step, we can guarantee that we perform no more computation than an iterative tool that

processes one trace at a time, while, on the other hand, saving on the number of times the language emptiness routine is called. This step need not be performed if the number of RQ sequences in G_{err} is not large.

3.4 Conclusions

We presented a technique for iterative verification with timed automata that does not restrict the timing constraints used, and that exploits the SPP if the automaton has this property. This technique eliminates the need to decide whether the automaton has the SPP, and is a complete decision procedure for automata for which iterative algorithms already exist. We also proposed the use of a set of error traces at each iteration of a timing verifier. We argued that no efficiency is lost, and significant reductions in the number of iterations are possible.

The use of the techniques presented in this chapter is not restricted to timed automata: any finite transition structure coupled with a set of real-valued variables and constraints and assignments to these variables on the transitions can be handled in this framework.

Our conclusion from the study reported in this chapter was that iterative abstraction methods reduce verification complexity significantly if much of the timing information in the system description is redundant. However, in general, this form of complexity relief is not sufficient for handling systems with a large number of concurrent constraints for which all of the timing information is needed at least some of the time. This provides the motivation for the hierarchical and compositional abstraction methods to be developed in the forthcoming chapters.

Chapter 4

Abstraction Relations for Timed Systems

4.1 Introduction

This chapter addresses the problem of checking the correctness of abstractions of real-time systems. For two given representations of a system, we would like to prove that the more refined one is a correct implementation of the more abstract one. This is referred to as a *refinement check* or an *abstraction check*. By checking refinement, we are assured that the properties proved using the abstract description are satisfied by the refined version. This scenario may arise either because a design is being carried out in a top-down fashion through a series of refinements or because a system is too complex for automatic verification algorithms and a simpler, more abstract model of the system needs to be used. This paradigm is also referred to as *hierarchical verification*: a hierarchy of descriptions at varying levels of abstraction are given for a system, and for adjacent levels in this hierarchy, it is required that the lower one be a correct implementation of the upper one.

Several alternatives exist for formally defining what it means for a real-time system to "correctly implement" another. In Section 2.4.3 we defined the timed implementation relation $\preceq_L$ to be timed language inclusion: a real-time system modeled by the timed automaton A implements B if $A \preceq_L B$. While this relation is a natural choice for an abstraction relation, it was proved in [AD94] that language containment is undecidable for non-deterministic systems. Since abstract descriptions often involve non-determinism, abstraction relations

that are computationally tractable are needed for hierarchical verification. Moreover, when a refinement check is performed, often there is a tighter relation between the abstract and the concrete descriptions: the concrete description makes explicit detail that is left out in the abstract version. Thus, each implementation state corresponds to a state of the abstract description. Abstraction relations that require such correspondence are stronger notions of implementation than language inclusion and are less complex to check. In this chapter, we investigate two such relations: timed simulations and homomorphisms for timed systems. In Section 4.2 we propose timed simulation relations as a sufficient condition for implementation. For each state in the refined automaton, a timed simulation relation specifies states in the abstract automaton that exhibit the same input-output behavior. In Section 4.2.2 we prove that it is decidable (in EXPTIME) to check whether such a relation exists between two systems. Section 4.3 explores homomorphisms, which are restricted forms of simulation relations, as an alternative way of specifying and verifying timed abstractions. The homomorphism-based approach requires the user to specify a correspondence between the locations (not states) of the abstract and refined versions of a system, but is computationally more feasible, and is a generalization of the existing homomorphism checks supported by the system COSPAN. In Section 4.3, we present an algorithm for checking if a given mapping between the locations of two timed systems implies inclusion of timed languages. This algorithm was implemented in COSPAN and was used in the hierarchical verification of two asynchronous circuits, presented in Section 5.4 and Chapter 6. Section 4.4 contrasts the work presented in this chapter with other existing approaches to verifying abstractions. Section 4.5 concludes the chapter and presents the motivation for compositional strategies for hierarchical verification.

4.2 Timed Simulation Relations

Suppose we would like to prove that a timed automaton A is a correct implementation of B. A timed simulation relation pairs states of A and B that have identical responses to all timed events. If it is possible to match each initial state of A with some initial state of B in this manner, then B can imitate all traces of A and thus $A \preceq_L B$. Below we define timed simulation relations formally.

4.2.1 Definitions

Suppose that A is comparable to B. A *timed simulation relation* from A to B is a binary relation $\Omega \subseteq \Sigma_A \times \Sigma_B$ among the states of the two automata such that for every $(\sigma, \tau) \in \Omega$, and for every timed event γ in Γ_A, if $\sigma \xrightarrow{\gamma} \sigma'$ then there exists $\tau' \in \Sigma_B$ such that $\tau \xrightarrow{\gamma} \tau'$ and (σ', τ') is in Ω. This requirement is referred to as the *simulation condition* for the pair of states (σ, τ). The timed simulation relation Ω is said to be *initialized* iff for every initial state σ of A, there exists an initial state τ of B with $(\sigma, \tau) \in \Omega$. If the timed automaton A is comparable to the timed automaton B, and an initialized timed simulation relation from A to B exists, then A is said to *timed-simulate* B, written $A \preceq_S B$.

Proposition 4.2.1 (Preorder) *The $\preceq_S$ relation is reflexive and transitive.*

Two timed automata A and B are *timed simulation equivalent*, written $A \cong_S B$, iff both $A \preceq_S B$ and $B \preceq_S A$. It follows that the relation $\cong_S$ is an equivalence relation. Timed simulation implies timed language inclusion, but is a stronger requirement.

Proposition 4.2.2 (Languages and simulation) *If $A \preceq_S B$, then $A \preceq_L B$.*

It is known [Mil71] that the converse of Proposition 4.2.2 does not hold. Simulation is a condition about the transition structures of two automata that involves the internal states as well as observed behavior. Thus, it is more restrictive than language inclusion.

The properties of timed simulation formalized in Propositions 4.2.3 and 4.2.4 make it suitable for use in a hierarchical and compositional verification framework.

Proposition 4.2.3 *The composed automaton is an implementation of each component: $A \parallel C \preceq_S A$.*

Proposition 4.2.4 (Compositionality) *Timed simulation is a congruence with respect to the composition operator $\parallel$: If $A \preceq_S B$ then $A \parallel C \preceq_S B \parallel C$.*

The next section shows that timed simulation is computationally tractable.

4.2.2 Decidability of Timed Simulation

A language inclusion check between non-deterministic timed automata is undecidable in the general case[1], while [Čerāns92] has proved that computing timed bisimulation is

[1]The halting problem for non-deterministic 2-counter machines can be reduced to the language inclusion problem in a similar manner to [AD94]. There are two key differences from [AD94]: the non-halting

decidable. The existence of a timed simulation relation is a sufficient condition for language inclusion. In the following, we will show that the problem of checking the existence of a timed simulation relation is decidable. It follows that simulation relations are a good choice for the implementation preorder.

Because of the dense real time domain involved, the decidability of simulation is not a straightforward extension of the corresponding result for untimed automata. The state spaces of the two automata involved in the refinement check are uncountably infinite. A simulation relation must specify for each pair of states whether they are related, which makes the number of relations that are candidates for being a simulation also uncountably infinite. Furthermore, for a pair of states in a given candidate relation, it must be checked for each timed event that they behave in the required way. Because of the real-valued time increment δ involved in a timed event $\gamma = \langle \delta, \mathbf{f}, \mathbf{f}' \rangle$, there are uncountably many events that the simulation condition needs to be checked for.

It may appear at first that this can be achieved by checking whether the untimed automaton $Region(A)$ is simulated by $Region(B)$. This condition does not imply timed simulation. Instead, it gives rise to *time-abstract simulation* [ČGL93, TY96]. We cannot infer from time-abstract simulation that the timed language of A is contained in that of B, thus, timed properties of B do not carry over to A. However, properties not involving time are preserved: for each timed sequence $\overline{\gamma^B} = \gamma_1^B, \gamma_2^B, ... \in \mathcal{L}(B)$ there exists $\overline{\gamma^A} = \gamma_1^A, \gamma_2^A, ... \in \mathcal{L}(A)$ where, for each i, if $\gamma_i^B = \langle \delta_i^B, \mathbf{f}_i, \mathbf{f}_i' \rangle$, $\gamma^A = \langle \delta_i^A, \mathbf{f}_i, \mathbf{f}_i' \rangle$ for a possibly different time increment δ_i^A. The sequence of input-output transitions is preserved, but time durations are not. For hierarchical verification of timed properties, this is not sufficient.

We check timed simulation by reducing the problem to a finite check on the (finitely many) equivalence classes of a relation defined on $\Sigma_A \times \Sigma_B$. This relation extends region equivalence to the product state space in the same manner as [Čerāns92]. Our proof generalizes to timed simulations the result of [Čerāns92] on the decidability of bisimulations.

Uniform functions and region equivalence

Uniform functions help us relate clock valuations that belong to the same region. We will use them for showing that two region equivalent valuations can imitate each other's transitions. In the following, A is a timed automaton and $\equiv$ is interpreted on A.

computations of the 2-counter machine are encoded as finite strings and accepting states of automata are mimicked by using output variables.

Definition 4.2.5 *A function $\rho : \mathbb{R} \to \mathbb{R}$ is said to be* **uniform** *iff*

- *ρ is continuous,*
- *$x < y \Rightarrow \rho(x) < \rho(y)$: ρ is strictly increasing, and,*
- *$\rho(0) = 0$, and, for every $k \in \mathbb{N}$ and $x \in \mathbb{R}$, $\rho(x + k) = \rho(x) + k$.*

Observe that a uniform function is uniquely defined by specifying its value on the interval $(0, 1)$. For X-valuations Φ and Θ, we write $\Phi = \rho(\Theta)$ to denote the fact that for each $x \in X$ either $\Phi(x), \Theta(x) > K_x$, or $\Phi(x) = \rho(\Theta(x))$.

Lemma 4.2.6 *Let $\rho(x)$ be a uniform function. Then the following functions are also uniform:*

- *$\rho^{-1}(x)$, the inverse of ρ,*
- *$-\rho(-x)$,*
- *$\rho_\Delta(x) =_{def} \rho(x - \Delta) - \rho(-\Delta)$.*

Lemma 4.2.7 (Region equivalence and uniform functions) *$\langle \sigma, \Phi \rangle \equiv \langle \tau, \Theta \rangle$ if and only if there is a uniform mapping ρ such that $\Theta = \rho(\Phi)$.*

Proof:

($\Rightarrow$) Let Φ and Θ be region equivalent. Let $u_1 \leq u_2 \leq ... \leq u_{n-1}$ and $u'_1 \leq u'_2 \leq ... \leq u'_{n-1}$ be the non-zero fractional parts of the clock values in $\Phi(x)$ and $\Theta(x)$, respectively, for values of $x \in X$ such that $\Phi(x), \Theta(x) < K_x$. Define $u_0 = u'_0 = 0$ and $u_n = u'_n = 1$. Let $\rho(u_i) = u'_i$ and for any $x \in (0, 1)$ such that $u_j < x < u_{j+1}$, let

$$\rho(x) = \rho(u_j) + (\rho(u_{j+1}) - \rho(u_j))(x - u_j)/(u_{j+1} - u_j)$$

ρ as defined is uniform, and $\Theta = \rho(\Phi)$.

($\Leftarrow$) Follows from the facts that $\rho(k) = k$ for $k \in \mathbb{N}$, ρ is strictly increasing, and $Fr(\rho(x)) = Fr(\rho(x'))$ iff $Fr(x) = Fr(x')$ or $x, x' > K_x$.

■

Corollary 4.2.8 *For an integer $k \leq max_{x \in X} K_x$, a uniform function ρ and $\diamond$ denoting one of $\leq$, $\geq$ and $=$, $x \diamond k$ if and only if $\rho(x) \diamond k$. Therefore, a clock predicate on an edge of A is satisfied by Φ iff it is satisfied by Θ.*

Lemma 4.2.9 *Suppose that $\langle s, \Phi \rangle \stackrel{\langle \delta, \mathbf{f}, \mathbf{f}' \rangle}{\rightarrow} \langle t, \tilde{\Phi} \rangle$ for some timed event $\langle \delta, \mathbf{f}, \mathbf{f}' \rangle$ and $\langle s, \Phi \rangle \equiv \langle s, \Theta \rangle$ such that $\Theta = \rho(\Phi)$ for a uniform function ρ. Then $\langle s, \Theta \rangle \stackrel{\langle \delta', \mathbf{f}, \mathbf{f}' \rangle}{\rightarrow} \langle t, \tilde{\Theta} \rangle$, where $\delta' = -\rho(-\delta)$ and $\langle t, \tilde{\Phi} \rangle \equiv \langle t, \tilde{\Theta} \rangle$ such that $\tilde{\Theta} = \rho_\delta(\Phi)$ for the uniform function ρ_δ.*

Proof: We will first argue that $wait(\langle s, \Theta \rangle, \delta')$. Let us pick arbitrary $\Delta' < \delta'$. We must show that $\Theta + \Delta' \models \alpha(s)$. Since $wait(\langle s, \Phi \rangle, \delta)$, we have $(\Phi + \Delta) \models \alpha(s)$ for all $\Delta < \delta$. Recall (from Lemma 4.2.6) that $-\rho(-x)$ is a uniform function. By the continuity of $-\rho(-x)$, we know that there exists $\Delta < \delta$ satisfying $\Delta' = -\rho(-\Delta)$. Then $\Theta + \Delta' = \rho(\Phi) - \rho(-\Delta) = \rho_\Delta(\Phi + \Delta)$. By Corollary 4.2.8 and the fact that ρ_Δ is uniform, we infer $(\Theta + \Delta') \models \alpha(s)$. Thus, $wait(\langle s, \Theta \rangle, \delta')$.

After waiting for δ' starting at $\langle s, \Theta \rangle$, we reach $\tau = \langle s, \Theta + \delta' \rangle = \langle s, \rho_\delta(\Phi + \delta) \rangle$. Let $e = \langle s, t, \varphi, \chi, R \rangle$ be the edge associated with the transition $\langle s, \Phi \rangle \stackrel{\langle \delta, \mathbf{f}, \mathbf{f}' \rangle}{\rightarrow} \langle t, \tilde{\Phi} \rangle$. Again by Corollary 4.2.8, the clock values at the state τ satisfy φ, since $(\Phi + \delta) \models \varphi$. Thus, we can take the edge e at state τ. The final values of the clocks after taking e starting at τ are given by $(\rho_\delta(\Phi + \delta))[R := 0] = \rho_\delta((\Phi + \delta)[R := 0])$, which proves that $\tilde{\Theta} = \rho_\delta(\tilde{\Phi})$. ■

Simulation relations and region equivalence

As argued earlier, checking whether a timed automaton A timed simulates B requires us to define an equivalence relation that is finer than the region equivalence of each automaton. We will use a region equivalence defined on pairs of clock valuations, i.e., in the product state-space of A and B. Let Φ^A and Φ^B denote clock valuations for A and B respectively. We denote by $\Phi^{A||B} = \Phi^A \cup \Phi^B$ the following clock valuation:

$$\Phi(x) = \begin{cases} \Phi^A(x), \text{ if } x \in X^A \\ \Phi^B(x), \text{ if } x \in X^B \end{cases}$$

Region equivalence on pairs of clock valuations is defined as follows: $\langle \Phi^A, \Phi^B \rangle \equiv \langle \Theta^A, \Theta^B \rangle$ iff for all $x \in X^A \cup X^B$, either both $\Phi^{A||B}(x)$ and $\Theta^{A||B}(x)$ are larger than K_x, or the following hold

- $\lfloor \Phi^{A||B}(x) \rfloor = \lfloor \Theta^{A||B}(x) \rfloor$, and

- For all $x' \in X$ such that $\Phi^{A||B}(x') \leq K_{x'}$, $\quad Fr(\Phi^{A||B}(x)) \geq Fr(\Phi^{A||B}(x'))$ iff $Fr(\Theta^{A||B}(x)) \geq Fr(\Theta^{A||B}(x'))$, and
- $Fr(\Phi^{A||B}(x)) = 0 \quad \Leftrightarrow \quad Fr(\Theta^{A||B}(x)) = 0.$

Note that this equivalence relation requires the ordering of fractional parts of all of $X^A \cup X^B$ be the same in the two pairs of valuations. This is a stronger requirement than the product of the region equivalences defined on each automaton alone. In the rest of this chapter, we will use this version of region equivalence.

Let $\sigma^A = \langle s^A, \Phi^A \rangle$, $\sigma^B = \langle s^B, \Phi^B \rangle$, $\tau^A = \langle t^A, \Theta^A \rangle$, and $\tau^B = \langle t^B, \Theta^B \rangle$. We write $\langle \sigma^A, \sigma^B \rangle \equiv \langle \tau^A, \tau^B \rangle$ iff $s^A = t^A$, $s^B = t^B$, and $\langle \Phi^A, \Phi^B \rangle$ is region equivalent with $\langle \Theta^A, \Theta^B \rangle$. For a state $\sigma = \langle s, \Phi \rangle$ we write $\rho(\sigma)$ to denote the state $\langle s, \rho(\Phi) \rangle$.

The following theorem stipulates that any simulation relation can be extended to one consisting of equivalence classes of $\equiv$. With the aid of this theorem, it will be sufficient to consider a finite number of candidates for timed simulation relations.

Theorem 4.2.10 *Let Ω be a timed simulation from A to B. Define*

$$\Omega' = \left\{ \langle \sigma^A, \sigma^B \rangle \;\middle|\; \langle \sigma^A, \sigma^B \rangle \equiv \langle \tau^A, \tau^B \rangle \text{ for some } \langle \tau^A, \tau^B \rangle \in \Omega \right\}$$

Then, Ω' is also a timed simulation relation from A to B. Furthermore, if Ω is initialized, then so is Ω'.

Proof: Let $\langle \sigma^A, \sigma^B \rangle \in \Omega'$ and $\sigma^A \xrightarrow{\gamma} \tilde{\sigma}^A$ for a timed event $\gamma = \langle \delta, \mathbf{f}, \mathbf{f}' \rangle$. We need to prove that there exists $\tilde{\sigma}^B$ such that $\sigma^B \xrightarrow{\gamma} \tilde{\sigma}^B$ and $\langle \tilde{\sigma}^A, \tilde{\sigma}^B \rangle \in \Omega'$.

Since $\langle \sigma^A, \sigma^B \rangle \in \Omega'$, there exists $\langle \tau^A, \tau^B \rangle \in \Omega$ such that $\langle \sigma^A, \sigma^B \rangle \equiv \langle \tau^A, \tau^B \rangle$. By Lemma 4.2.7, there exists a uniform function ρ satisfying $\langle \tau^A, \tau^B \rangle = \rho(\langle \sigma^A, \sigma^B \rangle)$. By Lemma 4.2.9, $\tau^A \xrightarrow{\theta} \tilde{\tau}^A$ for $\tilde{\tau}^A = \rho_\delta(\tilde{\sigma}^A)$, where $\theta = \langle \delta', \mathbf{f}, \mathbf{f}' \rangle$ and $\delta' = -\rho(-\delta)$. The fact that $\langle \tau^A, \tau^B \rangle \in \Omega$ implies that $\tau^B \xrightarrow{\theta} \tilde{\tau}^B$ for some $\tilde{\tau}^B$ such that $\langle \tilde{\tau}^A, \tilde{\tau}^B \rangle \in \Omega$. Since $\sigma^B = \rho^{-1}(\tau^B)$, and since ρ^{-1} is a uniform function, from Lemma 4.2.9 we infer that $\sigma^B \xrightarrow{\gamma'} \tilde{\sigma}^B$ for $\gamma' = \langle \delta'', \mathbf{f}, \mathbf{f}' \rangle$, $\delta'' = -\rho^{-1}(-\delta') = \delta$, and $\tilde{\sigma}^B = (\rho^{-1})_{\delta'}(\tilde{\tau}^B)$. But then $\gamma' = \gamma$, and $\tilde{\sigma}^B = (\rho^{-1})_{\delta'}(\tilde{\tau}^B)$ implies $\tilde{\tau}^B = \rho_\delta(\tilde{\sigma}^B)$. Thus we have $\langle \tilde{\tau}^A, \tilde{\tau}^B \rangle = \rho_\delta(\langle \tilde{\sigma}^A, \tilde{\sigma}^B \rangle)$, which means, by Lemma 4.2.9, $\langle \tilde{\tau}^A, \tilde{\tau}^B \rangle \equiv \langle \tilde{\sigma}^A, \tilde{\sigma}^B \rangle$ which in turn implies $\langle \tilde{\sigma}^A, \tilde{\sigma}^B \rangle \in \Omega'$. ■

Theorem 4.2.10 implies that the maximal timed simulation relation consists of a union of equivalence classes of $\equiv$. In the following, we will develop machinery to show that given A and B, the problem of deciding if a timed simulation from A to B exists can be converted

to a condition on the equivalence classes of $\equiv$ which can be checked in finitely many steps. From this we will infer that the problem is decidable.

Symbolic simulations.

For timed automata A and B, let $\mathcal{EQ}_{A||B}$ be the set of equivalence classes of $\equiv$ on $\Sigma_{A||B}$ as defined above. With $\mathcal{EQ}(\sigma^A, \sigma^B)$, denote the equivalence class that the state $\langle \sigma^A, \sigma^B \rangle \in \Sigma_{A||B}$ belongs to. Let $\sigma^A = \langle s^A, \Phi^A \rangle$ and $\sigma^B = \langle s^B, \Phi^B \rangle$ Let $u_0 \leq u_1 \leq ... \leq u_k$ be the fractional parts of clock values specified by the clock valuation $\Phi^{A||B} = \Phi^A \cup \Phi^B$. Let k_A be such that u_{k_A} is the largest fractional part of a clock value of σ^A. Define

$$Times(\langle \sigma^A, \sigma^B \rangle) = \{(1 - u_i) \mid k_A \leq i \leq k\} \cup \{(1 - 0.5(u_i + u_{i+1})) \mid k_A \leq i \leq k\}$$

with the convention that $u_{k+1} = 1$. If we imagine the fractional parts of clocks ordered on the real line and if we also include the mid-points between each two adjacent fractional parts, $Times$ consists of the distances between these points and the next integer point.

Lemma 4.2.11 *Suppose that* $\langle \tau^A, \tau^B \rangle \equiv \langle \sigma^A, \sigma^B \rangle$ *and let*

$$Times(\langle \sigma^A, \sigma^B \rangle) = \{a_1, ..., a_n\}$$

with $a_1 < a_2 < ... < a_n$ *and*

$$Times(\langle \tau^A, \tau^B \rangle) = \{b_1, ..., b_n\}$$

where the b_i*'s are similarly ordered.*

Then there exists a uniform function ρ *such that* $\langle \tau^A, \tau^B \rangle = \rho(\langle \sigma^A, \sigma^B \rangle)$ *and for each* i, $\rho(a_i) = b_i$.

Proof: Let $0 =_{def} u_0 \leq u_1 \leq u_2 \leq ... \leq u_k \leq u_{k+1} =_{def} 1$ be the fractional parts of the clock values in $\langle \tau^A, \tau^B \rangle$. We construct $0 = p_0 \leq p_1 \leq p_2 \leq ... \leq p_{2k} \leq p_{2k+1} = 1$ such that for each $1 \leq i \leq k$, $p_{2i} = u_i$ and $p_{2i-1} = 0.5(u_i + u_{i-1})$. Let $= q_0 \leq q_1 \leq q_2 \leq ... \leq q_{2k} \leq q_{2k+1} = 1$ the corresponding list for $\langle \sigma^A, \sigma^B \rangle$. A uniform function mapping the q_j's to the p_j's in the same manner as in the proof of Lemma 4.2.7 satisfies the requirements of this lemma. ■

We say that a subset of equivalence classes $\mathcal{X} \subseteq \mathcal{EQ}_{A||B}$ is a *symbolic simulation* from A to B iff for each $\mathcal{EQ}(\sigma^A, \sigma^B) \in \mathcal{X}$ the following condition is satisfied: for every timed event $\gamma = \langle t, \mathbf{f}, \mathbf{f}' \rangle$ where $t \in Times(\langle \sigma^A, \sigma^B \rangle)$, if $\sigma^A \xrightarrow{\gamma} \tau^A$, then

(1) $\sigma^B \xrightarrow{\gamma} \tau^B$ for some τ^B such that $\mathcal{EQ}(\tau^A, \tau^B) \in \mathcal{X}$, and

(2) If A can wait in σ^A for time t, then so can B in state σ^B.

Note that, by Lemmas 4.2.9 and 4.2.11, the conditions above are independent of what representative is chosen for each equivalence class.

Theorem 4.2.12 *Given $\mathcal{X} \subseteq \mathcal{EQ}_{A\|B}$ let $\mathcal{R}_{\mathcal{X}} = \left\{ \langle \sigma^A, \sigma^B \rangle \;\middle|\; \mathcal{EQ}(\sigma^A, \sigma^B) \in \mathcal{X} \right\}$. $\mathcal{R}_{\mathcal{X}}$ is a timed simulation relation from A to B iff $\mathcal{X}$ is a symbolic simulation from A to B.*

Proof:

($\Rightarrow$) Assume that $\mathcal{R}_{\mathcal{X}}$ is a timed simulation relation from A to B. Note that every timed simulation relation is included in such an $\mathcal{R}_{\mathcal{X}}$ by Theorem 4.2.10. Let $\mathcal{EQ}(\sigma^A, \sigma^B) \in \mathcal{X}$ and $\gamma = \langle t, \mathbf{f}, \mathbf{f}' \rangle$ be such that $t \in Times(\langle \sigma^A, \sigma^B \rangle)$ and $\sigma^A \xrightarrow{\gamma} \tau^A$. That τ^B satisfying conditions (1) and (2) of a symbolic simulation exists immediately follows from the fact that $\mathcal{R}_{\mathcal{X}}$ is a timed simulation relation.

($\Leftarrow$) Suppose that $\mathcal{X}$ is a symbolic simulation from A to B. Let $\langle \sigma^A, \sigma^B \rangle \in \mathcal{R}_{\mathcal{X}}$ and let $\sigma^A \xrightarrow{\gamma} \tau^A$ for some timed event $\gamma = \langle t, \mathbf{f}, \mathbf{f}' \rangle$. We need to show that there exists a τ^B such that $\sigma^B \xrightarrow{\gamma} \tau^B$ and $\langle \tau^B, \tau^B \rangle \in \mathcal{R}_{\mathcal{X}}$. If $t \in Times(\langle \sigma^A, \sigma^B \rangle)$, the claim holds by the definition of a symbolic simulation. Otherwise, let t_{max} be the largest element of $Times(\langle \sigma^A, \sigma^B \rangle)$. We will denote the latter with $Times$ in the rest of the proof. The proof splits into two cases.

(I) $t \leq t_{max}$. Then, $p < t < q$ for some p and q in $Times$. One of p and q has the form $1 - 0.5(u_i + u_{i+1})$ (call this one r) and the other has the form $1 - u_j$, for some i and j. Let $\sigma^A = \langle s^A, \Phi^A \rangle$ and $\sigma^B = \langle s^B, \Phi^B \rangle$. Define $\theta = \langle r, \mathbf{f}, \mathbf{f}' \rangle$, i.e., the time increment in γ replaced with r. $\sigma^A \xrightarrow{\theta} \tau^A$, since r and t are both less than $1 - u_{k_A}$, which means that the same timing predicates are satisfied by Φ^A, $\Phi^A + t$ and $\Phi^A + r$. By the fact that $\mathcal{X}$ is a symbolic simulation, and $r \in Times$, $\sigma^B \xrightarrow{\theta} \tau^B$ for some τ^B such that $\langle \tau^A, \tau^B \rangle \in \mathcal{R}_{\mathcal{X}}$. We claim that $\sigma^B \xrightarrow{\gamma} \tau^B$, which will imply the desired result. This claim follows from the fact that $\Phi^B + t$ satisfies the same clock predicates as $\Phi^B + r$, since no clock value crosses an integer value between these two clock valuations.

(II) $t > t_{max}$. Let us consider all points in time when a clock of A takes on an integer value as A waits for time t starting from state σ^A. Clearly there must be a finite

number of such points since t is finite and there are a finite number of clocks. More precisely, let $\sigma_1^A, \sigma_2^A, ..., \sigma_k^A$ be the maximal sequence of states such that (1) σ_i^A is reached from σ^A by waiting for time δ_i, and has some clock with an integer value, and (2) $\sigma_k^A \stackrel{\langle \delta, \mathbf{f}, \mathbf{f}' \rangle}{\rightarrow} \tau^A$, where $\delta = t - \delta_k$. We will prove the original claim in ($\Leftarrow$) by induction on k. The case where $k = 0$ follows from (I). Let us assume that the claim holds for k. We will show that the claim holds for $k+1$. Consider $\sigma_1^A, \sigma_2^A, ..., \sigma_k^A, \sigma_{k+1}^A$. Observe that $\delta_1 \in Times(\langle \sigma^A, \sigma^B \rangle)$. Since $\mathcal{X}$ is a symbolic simulation, and since A can wait for δ_1 at σ^A to reach σ_1^A, B can wait at σ^B for δ_1 to reach σ_1^B such that $\mathcal{EQ}(\sigma_1^A, \sigma_1^B) \in \mathcal{X}$. Applying the induction assumption to the transition $\sigma_1^A \stackrel{\langle t - \delta_1, \mathbf{f}, \mathbf{f}' \rangle}{\rightarrow} \tau^A$, we conclude that there exists τ^B such that $\sigma_1^B \stackrel{\langle t - \delta_1, \mathbf{f}, \mathbf{f}' \rangle}{\rightarrow} \tau^B$. Thus, at state σ^B, B can wait for a total of t and take a transition to τ^B, such that $\mathcal{EQ}(\tau^A, \tau^B) \in \mathcal{X}$, i.e., $\langle \tau^A, \tau^B \rangle \in \mathcal{R}_{\mathcal{X}}$ as desired.

■

Given a timed automaton A that is comparable with B, we can construct the finite set of equivalence classes $\mathcal{EQ}_{A||B}$, and then for each subset $\mathcal{X}$ of $\mathcal{EQ}_{A||B}$, we can check whether $\mathcal{X}$ is a symbolic simulation from A to B. This check is finite, since for each member of $\mathcal{X}$, we need to check a finite number of conditions. If $\mathcal{X}$ is indeed a symbolic simulation, then we can obtain a simulation relation $\mathcal{R}_{\mathcal{X}}$ from $\mathcal{X}$ by Theorem 4.2.12. If $\mathcal{R}_{\mathcal{X}}$ is initial for some $\mathcal{X} \subseteq \mathcal{EQ}_{A||B}$, then $A \preceq_S B$. The converses of these statements are also true by Theorem 4.2.12. It follows that determining whether $A \preceq_S B$ is decidable: this problem can be reduced to computing the maximal symbolic simulation relation over the equivalence classes $\mathcal{EQ}_{A||B}$. For this purpose, any of the existing algorithms for computing simulation (e.g. see [KS90, HHK95]) can be adopted to obtain an algorithm with complexity polynomial in the size of $\mathcal{EQ}_{A||B}$. The size of $\mathcal{EQ}_{A||B}$ is polynomial in the number of locations and exponential in the number of clocks and the size of binary encodings of the clock constraints. This gives an exponential-time algorithm for checking timed simulation:

Theorem 4.2.13 *Given two timed automata A and B, the problem of checking whether A timed-simulates B is solvable in* EXPTIME.

4.3 Verification Using Homomorphisms

In this section, we propose homomorphisms as an alternative way of proving timed refinement. Homomorphisms are mappings from the locations of one process to those of the other which conserve the transition structure. There are several reasons for exploring this alternative approach. First, the algorithm of the previous section is computationally expensive, and we do not know of good heuristics to proceed with an implementation. Second, the tool COSPAN already supports the use of homomorphisms for proving refinements for untimed systems, which can naturally be generalized to timed systems. Third, homomorphisms capture the user's intuition of the correspondence between the two levels, and it is desirable to make use of this knowledge to reduce the computational cost of checking refinement[2].

4.3.1 Definitions

A mapping $h : S^A \to S^B$ is said to *preserve untimed behavior* iff for each run $s_0 \stackrel{\theta_0}{\to} s_1 \stackrel{\theta_1}{\to} \cdots \stackrel{\theta_{k-1}}{\to} s_k$ of $Untime(A)$ with $s_0 \in S_0^A$, $h(s_0) \stackrel{\theta_0}{\to} h(s_1) \stackrel{\theta_1}{\to} \cdots \stackrel{\theta_{k-1}}{\to} h(s_k)$ is a run of $Untime(B)$ with $h(s_0) \in S_0^B$. The mapping h preserves untimed behavior iff h is a state homomorphism from the reachable locations of $Untime(A)$ to $Untime(B)$ (see [Kur94] for the definition of homomorphisms for untimed systems). h is a mapping to be supplied by the user capturing the intuition regarding the correspondence of locations at different levels. The syntax of S/R [Gla95], the input language of COSPAN, allows specifications of such maps. The verifier COSPAN checks, using either an on-the-fly depth-first-search or a BDD-based symbolic search of the product of the two state-spaces, whether the user-supplied mapping h preserves untimed behavior. In the following, we generalize this to timed behavior.

A mapping $h : S^A \to S^B$ is said to *preserve timed behavior* iff for each run $\langle s_0, \Phi_0 \rangle \stackrel{\gamma_0}{\to} \langle s_1, \Phi_1 \rangle \stackrel{\gamma_1}{\to} \cdots \stackrel{\gamma_{k-1}}{\to} \langle s_k, \Phi_k \rangle$ of A starting at an initial state, there exist $\Theta_0, \ldots, \Theta_k$ such that $\langle h(s_0), \Theta_0 \rangle \stackrel{\gamma_0}{\to} \langle h(s_1), \Theta_1 \rangle \stackrel{\gamma_1}{\to} \ldots \stackrel{\gamma_{k-1}}{\to} \langle h(s_k), \Theta_k \rangle$ is a run of B that starts at an initial state. Note that, since there is at most one edge between two locations (see Section 2.3), if such $\Theta_0, \ldots, \Theta_k$ exist, then they are uniquely determined. It is straightforward to prove the following proposition:

Proposition 4.3.1 *If $h : S^A \to S^B$ preserves timed behavior, then $A \preceq_L B$.*

[2] A user-given simulation relation has this property as well, but is harder to specify.

Observe that for a mapping to preserve timed or untimed behavior, it must map initial locations to initial locations. We restrict our attention to such mappings.

4.3.2 Checking Preservation of Timed Behavior

The remaining part of this section describes a method for checking if a given mapping h preserves timed behavior. This is achieved by converting the problem to an untimed homomorphism check, which can then be performed by a tool such as COSPAN [Kur94]. The basic idea is to construct an untimed transition structure $\mathcal{R}$ in the fashion of the region automaton of Section 4.2.2. This approach has the added advantage that we use the homomorphism checking in COSPAN as a black-box, and thus, we can use either an on-the-fly depth-first search or a BDD-based symbolic search.

The untimed process $\mathcal{R}$ has no outputs and the same inputs as A, with an additional input specifying the location of B. For notational convenience, an input event of $\mathcal{R}$ will be given as an input event of A together with the old and new locations of B, s^B and t^B, denoted as $\langle \mathbf{f} + s^B, \mathbf{f}' + t^B \rangle$. The location of $\mathcal{R}$ keeps track of the location of A and the equivalence class of the clock valuations of A and B. Let $[\Phi]$ denote the equivalence class of $\equiv$ that Φ belongs to. The locations $S^{\mathcal{R}}$ of $\mathcal{R}$ are given by

$$\{\langle s^A, [\Phi^A, \Phi^B] \rangle \mid s^A \in S^A \text{ and } \Phi^A \text{ and } \Phi^B \text{ are } X^A \text{ and } X^B \text{ valuations } \} \cup \{Bad\}$$

The initial locations of $\mathcal{R}$ are all locations of the form $\langle s_0^A, [\mathbf{0}, \mathbf{0}] \rangle$ where $s_0^A \in S_0^A$.

Consider a sequence of (untimed) observation events $\overline{\gamma'} = \gamma'_0, \gamma'_1, ..., \gamma'_{k-1}$ where $\gamma'_i = \langle \mathbf{f}_i, \mathbf{f}'_i \rangle$. Note that there are many timed event sequences $\overline{\gamma} = \gamma_0, \gamma_1, ..., \gamma_{k-1}$ such that for all i, $\gamma_i = \langle \delta_i, \mathbf{f}_i, \mathbf{f}'_i \rangle$ for some $\delta_i > 0$. Imagine a run of timed processes A and B on $\overline{\gamma}$, where $\overline{s} = s_0^B, s_1^B, ..., s_k^B$ is such that s_i^B is the location of B after timed event γ_i. Each location that $\mathcal{R}$ can reach on $\overline{\gamma'}$ and $\overline{s}$ corresponds to some such $\overline{\gamma}$ and carries the information about the location that A reaches and the equivalence classes that the clock values of A and B are in.

$\mathcal{R}$ has an edge from $\langle s^A, [\Phi^A, \Phi^B] \rangle$ to $\langle t^A, [\Theta^A, \Theta^B] \rangle$ on observation event $\gamma' = \langle \mathbf{f} + s^B, \mathbf{f}' + t^B \rangle$ iff for some timed event $\gamma = \langle \delta, \mathbf{f}, \mathbf{f}' \rangle$ we have (1) $\langle s^A, \Phi^A \rangle \xrightarrow{\gamma} \langle t^A, \tilde{\Theta}^A \rangle$ and (2) $\langle s^B, \Phi^B \rangle \xrightarrow{\gamma} \langle t^B, \tilde{\Theta}^B \rangle$ for some $[\tilde{\Theta}^A, \tilde{\Theta}^B] = [\Theta^A, \Theta^B]$. $\mathcal{R}$ has a transition to location Bad from $\langle s^A, [\Phi^A, \Phi^B] \rangle$ on γ' if for some γ as above (1) holds for some $\tilde{\Theta}^A$ and (2) does not hold for any $\tilde{\Theta}^B$. Bad has a self-loop on all observation events.

By Lemma 4.2.9, the choice of the representatives from equivalence classes is immaterial, therefore the construction of $\mathcal{R}$ outlined above is well-defined. Observe that, since clock predicates on edges cannot distinguish clock valuations that belong to the same equivalence class, for the choice of the time increment of the timed event, one need only consider values of δ that lead to distinct equivalence classes, i.e., different $[\Phi^A + \delta, \Phi^B + \delta]$. Given one clock valuation, the least time increment that will lead to a new equivalence class is $1 - u_{max}$, where u_{max} is the largest fractional part of the clocks.

4.3.3 Soundness of the Algorithm

Theorem 4.3.2 *Let h be a mapping from A to B and let $h' : (S^{\mathcal{R}} \setminus \{Bad\}) \to S^B$ given by $h'(\langle s^A, [\Phi^A, \Phi^B]\rangle) = h(s^A)$. h preserves timed behavior iff, assuming that whenever $\mathcal{R}$ is at s B is at the location $h'(s)$, (1) h' preserves untimed behavior and (2) Bad is unreachable in $\mathcal{R}$.*

Proof:

($\Rightarrow$) Assume that h preserves timed behavior. Let $\overline{\gamma'} = \gamma'_0, \gamma'_1, ..., \gamma'_{k-1}$ be a sequence of observation events for A. We show by induction on the length of $\overline{\gamma'}$ that conditions (1) and (2) of the theorem are not violated by any run on $\overline{\gamma'}$. If $k = 0$, then both claims hold trivially. Assume that we have the desired properties for k. Consider $k+1$. By the induction assumptions on (1) and (2), observe that all runs of $\mathcal{R}$ on $\gamma'_0, \gamma'_1, ..., \gamma'_{k-1}$ where the location of B is given by h' must have the form $t_0, ..., t_k$ where $t_i = \langle s_i, [\tilde{\Phi}_i, \tilde{\Theta}_i]\rangle$. Let us take one such run. By the construction of $\mathcal{R}$, we can find runs $\langle s_0, \Phi_0\rangle \stackrel{\gamma_0}{\to} \langle s_1, \Phi_1\rangle \stackrel{\gamma_1}{\to} ... \stackrel{\gamma_{k-1}}{\to} \langle s_k, \Phi_k\rangle$ of A, and $\langle h(s_0), \Theta_0\rangle \stackrel{\gamma_0}{\to} \langle h(s_1), \Theta_1\rangle \stackrel{\gamma_1}{\to} ... \stackrel{\gamma_{k-1}}{\to} \langle h(s_k), \Theta_k\rangle$ of B such that $[\Phi_i, \Theta_i] = [\tilde{\Phi}_i, \tilde{\Theta}_i]$ for $1 \leq i \leq k$ where $\gamma_i = \langle \delta_i, \mathbf{f}_i, \mathbf{f}'_i\rangle$ for all i. Now consider γ'_k. If $\mathcal{R}$ has a transition on γ'_k, then there must exist $\gamma_k = \langle \delta_k, \mathbf{f}_k, \mathbf{f}'_k\rangle$ such that $\langle s_k, \Phi_k\rangle \stackrel{\gamma_k}{\to} \langle s_{k+1}, \Phi_{k+1}\rangle$. Let us pick any such transition. Since h preserves timed behavior and since h and $\gamma_0, ..., \gamma_{k-1}$ uniquely determine B's run up to $\langle h(s_k), \Theta_k\rangle$, we must have $\langle h(s_k), \Theta_k\rangle \stackrel{\gamma_k}{\to} \langle h(s_{k+1}), \Theta_{k+1}\rangle$. Thus, conditions (1) and (2) are satisfied for k as well.

($\Leftarrow$) Assume (1) and (2). Let $\gamma_0, \gamma_1, ..., \gamma_{k-1}$ be a timed event sequence, where $\gamma_i = \langle \delta_i, \mathbf{f}_i, \mathbf{f}'_i\rangle$ for all i. We will proceed by induction on k. For $k = 0$, ($\Leftarrow$) holds by the fact that initial locations are mapped to initial locations by h. Assume that ($\Leftarrow$)

holds for k. Consider $k+1$. Let $\langle s_0, \Phi_0\rangle \stackrel{\gamma_0}{\rightarrow} \langle s_1, \Phi_1\rangle \stackrel{\gamma_1}{\rightarrow} ... \langle s_k, \Phi_k\rangle \stackrel{\gamma_k}{\rightarrow} \langle s_{k+1}, \Phi_{k+1}\rangle$ be a run of A. By the induction assumption, there exist $\Theta_0, ..., \Theta_k$ such that $\langle h(s_0), \Theta_0\rangle \stackrel{\gamma_0}{\rightarrow} \langle h(s_1), \Theta_1\rangle \stackrel{\gamma_1}{\rightarrow} ... \stackrel{\gamma_{k-1}}{\rightarrow} \langle h(s_k), \Theta_k\rangle$ is a run of B. Then, by (2) and the construction of $\mathcal{R}$, $t_0 \stackrel{\gamma'_0}{\rightarrow} t_1 \stackrel{\gamma'_1}{\rightarrow} ... \stackrel{\gamma'_{k-1}}{\rightarrow} t_k$ is in $\mathcal{R}$, where $\gamma'_i = \langle \mathbf{f}_i, \mathbf{f}'_i\rangle$ and $t_i = \langle s_i, [\Phi_i, \Theta_i]\rangle$ for all i, and the location of B is specified by h'. Since $\langle s_k, \Phi_k\rangle \stackrel{\gamma_k}{\rightarrow} \langle s_{k+1}, \Phi_{k+1}\rangle$ in A, by the construction of $\mathcal{R}$, $t_k \stackrel{\gamma'_k}{\rightarrow} t_{k+1}$ for some t_{k+1}, while the location of B changes to $h(s_{k+1})$ from $h(s_k)$. By (2), Bad is unreachable from t_k in this case. Then, by the construction of $\mathcal{R}$ and (1), there must exist Θ_{k+1} such that $\langle h(s_k), \Theta_k\rangle \stackrel{\gamma_k}{\rightarrow} \langle h(s_{k+1}), \Theta_{k+1}\rangle$, which is what was required.

■

The size of $\mathcal{R}$ is exponential in the number of clocks in A and B and the constants used in the timing predicates, and linear in $|S_A \times S_B|$. The complexity of checking (1) and (2) in the theorem is then linear in the reachable part of $\mathcal{R}$.

4.3.4 Implementation

We have implemented an algorithm within the COSPAN [Kur94] system to construct $\mathcal{R}$ from A and B. COSPAN supports timing verification and homomorphism checks (see [AK96] for an overview). Our implementation integrates these two capabilities. Given $\mathcal{R}$ and B, conditions (1) and (2) of Theorem 4.3.2 can be checked simultaneously by COSPAN by an on-the-fly search. If (1) fails, this points to the fact that h does not preserve untimed behavior in the reachable parts of A and B. If (2) fails, on the other hand, then B can not follow A due to timing constraints. In the coming chapters, we will use this algorithm to check whether one timed automaton is a correct refinement of another.

4.4 Related Research

In recent years, many tools have been developed to support verification of real-time systems (for instance, ORBITS [Rok93], KRONOS [DOY94], timed COSPAN [AK96] and UPPAAL [LPY95]). These tools consider the problem of model checking, that is, verifying that a mathematical model of a system satisfies its specification. The problem of proving refinements of timed systems has been considered only in the context of manual proofs (see, for instance, [AL91, Sha92, LA92]).

For timed systems, a variety of implementation relations can be considered. Timed language containment is undecidable [AD94]. Timed bisimulation is decidable [Čerāns92], but is not appropriate for refinements. Time-abstract simulation is considered in [HHK95], but does not preserve timed properties. Timed simulations can be viewed as a special case of the timed modal specifications of [ČGL93], but this study neither provides an algorithm nor a proof of decidability for timed simulation.

The use of simulation mappings or homomorphisms to prove refinements is common in the literature (for instance, [LT87, AL91, Sha92, Kur94]). Among automated tools, COSPAN provides support to check whether the user-supplied mappings actually define a homomorphism. The algorithm presented in Section 4.3 generalizes this to checking the preservation of timed behaviors.

4.5 Conclusions

In this chapter, we proved that the problem of checking the existence of timed simulation relations is decidable, generalized the notion of state homomorphisms to timed automata, and gave an algorithm to check if a given map between the locations of two automata preserves timed behavior. The proposed algorithm was implemented in COSPAN, and used in the verification of two asynchronous circuits, to be described in Section 5.4 and Chapter 6.

Unless one has a good understanding of the two systems involved in the implementation check, it can be difficult to specify a correct homomorphism and usually several iterations of specification and verification are required. Therefore, it is valuable to have the capability to check for timed simulation relations without the user needing to provide the relation. However, such an algorithm is likely to be too complex in practice. It may be possible to devise heuristics for certain application domains.

The verification method described in Section 4.3 combines the region automaton approach for timing verification and homomorphism checking for the verification of abstractions. In principle, other methods for performing these tasks could be combined in a similar fashion, and may yield more efficient algorithms.

We believe the results reported in this chapter will be useful in applications such as circuit timing verification and hardware software co-design, where the capability of having abstract descriptions that include timing information is valuable.

The bottleneck in hierarchical verification of real-time systems is most often the size of the representation for the refined system model, as the case studies in Chapters 5 and 6 will demonstrate. A monolithic representation for practical systems is often too large for non-hierarchical *and* hierarchical verification algorithms. The main purpose of performing verification hierarchically, in fact, is to avoid constructing such a representation. This is achieved by the use of a compositional strategy. The hierarchical verification problem is divided into sub-problems, each corresponding to a well-defined sub-module of the system. These "lemmas" are then put together to form a proof of correct implementation for the entire system. The next chapter explores a compositional verification paradigm for alleviating verification complexity of real-time systems.

Chapter 5

Modular Verification of Abstractions

5.1 Introduction

A typical electronic system consists of coordinating components, each of which implements a certain aspect of the system's function. The size of the system's state space grows roughly proportionately to the product of the sizes of the components' state spaces. This fact, referred to as the "state space explosion problem" limits the size of systems that can be handled by state space traversal-based automatic verification algorithms. In fact, almost all practical designs today[1], when modeled in their entirety, are beyond the reach of the state-of-the-art verification tools. For real-time systems, this complexity is compounded by the component of the state-space that represents the timing information. The hierarchical verification algorithms presented in the previous chapter operate by traversing the state-spaces of region automata, and therefore suffer from state space explosion.

The aim of compositional verification is to decompose the problem of verifying refinement for a system into subproblems so that a monolithic representation for the system does not have to be built. The decomposition is typically inferred from the physical or logical structure of the system. Refined and abstract models are described as compositions of modules, where each module corresponds to one component in the system. To prove that a refined system with two components A and B implements an abstraction consisting of

[1] With the exception of a few circuits that are well suited for a particular data structure or verification technique, such as the bus protocol of [CGH+93]

two components C and D, the compositionality principle asserts that it suffices to prove A implements C and B implements D separately, thereby avoiding computation on $A \parallel B$.

For practical applicability of hierarchical verification, it is essential for the implementation relation to be compositional. However, the compositionality principle by itself is often not powerful enough to support a modular refinement proof. In the stronger, and often more useful assumption-guarantee rule one proves that (1) the composition of A and D implements C, and (2) the composition of B and C implements D from which one infers that $A \parallel B$ implements $C \parallel D$. Three observations about this rule are important. First, the rule addresses the issue that the environment of A has to be suitably constrained in order to implement C, and similarly for B. Second, the rule still avoids operating on the compound implementation $A \parallel B$, which typically has the largest of the involved state spaces. Third, unlike the compositionality principle, the assume-guarantee principle is circular and, as such, not propositionally valid. Its proof requires induction on the lengths of traces and a non-blocking restriction on module descriptions; each module must be able to react to any input and not block the progress of time. This chapter investigates assume-guarantee style proof rules for real-time systems.

In Section 5.2 we introduce the non-blocking property and prove the soundness of assume-guarantee style verification for non-blocking systems, taking timed language inclusion as the most general notion for implementation. In Section 5.3 we explore assume-guarantee style verification with simulation as the implementation preorder. Section 5.4 is dedicated to a case study on compositional and hierarchical techniques: the asynchronous Seitz queue element [Rok93] is verified. Section 5.5 discusses related work and Section 5.6 concludes the chapter.

5.2 Modularity in Hierarchical Verification

Let $\preceq$ denote one of $\preceq_S$ or $\preceq_L$, an implementation relation for real-time systems. Then the following compositionality principle holds for hierarchical proofs:

Proposition 5.2.1 (Hierarchy and Compositionality) *If $A \preceq C$ and $B \preceq D$, then $A \parallel B \preceq C \parallel D$.*

While Proposition 5.2.1 is required for modular refinement checks to be sound, it alone is of little use in practice. This is because, for a system consisting of several modules,

the abstract description for each module is correct only in the environment that the module is intended to operate in. In an arbitrary, possibly hostile environment, the refined module may exhibit behavior that is not consistent with the abstract specification, or it may not function at all. To be able to decompose a hierarchical verification problem into sub-problems involving one module each, we need a formal mechanism to incorporate information about the environment of each module into the refinement check. Furthermore, this environment representation must be compact, otherwise, the technique will defeat its purpose.

Toward this end, we investigate assumption-guarantee style reasoning principles for timed abstraction relations. With a certain restriction on system descriptions, we obtain a simple and powerful modularity principle. In the rest of section, timed language inclusion is taken as the notion of timed implementation, and the soundness of the assume-guarantee rule is proven. Section 5.3 achieves the same for timed simulation. First, we define the non-blocking property, aimed at ruling out problematic behavior that a properly modeled system should not exhibit.

5.2.1 The Non-Blocking Property

Definition 5.2.2 (Non-Blocking Property) *A timed automaton* $A = \langle S, S_0, O, I, X, \alpha, \mu, E \rangle$ *is said to be* **non-blocking** *iff for all states* σ *in* Σ_A,

(1) $\sigma \xrightarrow{\gamma}$ *for some timed event* γ, *and*

(2) if $\sigma \xrightarrow{\gamma}$ *for some timed event* $\gamma = \langle \delta, \mathbf{f}, \mathbf{f}' \rangle$ *then* $\sigma \overset{\langle \delta, \mathbf{g}, \mathbf{g}' \rangle}{\rightarrow}$ *for all* $\mathbf{g}$ *and* $\mathbf{g}'$ *with* $\mathbf{g}(O) = \mathbf{f}(O)$ *and* $\mathbf{g}'(O) = \mathbf{f}'(O)$.

The intuition behind this definition is that (1) a non-blocking automaton should be able to generate a trace no matter what sequence input events is applied to it, and (2) the output of a non-blocking timed automaton can not change instantaneously in reaction to an input-event – some non-zero time must elapse.

The execution of a non-blocking timed automaton can be viewed operationally as follows. Consider the timed automaton in state $\sigma = \langle s, \Phi \rangle$ with output $\mathbf{f} = \mu(s)$. The automaton chooses a time delay δ' such that $wait(\sigma, \delta')$, and simultaneously, the environment chooses a time delay δ''. Let δ be the minimum of δ' and δ''. The next observable event happens after a delay of δ. The timed automaton decides to update its output from $\mathbf{f}$

to $\mathbf{f}'$. The environment decides to update the input from $\mathbf{g}$ to $\mathbf{g}'$ independently. The timed automaton updates its state to $\sigma' = \langle s', \Phi' \rangle$ with $\mu(s') = \mathbf{f}'$ by choosing an appropriate edge. Such an update always exists due to the non-blocking requirement. Thus, while the update of outputs is independent of the update of inputs, the update of internal variables (i.e., the location and the clocks) depends on it. The non-blocking property requires that if there is a cycle in input-output variable dependencies, then there is a non-zero delay involved in this cycle. All properly modeled physical systems should satisfy the non-blocking property, as all interaction takes some non-zero time and an actual system always has some way of reacting to an input event.

The assume-guarantee rule to be proven in Section 5.2.2 requires that a system consist of non-blocking modules. For compositional proofs, it is important for the non-blocking property to be preserved upon composition.

Proposition 5.2.3 (Closure under composition) *If A and B are non-blocking, then so is $A \parallel B$.*

Proof: Let $A = \langle S^A, S_0^A, O^A, I^A, X^A, \alpha^A, \mu^A, E^A \rangle$ and $B = \langle S^B, S_0^B, O^B, I^B, X^B, \alpha^B, \mu^B, E^B \rangle$ be non-blocking. We will show that requirements (1) and (2) of the non-blocking property hold for $A \parallel B$. The input and output variables of A and B are partitioned as given in Figure 5.1 to simplify the presentation.

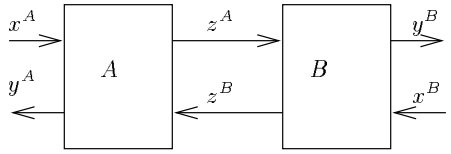

Figure 5.1: The partitioning of the input and output variables of A and B. $z^A = O^A \cap I^B$, $y^A = O^A \setminus z^A$. Similarly for B.

(1) Let $\sigma = \langle \sigma^A, \sigma^B \rangle$ be a state of $A \parallel B$. Since A and B are both non-blocking, there exist timed events $\gamma^A = \langle \delta^A, \mathbf{f}^A, \mathbf{g}^A \rangle$ and $\gamma^B = \langle \delta^B, \mathbf{f}^B, \mathbf{g}^B \rangle$ such that $\sigma^A \xrightarrow{\gamma^A}$ and $\sigma^B \xrightarrow{\gamma^B}$. Without loss of generality, let $\delta^A \leq \delta^B$. We consider the following two cases separately.

$\delta^A < \delta^B$ The fact that A is non-blocking implies that there exists a transition out of σ^A on the timed event $\eta^A = \langle \delta^A, \mathbf{h}^A, \mathbf{k}^A \rangle$ where $\mathbf{h}^A = \mathbf{f}^A(x^A \cup y^A \cup z^A) \cup \mathbf{f}^B(z^B)$ and

$\mathbf{k}^A = \mathbf{g}^A(x^A \cup y^A \cup z^A) \cup \mathbf{f}^B(z^B)$. Also, since $\delta^A < \delta^B$ and B is non-blocking, there is a transition out of σ^B on the timed event $\eta^B = \langle \delta^A, \mathbf{h}^B, \mathbf{k}^B \rangle$ where $\mathbf{h}^B = \mathbf{f}^A(z^A) \cup \mathbf{f}^B(x^B \cup y^B \cup z^B)$ and $\mathbf{k}^B = \mathbf{g}^A(z^A) \cup \mathbf{f}^B(x^B \cup y^B \cup z^B)$. η^A and η^B are consistent on the variables that are common to their support. Therefore, there exists a transition out of the composed state $\langle \sigma^A, \sigma^B \rangle$ on the timed event $\eta = \langle \delta^A, \mathbf{h}^A \cup \mathbf{h}^B, \mathbf{k}^A \cup \mathbf{k}^B \rangle$.

$\delta^A = \delta^B$ A is non-blocking, therefore there exists a transition out of σ^A on the timed event $\eta^A = \langle \delta^A, \mathbf{h}^A, \mathbf{k}^A \rangle$ where $\mathbf{h}^A = \mathbf{f}^A(x^A \cup y^A \cup z^A) \cup \mathbf{f}^B(z^B)$ and $\mathbf{k}^A = \mathbf{g}^A(x^A \cup y^A \cup z^A) \cup \mathbf{g}^B(z^B)$. By the same reasoning, there is a transition out of σ^B on the timed event $\eta^B = \langle \delta^A, \mathbf{h}^B, \mathbf{k}^B \rangle$, where $\mathbf{h}^B = \mathbf{f}^A(z^A) \cup \mathbf{f}^B(x^B \cup y^B \cup z^B)$ and $\mathbf{k}^B = \mathbf{g}^A(z^A) \cup \mathbf{g}^B(x^B \cup y^B \cup z^B)$. As in the previous case, η^A and η^B are consistent on their common support variables, which implies that there is a transition out of $\sigma = \langle \sigma^A, \sigma^B \rangle$ on the timed event $\eta = \langle \delta^A, \mathbf{h}^A \cup \mathbf{h}^B, \mathbf{k}^A \cup \mathbf{k}^B \rangle$.

(2) Let $\sigma = \langle \sigma^A, \sigma^B \rangle$ be a state of $A \parallel B$ and let $\sigma \xrightarrow{\eta}$ for some timed event $\eta = \langle \delta, \mathbf{h}, \mathbf{k} \rangle$. Let $\tilde{\eta} = \langle \delta, \tilde{\mathbf{h}}, \tilde{\mathbf{k}} \rangle$ such that $\mathbf{h}(O^{A \| B}) = \tilde{\mathbf{h}}(O^{A \| B})$ and $\mathbf{k}(O^{A \| B}) = \tilde{\mathbf{k}}(O^{A \| B})$. Since A and B are both non-blocking, and since η and $\tilde{\eta}$ differ only in x^A and x^B, there is a transition out of σ^A on $\tilde{\eta}^A = \langle \delta, \tilde{\mathbf{h}}(I^A \cup O^A), \tilde{\mathbf{k}}(I^A \cup O^A) \rangle$ and there is a transition out of σ^B on $\tilde{\eta}^B = \langle \delta, \tilde{\mathbf{h}}(I^B \cup O^B), \tilde{\mathbf{k}}(I^B \cup O^B) \rangle$. Thus, there is a transition out of σ on $\tilde{\eta}$, which is what was required.

■

5.2.2 Assume-Guarantee Style Verification

The following theorem provides the formal justification for assume-guarantee style verification.

Theorem 5.2.4 (Assumption-Guarantee) *For non-blocking timed automata, if* $A \parallel D \preceq_L C$ *and* $C \parallel B \preceq_L D$ *then* $A \parallel B \preceq_L C \parallel D$.

Observe the apparent circularity in the rule: to prove that $A \parallel B$ is a refinement of $C \parallel D$, it suffices to prove that (1) A is a refinement of C assuming that the environment behaves like D, and (2) B is a refinement of D assuming that the environment behaves like C. The proof relies on the fact that all automata are non-blocking. Let us make a

few observations before we give a detailed proof. First, the rule is incorrect if we remove the requirement of non-blocking. Second, recall that we have required in Section 2.3 all X-predicates to be closed, (i.e. all invariants labeling the locations are conjunctions of non-strict inequalities). If we allow predicates that define open sets (e.g. invariants of the form $x < 5$), the rule fails again. In the proof below, we show that a timed automaton cannot force infinitely many transitions within a finite interval (the so-called condition of non-Zenoness). This does not hold automatically with open invariants. In this case, the non-blocking requirement needs to be strengthened by replacing requirement (1) in the definition of non-blocking using games. In [GSSAL94], such a framework is developed for (asynchronous) timed I/O automata. However, this complicates the development considerably Indeed, the proof that the non-blocking requirement is preserved under composition runs many pages in [GSSAL94].

Proof: The proof is by contradiction. Assume that there exists a timed event sequence $\overline{\gamma} = \gamma_0, \gamma_1, ..., \gamma_{n-1}$ in $\mathcal{L}(A \parallel B)$ and not in $\mathcal{L}(C \parallel D)$. Let $\overline{\gamma_k} = \gamma_0, \gamma_1, ..., \gamma_{k-1}$, where $k < n$ be the longest prefix of $\overline{\gamma}$ that is a trace of $C \parallel D$. Let $\tau_0 \stackrel{\gamma_0}{\rightarrow} \tau_1 \stackrel{\gamma_1}{\rightarrow} ... \stackrel{\gamma_{k-1}}{\rightarrow} \tau_k$ be in $C \parallel D$. Since $\overline{\gamma} \in \mathcal{L}(A \parallel B)$, we have a run in $A \parallel B$ of the form $\sigma_0 \stackrel{\gamma_0}{\rightarrow} ... \stackrel{\gamma_{k-1}}{\rightarrow} \sigma_k \stackrel{\gamma_k}{\rightarrow} ... \stackrel{\gamma_{n-1}}{\rightarrow} \sigma_n$. Let $\sigma_i = \langle \sigma_i^A, \sigma_i^B \rangle$ and $\tau_i = \langle \tau_i^C, \tau_i^D \rangle$ for all i. From the above, we have

$$\langle \sigma_0^A, \tau_0^D \rangle \stackrel{\gamma_0}{\rightarrow} \langle \sigma_1^A, \tau_1^D \rangle \stackrel{\gamma_1}{\rightarrow} ... \stackrel{\gamma_{k-1}}{\rightarrow} \langle \sigma_k^A, \tau_k^D \rangle \text{ in } A \parallel D$$

and

$$\langle \tau_0^C, \sigma_0^B \rangle \stackrel{\gamma_0}{\rightarrow} \langle \tau_1^C, \sigma_1^B \rangle \stackrel{\gamma_1}{\rightarrow} ... \stackrel{\gamma_{k-1}}{\rightarrow} \langle \tau_k^C, \sigma_k^B \rangle \text{ in } C \parallel B$$

The partitioning of the input and output variables of the automata in Figure 5.2 will be

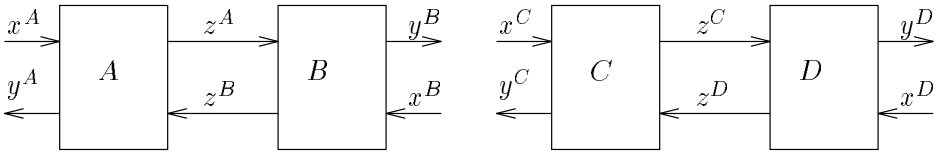

Figure 5.2: The partitioning of the input and output variables.

used in the rest of the proof. For $I \cup O$ valuations $\mathbf{f}$ and $\mathbf{g}$, we say that $\mathbf{f}$ and $\mathbf{g}$ *agree* (or are *consistent*) on $P \subseteq I \cup O$ iff $\mathbf{f}(P) = \mathbf{g}(P)$. For ease of notation, let σ_k^A, σ_k^B, τ_k^C and τ_k^D be denoted by $\boldsymbol{a}$, $\boldsymbol{b}$, $\boldsymbol{c}$ and $\boldsymbol{d}$ respectively. Also let $\gamma_k = \gamma = \langle \delta, \mathbf{f}, \mathbf{f}' \rangle$. We have $\langle \boldsymbol{a}, \boldsymbol{b} \rangle \stackrel{\gamma}{\rightarrow} \langle \boldsymbol{a}', \boldsymbol{b}' \rangle$ for some $\langle \boldsymbol{a}', \boldsymbol{b}' \rangle \in \Sigma_{A \parallel B}$ since $\overline{\gamma}$ is in the language. Consider $\langle \boldsymbol{a}, \boldsymbol{d} \rangle \in \Sigma_{A \parallel D}$

and $\langle \boldsymbol{c}, \boldsymbol{b} \rangle \in \Sigma_{C \| B}$. From the above, A and B can select an increment of δ at states $\boldsymbol{a}$ and $\boldsymbol{b}$ respectively. The rest of the proof splits into two cases.

Case 1. C and D can both select time increments of δ or greater in states $\boldsymbol{c}$ and $\boldsymbol{d}$. In this case, letting the environment choose the input events x^A and x^B according to γ_k, we have

(I) $\langle \boldsymbol{a}, \boldsymbol{d} \rangle \stackrel{\langle \delta, \mathbf{g}, \mathbf{g}' \rangle}{\rightarrow} \langle \boldsymbol{a}'', \boldsymbol{d}' \rangle$ for some $\boldsymbol{a}''$ and $\boldsymbol{d}'$ such that $\mathbf{g}$ and $\mathbf{g}'$ agree with $\mathbf{f}$ and $\mathbf{f}'$ on x^A, y^A and z^A respectively. Thus $\gamma_0, \gamma_1, ..., \gamma_{k-1}, \langle \delta, \mathbf{g}, \mathbf{g}' \rangle$ is a trace of $A \| D$, and since $A \| D \preceq_L C \| D$, it is a trace of $C \| D$.

(II) $\langle \boldsymbol{c}, \boldsymbol{b} \rangle \stackrel{\langle \delta, \mathbf{h}, \mathbf{h}' \rangle}{\rightarrow} \langle \boldsymbol{c}', \boldsymbol{b}'' \rangle$ for some $\boldsymbol{c}'$ and $\boldsymbol{b}''$ such that $\mathbf{h}$ and $\mathbf{h}'$ agree with $\mathbf{f}$ and $\mathbf{f}'$ on x^B, y^B and z^B respectively. We conclude that $\gamma_0, \gamma_1, ..., \gamma_{k-1}, \langle \delta, \mathbf{h}, \mathbf{h}' \rangle$ is a trace of $C \| D$ by a similar reasoning.

(I) and (II), together with the fact that C and D are non-blocking imply that γ_0, γ_1,- ..., γ_{k-1}, γ_k is a trace of $C \| D$, which contradicts our assumption that $\gamma_0, \gamma_1, ..., \gamma_{k-1}$ is maximal.

Case 2. Either C or D can not select a time increment of δ or greater in states $\boldsymbol{c}$ and $\boldsymbol{d}$. Let δ^C and δ^D be the largest increments that C and D can select and let Δ_0 be the minimum of δ^C and δ^D. Since $\Delta_0 < \delta$, for $\theta_0 = \langle \Delta_0, \mathbf{f}, \mathbf{f} \rangle$, we have $\langle \boldsymbol{a}, \boldsymbol{b} \rangle \stackrel{\theta_0}{\rightarrow} \langle \boldsymbol{a}_0^*, \boldsymbol{b}_0^* \rangle$ for some $\langle \boldsymbol{a}_0^*, \boldsymbol{b}_0^* \rangle$. Then,

(I) $\langle \boldsymbol{a}, \boldsymbol{d} \rangle \stackrel{\langle \Delta_0, \mathbf{g}_0, \mathbf{g}_0' \rangle}{\rightarrow} \langle \boldsymbol{a}_0, \boldsymbol{d}_0 \rangle$ for some $\boldsymbol{a}_0$ and $\boldsymbol{d}_0$ such that $\mathbf{g}_0$ and $\mathbf{g}_0'$ agree with $\mathbf{f}$ on x^A, y^A and z^A. With a similar reasoning to (I) in Case 1, we have that $\gamma_0, \gamma_1, ..., \gamma_{k-1}, \langle \Delta_0, \mathbf{g}_0, \mathbf{g}_0' \rangle$ is a trace of $C \| D$.

(II) $\langle \boldsymbol{c}, \boldsymbol{b} \rangle \stackrel{\langle \Delta_0, \mathbf{h}_0, \mathbf{h}_0' \rangle}{\rightarrow} \langle \boldsymbol{c}_0, \boldsymbol{b}_0 \rangle$ for some $\boldsymbol{c}_0$ and $\boldsymbol{b}_0$ such that $\mathbf{h}_0$ and $\mathbf{h}_0'$ agree with $\mathbf{f}$ on x^B, y^B and z^B. From this, we obtain $\gamma_0, \gamma_1, ..., \gamma_{k-1}, \langle \Delta_0, \mathbf{h}_0, \mathbf{h}_0' \rangle$ as a trace of $C \| D$.

From (I) and (II), we derive that $\gamma_0, \gamma_1, ..., \gamma_{k-1}, \theta_0$ is a trace of $C \| D$.

Note that $\overline{\gamma_k}$ is equivalent modulo stuttering to $\gamma_0, \gamma_1, ..., \gamma_{k-1}, \theta_0, \gamma_k^{(0)}$, where $\gamma_k^{(0)} = \langle \delta - \Delta_0, \mathbf{f}, \mathbf{f}' \rangle$. It suffices to show that the latter is a trace of $C \| D$ to obtain a contradiction.

Consider $\overline{\phi_0} = \gamma_0, \gamma_1, ..., \gamma_{k-1}, \theta_0$ and some run of $C \parallel D$ on this trace. Let $\langle \boldsymbol{c}_0^*, \boldsymbol{d}_0^* \rangle$ be the state reached at the end of this run. We can replace $\overline{\gamma_k}$ by $\overline{\phi_0}$, and $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}$ and $\boldsymbol{d}$ with $\boldsymbol{a}_0^*, \boldsymbol{b}_0^*, \boldsymbol{c}_0^*$ and $\boldsymbol{d}_0^*$ and repeat the argument in the proof so far. If it is the case that it is possible to select a time increment of $\delta - \Delta_0$ at states $\boldsymbol{c}_0^*$ and $\boldsymbol{d}_0^*$, we conclude similarly that $\gamma_0, \gamma_1, ..., \gamma_{k-1}, \theta_0, \gamma_k^{(0)}$ is a trace of $C \parallel D$, which contradicts the maximality of $\gamma_0, \gamma_1, ..., \gamma_{k-1}, \theta_0$. Otherwise, we can obtain $\theta_1 = \langle \Delta_1, \mathbf{f}, \mathbf{f} \rangle$ such that $\Delta_1 < \delta - \Delta_0$ and $\gamma_0, \gamma_1, ..., \gamma_{k-1}, \theta_0, \theta_1$ is a trace of $C \parallel D$. Then, the whole argument can be repeated again. If the automaton terminates after finitely many repetitions, we obtain $\gamma_0, \gamma_1, ..., \gamma_{k-1}, \theta_0, \theta_1, ..., \theta_m, \gamma_k^{(m)}$, where $\gamma_k^{(m)} = \langle \delta - \sum_{i=0}^{m} \Delta_i, \mathbf{f}, \mathbf{f}' \rangle$, which is equivalent modulo stuttering to $\overline{\gamma_k}$, the contradiction we sought. Otherwise, we must have an infinite sequence $\gamma_0, \gamma_1, ..., \gamma_{k-1}, \theta_0, \theta_1, \theta_2, \theta_3, ...$ where $\theta_i = \langle \Delta_i, \mathbf{f}, \mathbf{f} \rangle$ and $\sum_{i=0}^{\infty} \Delta_i < \delta$. We will now show that this can not be the case. Since the sum of the Δ_i's converges, there must exist l such that $\sum_{i=l}^{\infty} \Delta_i < 1$. Let $r > l$ be such that no clock value assumes a non-zero integer value after the edge corresponding to θ_r. Such an r must exist for the following reason: The non-zero integer crossing points corresponding to a clock must be separated by 1 time unit, so there can not be more than one such crossing per clock after θ_r. But there are a finite number of clocks, therefore only finitely many such points, which means there is a last one. We argue that after θ_r, the same set of clock predicates of the form $x \diamond k$ are satisfied. This is because no clocks cross integer boundaries other than 0, and the clocks that get reset after θ_r satisfy $0 < x \leq 1$ on all edges after θ_r. Let us choose any $q > r$. By the preceding argument, Δ_q could have been increased to $\sum_{i=q}^{\infty} \Delta_i$, which contradicts the maximality of Δ_q.

■

Corollary 5.2.5 *For non-blocking timed automata, if* $A \parallel D \preceq_S C \parallel D$ *and* $C \parallel B \preceq_S C \parallel D$ *then* $A \parallel B \preceq_L C \parallel D$.

5.3 An Assume-Guarantee Rule For Checking Simulation

The simulation preorder is useful both as a sufficient condition for language inclusion that can be checked efficiently and in its own right as a notion of refinement that draws a correspondence between transition structures. As argued earlier, for composite systems, due to the exponential explosion of the state space, there is a need for decomposing a simulation

check of the form $P \preceq_s Q$ into simpler simulation checks on the components of P and Q. In this section, we present an assume-guarantee rule that enables such a decomposition. To the best of our knowledge, this is the first assume-guarantee rule for real-time systems that applies to a refinement relation different from language inclusion. We provide a constructive proof: given simulation relations that witness the simulation preorder between components, we provide a procedure for constructing a witness relation for $P \preceq_s Q$. In [HQRT98], we proved the correctness of the assume-guarantee rule in conjunction with the simulation preorder for untimed systems and extended the rule to account for fairness assumptions. This section generalizes this to timed systems.

5.3.1 Motivation

As argued in Section 4.1, the natural choice for the implementation pre-order $\preceq$ is trace containment. In this case, $P \preceq Q$ asserts that every sequence of inputs and outputs that is possible for P is also possible for Q. While trace containment is simple and intuitive, it has several shortcomings. First, it is too costly to check trace containment automatically in most cases even for untimed systems, because the check is exponential in the number of states of Q if Q is non-deterministic [Tas95]. Second, in a top-down design methodology, the system description P fleshes out detail that is left open in system description Q, and there is a much tighter relation between P and Q than trace containment would indicate; namely, each implementation state of P corresponds to a specification state of Q. In fact, [Kur94] argues that language containment without simulation is more often than not due to coincidence rather than systematically correct design.

Trace containment is defined globally, for input-output sequences of arbitrary length, whereas simulation is defined locally, by considering the behavior of pairs of states on observation events. As we observed in Chapter 4, it is this locality in the definition of simulation that leads to the significant computational advantages: if Q is claimed to simulate P, then a witness to this claim can be produced in the form of a relation between states of P and states of Q, and the witness can be efficiently represented as a symbolic simulation and checked for correctness as described in Section 4.2. Witness relations have been widely used in verification tools for untimed systems, under various names like homomorphisms [Kur94] and refinement mappings [AL88, Lyn96]. Even if no witness is available, the existence of a simulation can be checked using the algorithm developed in Chapter 4. However the number

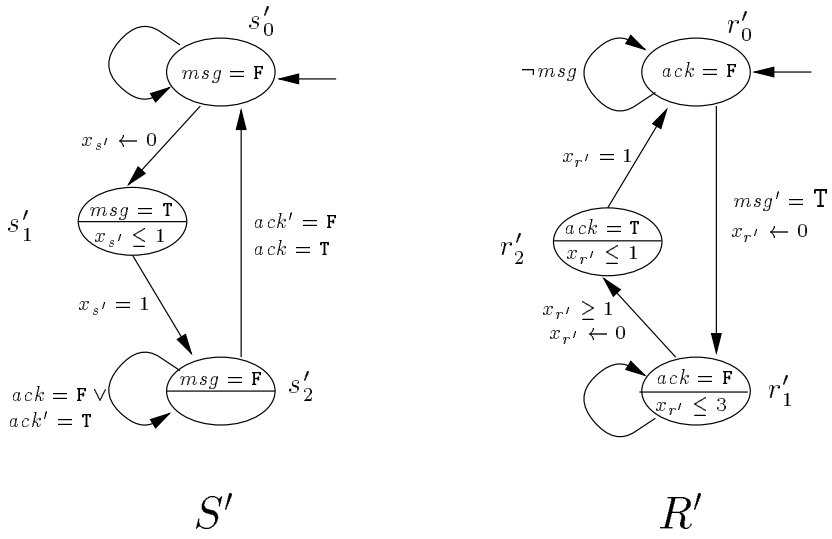

Figure 5.3: Specifications of sender and receiver

of states, as discussed earlier, depends exponentially on the size of the system description and thus, even algorithms that are linear in the number of states are often infeasible in practice. This motivates techniques for dividing a given simulation check into simpler, smaller checks.

Compositional techniques for dividing the verification task $P \preceq Q$ into simpler subtasks are guided by the structures of P and Q. Under certain modeling assumptions (namely, non-blocking and finite nondeterminism), the compositional principle can be strengthened to an *assume-guarantee principle* [Sta85, CLM89, GL94, AL95, AH96, TAKB96, McM97]. In the previous section, we extended the applicability of this rule to non-blocking timed systems with language inclusion as the implementation preorder. By contrast, if the refinement relation $\preceq$ is interpreted as "is simulated by," then little is known about compositional techniques, even for untimed systems. For the reasons outlined above, it is useful to have an assume-guarantee principle for simulation, which, given witnesses for the two subtasks $P_1 \| Q_2 \preceq Q_1$ and $Q_1 \| P_2 \preceq Q_2$, lets us construct a witness for $P \preceq Q$. In this section, we show that under the same modeling assumptions under which the assume-guarantee principle is sound for trace containment, it is also sound for simulation. We achieve this by describing how the compound witness can be constructed from the witnesses for the subtasks.

Let us first illustrate the assume-guarantee rule for simulation using an example.

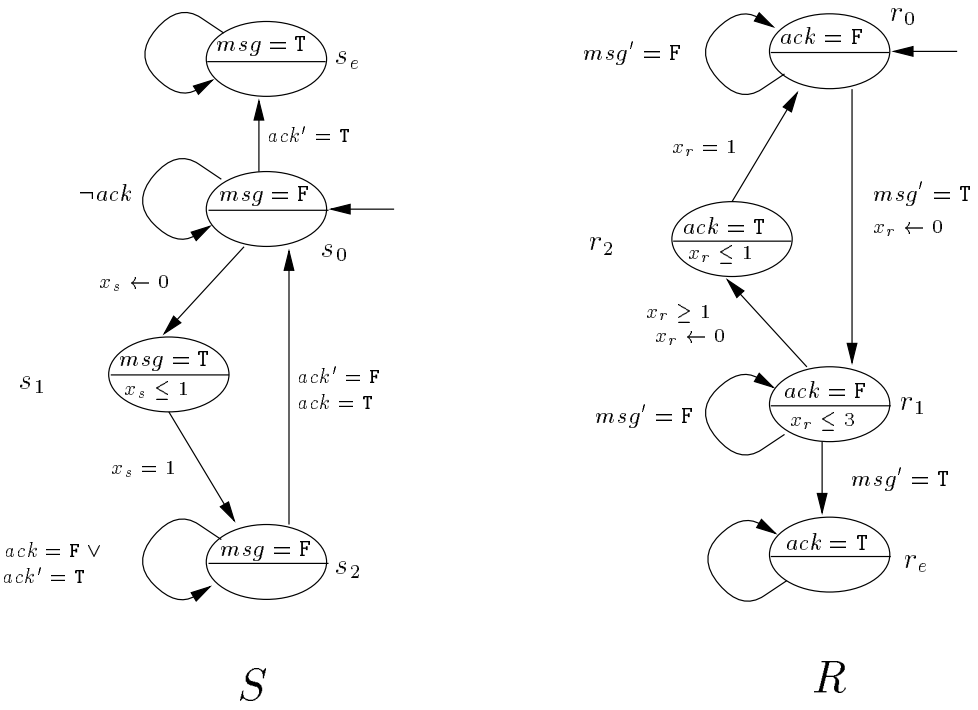

Figure 5.4: Implementations of sender and receiver

Example 5.3.1 The automata in Figure 5.3 are the specifications for a sender (S') and a receiver (R') in a communication protocol. $x_{s'}$ and $x_{r'}$ are the clock variables of S' and R' respectively. The initial states s'_0 of the sender and r'_0 of the receiver are marked with arrows. The sender has one output variable, msg, which becomes true (denoted by T) whenever a message is produced, and one input proposition, namely ack, which, when it takes the value T, acknowledges the receipt of a message by the receiver. The sender starts at s'_0 and stays there for an arbitrary period. It nondeterministically produces a message by moving to s'_1. Once in s'_1, it is forced to move to s'_2 after one time unit. Then, the sender waits in s'_2 until it receives an ack. On receiving the ack it goes back to s'_0 when ack becomes F again. The receiver starts at r'_0 and moves to r'_1 when its input msg becomes T. It stays in r'_1 between 1 to 3 time units and then moves to r'_2. An ack is output at r'_2, and after one time unit the receiver moves back to r'_0.

Figure 5.4 depicts the implementations of the sender (S) and the receiver (R). If S receives an ack while at s_0, it goes to an "error-location" s_e, from which it keeps sending messages continuously. Such behavior is not allowed by the specification S'. Thus, S alone

is not simulated by S'. However, after composing with the specification of the receiver, it is seen that $S \parallel R' \preceq_s S'$, since the acknowledgments can never be received by S while at s_0. The relation $\Theta_S = \{(\langle s_0, r'_0\rangle, s'_0), (\langle s_1, r'_0\rangle, s'_1), (\langle s_2, r'_1\rangle, s'_2), (\langle s_2, r'_2\rangle, s'_2)\}$ is the witness to this simulation, with the interpretation that $(\langle s, r'\rangle, s')$ is shorthand for the following pairs of states

$$\langle\langle s, \Phi\rangle, \langle r', \Theta\rangle, \rangle\langle s', \Psi\rangle\rangle \text{ such that } \Phi(x_s) = \Psi(x_{s'})$$

Similarly, R' does not simulate R, but $S' \parallel R \preceq_s R'$ with the relation $\Theta_R = \{(\langle s'_0, r_0\rangle, r'_0)$,- $(\langle s'_1, r_0\rangle, r'_0), (\langle s'_2, r_1\rangle, r'_1), (\langle s'_2, r_2\rangle, r'_2)\}$ as the witness, using a similar shorthand which requires that x_r and $x_{r'}$ be assigned to the same value. In Section 5.3.2, we prove that the existence of simulation relations from $S \parallel R'$ to S' and from $S' \parallel R$ to R' is sufficient to conclude the existence of a simulation relation Θ from $S \parallel R$ to $S' \parallel R'$, and give a procedure to construct Θ, given Θ_S and Θ_R. For our example, the witness simulation relation is given by $\Theta = \{(\langle s_0, r_0\rangle, \langle s'_0, r'_0\rangle), (\langle s_1, r_0\rangle, \langle s'_1, r'_0\rangle), (\langle s_2, r_1\rangle, \langle s'_2, r'_1\rangle), (\langle s_2, r_2\rangle, \langle s'_2, r'_2\rangle)\}$ Here also it is required that the corresponding clocks in the specification and the implementation be equal. ■

In the next section, we prove the validity of the assume-guarantee rule for the simulation preorder and describe how the compound witness simulation relation is constructed from the witnesses for the components. The results developed apply to other non-blocking, finitely non-deterministic models such as reactive modules [AH96] also.

5.3.2 Soundness

Before proving the result formally, let us provide some intuition into the construction. Suppose we are given a specification $P' \parallel Q'$ and an implementation $P \parallel Q$, where $P \parallel Q' \preceq_s P'$ and $P' \parallel Q \preceq_s Q'$. Consider a specific state (p, q') of $P \parallel Q'$. Suppose that this state is simulated by state p' of P'. Further, suppose that there exists q such that state (p', q) of $P' \parallel Q$ is simulated by state q' of P'. Then, it seems plausible (and indeed it is true, as we show below) that state (p, q) of $P \parallel Q$ is simulated by state (p', q') of $P' \parallel Q'$. Difficulty arises when state (p', q) of $P' \parallel Q$ is simulated by state q'' of Q' that is different from q'. We then examine the state (p, q'') of $P \parallel Q'$ and find a state p'' of P' that simulates it. We continue by finding a state of Q' that simulates state (p'', q) of $P' \parallel Q$. In this process, if we reach a cycle that includes p' and q', we will still be able to show that state (p, q) of $P \parallel Q$ is simulated by state (p', q') of $P' \parallel Q'$.

Theorem 5.3.2 *Let P, Q, P', Q' be timed automata such that $P \parallel Q$ and $P' \parallel Q'$ exist where P is comparable to P' and Q is comparable to Q'. Suppose $P \parallel Q' \preceq_s P'$ and $P' \parallel Q \preceq_s Q'$ with witnessing simulation relations Θ_P and Θ_Q respectively. Then, we can construct a simulation relation Ω from $P \parallel Q$ to $P' \parallel Q'$. Moreover, if Θ_P and Θ_Q are initialized, then so is Ω.*

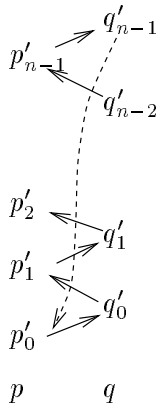

Figure 5.5: Figure demonstrating the definition of Ω.

Proof: Let $\Omega \subseteq (\Sigma_P \times \Sigma_Q) \times (\Sigma_{P'} \times \Sigma_{Q'})$ be defined as as follows. $(\langle p, q\rangle, \langle p', q'\rangle) \in \Omega$ iff there exist $p'_0, p'_1, ..., p'_n \in \Sigma_{P'}$ and $q'_0, q'_1, ..., q'_{n-1} \in \Sigma_{Q'}$ such that (See Figure 5.5)

- $p'_0 = p'_n = p'$ and $q'_0 = q'$, and
- for all $i < n$, we have (1) $(\langle p'_i, q\rangle, q'_i) \in \Theta_Q$, and (2) $(\langle p, q'_i\rangle, p'_{i+1}) \in \Theta_P$.

We will first show that the simulation condition of Section 4.2 is satisfied by every pair of states in Ω. Suppose that $(\langle p, q\rangle, \langle p', q'\rangle) \in \Omega$, i.e., there exist $p'_0, p'_1, ..., p'_n \in \Sigma_{P'}$ and $q'_0, q'_1, ..., q'_{n-1} \in \Sigma_{Q'}$ satisfying the conditions given above. In the following, for a timed event γ, when $\xrightarrow{\gamma}$ is used in conjunction with a timed automaton whose input and output variables are a subset of those of $P \parallel Q$, the projection of γ to the appropriate set of variables is to be inferred.

Let $\langle p, q\rangle \xrightarrow{\xi} \langle \tilde{p}, \tilde{q}\rangle$ for some $\langle \tilde{p}, \tilde{q}\rangle$ and an observation event ξ of $P \parallel Q$. Suppose that $\xi = \langle \delta, \mathbf{f}, \mathbf{g}\rangle$. Define $\gamma = \langle \delta, \mathbf{f} \cup \mu(p), \mathbf{g} \cup \mu(q)\rangle$. Observe that γ includes all inputs and outputs of the machines P, Q, P' and Q'. We want to show that there exists $(\langle \tilde{p}, \tilde{q}\rangle, \langle \tilde{p}', \tilde{q}'\rangle) \in \Omega$ such that $\langle p', q'\rangle, \xrightarrow{\gamma} \langle \tilde{p}', \tilde{q}'\rangle$. We arrive at the desired chain of states through the following steps:

- Since all automata are non-blocking, we have $p'_0 \xrightarrow{\gamma} \tilde{p}'_0$ for some $\tilde{p}'_0$. Hence, $\langle p'_0, q\rangle \xrightarrow{\gamma} \langle \tilde{p}'_0, \tilde{q}\rangle$.
- The fact that $(\langle p'_0, q\rangle, q'_0) \in \Theta_Q$ implies that there exists some $\tilde{q}'_0$ such that $q'_0 \xrightarrow{\gamma} \tilde{q}'_0$ and $(\langle \tilde{p}'_0, \tilde{q}\rangle, \tilde{q}'_0) \in \Theta_Q$.
- $q'_0 \xrightarrow{\gamma} \tilde{q}'_0$ implies that $\langle p, q'_0\rangle \xrightarrow{\gamma} \langle \tilde{p}, \tilde{q}'_0\rangle$.
- $(\langle p, q'_0\rangle, p'_1) \in \Theta_P$, therefore, there exists $\tilde{p}'_1$ such that $p'_1 \xrightarrow{\gamma} \tilde{p}'_1$ and $(\langle \tilde{p}, \tilde{q}'_0\rangle, \tilde{p}'_1) \in \Theta_P$

Repeating this process, we obtain $\tilde{p}'_0, \tilde{p}'_1, \tilde{p}'_2, ...$, and $\tilde{q}'_0, \tilde{q}'_1, \tilde{q}'_2, ...$ such that

- for all $k \in \mathbb{N}$, we have $p'_0 \xrightarrow{\gamma} \tilde{p}'_{kn}$ and $q'_0 \xrightarrow{\gamma} \tilde{q}'_{kn}$, and
- for all i, we have $(\langle \tilde{p}'_i, \tilde{q}\rangle, \tilde{q}'_i) \in \Theta_Q$ and $(\langle \tilde{p}, \tilde{q}'_i\rangle, \tilde{p}'_{i+1}) \in \Theta_P$.

Since there are only finitely many edges out of each location in P' each of which update the clocks deterministically, there exist $a, b \in \mathbb{N}$ with $b > a$ such that $\tilde{p}'_{an} = \tilde{p}'_{bn}$. Consider the states $\tilde{p}'_{an}, \tilde{p}'_{an+1}, \ldots, \tilde{p}'_{bn} \in \Sigma_{P'}$ and $\tilde{q}'_{an}, \tilde{q}'_{an+1}, \ldots, \tilde{q}'_{bn-1} \in \Sigma_{Q'}$. We know that for all $an \leq i < bn$, $(\langle \tilde{p}'_i, \tilde{q}\rangle, \tilde{q}'_i) \in \Theta_Q$ and $(\langle \tilde{p}, \tilde{q}'_i\rangle, \tilde{p}'_{i+1}) \in \Theta_P$. It follows from the definition of Ω that $(\langle \tilde{p}, \tilde{q}\rangle, \langle \tilde{p}'_{an}, \tilde{q}'_{an}\rangle) \in \Omega$. We also have that $\langle p', q'\rangle \xrightarrow{\gamma} \langle \tilde{p}'_{an}, \tilde{q}'_{an}\rangle$. Thus, the simulation condition is satisfied.

Let us now suppose that Θ_P and Θ_Q are initialized simulation relations. To simplify the presentation, let us assume that P, Q, P' and Q' each have a unique initial state. We do not lose any generality with this assumption for the following reason. Suppose a timed automaton A has more than one initial location. Recall that all timers are initialized to 0. We will create a new timed automaton $\tilde{A}$ that will essentially have the same behavior as A but only one initial location, s_0. Formally, $S^{\tilde{A}} = S^A \cup \{s_0\}$ where $s_0 \notin S^A$, and $S_0^{\tilde{A}} = \{s_0\}$. $\tilde{A}$ has the same inputs, outputs, and clocks as A. The location invariants and outgoing edges for all locations in S^A are left intact. We only need to describe the invariants and outgoing edges associated with s_0 in $\tilde{A}$. Let x be a clock of A. The invariant in s_0 is given by $\alpha(s_0) = \{x \leq 1\}$ and there is an outgoing edge from s_0 to each location in $t_0 \in S_0^A$ of the form $\langle s_0, t_0, \varphi, \chi, R\rangle$ where φ is the clock predicate $x = 1$, χ always evaluates to $True$, and $R = X^{\tilde{A}}$, i.e., all clocks are reset to 0. Now observe that the behavior of $\tilde{A}$ starting at time 1 is identical to the behavior of A starting at time 0. Moreover $A \preceq B$ for $\preceq \in \{\preceq_S, \preceq_L\}$ if

and only if $\tilde{A} \preceq \tilde{B}$ [2]. Thus, we can restrict our attention to timed automata with a single initial location.

We now let σ_0^A denote the unique initial state of A. Since Θ_P and Θ_Q are initialized, we have $(\langle \sigma_0^P, \sigma_0^{Q'} \rangle, \sigma_0^{P'}) \in \Theta_P$ and $(\langle \sigma_0^{P'}, \sigma_0^Q \rangle, \sigma_0^{Q'}) \in \Theta_Q$ Let $n = 1$, $p'_0 = p'_1 = \sigma_0^{P'}$ and $q'_0 = \sigma_0^{Q'}$. Then, we have $(\langle \sigma_0^P, \sigma_0^Q \rangle, \langle \sigma_0^{P'}, \sigma_0^{Q'} \rangle) \in \Omega$, thus Ω is also initialized.

■

5.4 Modular Verification Example

This section demonstrates the assumption-guarantee reasoning paradigm developed in this chapter and the timed refinement checking algorithm of the previous chapter on a small asynchronous circuit, the Seitz queue element.

5.4.1 The Seitz Queue Element

The Seitz queue element (Figure 5.6) is a self-timed circuit that constitutes one stage of a first-in first-out (FIFO) queue. It was studied in detail in [Rok93]. The correct functioning of this circuit depends critically on the ranges of gate delays: if the deviation of the gate delays from the specified values exceeds 20%, the circuit does not function correctly [Rok93]. We would like to verify that this circuit functions as a queue element and show that it responds to requests within certain time bounds.

We model each gate by a timed automaton. We use the inertial delay model: each gate is modeled by an ideal, delayless gate followed by a buffer with inertial delay (see Figure 2.1) where d_{min} is the nominal delay of the gate shown in Figure 5.6 and $d_{max} = (1 + \alpha)d_{min}$, α denoting the variation in gate delay. Each transistor was modeled as a pass gate with its gate terminal as the control input. As can be seen in Figure 2.1, each such automaton requires one clock variable. The modeling of the gates marked "H" (for "hazard") is more involved. These gates can have several transitions at their inputs while an output change is still pending, and requires two clocks each. For a detailed description of hazard gates refer to [Rok93].

The circuit consists of four blocks. Data is fed into the queue element through the

[2] Note that we have not specified the output in location s_0. For this claim to hold, the outputs in locations $s_0^{\tilde{A}}$ and $s_0^{\tilde{B}}$ must match. This can be achieved easily by selecting a random output valuation for $s_0^{\tilde{A}}$ and requiring that $s_0^{\tilde{B}}$ have a consistent output.

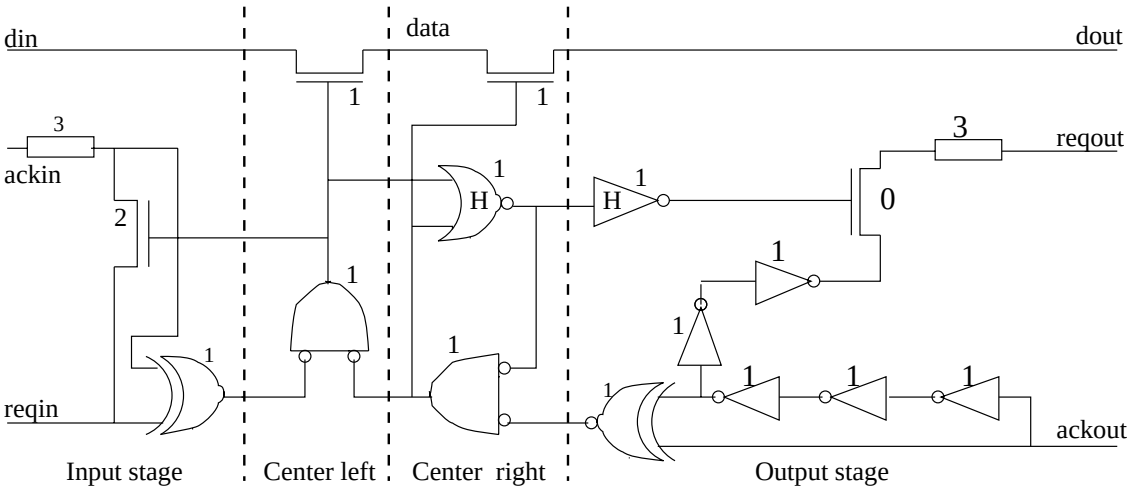

Figure 5.6: The Seitz queue element. The delay for each circuit component is indicated next to it.

block marked "input stage". A request to store new data in the queue element is expressed by toggling the "reqin" signal. The input stage is then responsible for passing on to the rest of the circuit the request for new data to be read in, and signaling an acknowledge to the requester when the queue element is ready to receive the data. Similarly, "output stage" signals a request for passing on the data to the circuit connected to "reqout" by toggling this signal, and when this request is acknowledged and data passed on, signals to the rest of the Seitz circuit that new data can be read in. The FIFO stage stores the data at the node marked "data". The "center left" and "center right" stages control the gates of the two transistors on each side of the data node, thereby controlling when data is read, passed to the output, and isolated from the data that is at "din". The timed automaton representing each block is the composition of the timed automata for its gates and transistors.

5.4.2 Verification Steps

Let $\mathcal{A}_{inp}$, $\mathcal{A}_{cl}$, $\mathcal{A}_{cr}$ and $\mathcal{A}_{out}$ be the timed automata describing the input, center left, center right and output stages respectively. The verification of the FIFO stage consisted of two phases. In the first phase, an abstraction for each circuit block was constructed by hand. We denote these by $\mathcal{A}_{inp}^{abs}$, $\mathcal{A}_{cl}^{abs}$, $\mathcal{A}_{cr}^{abs}$ and $\mathcal{A}_{out}^{abs}$. These abstractions describe the function of each block qualitatively by hiding information about nodes that are internal to the block. In other words, the abstract automata encapsulate the "interface timing

behavior" of each block and do not describe how this behavior is internally realized by the circuit elements. For each block, we verified the correctness of the abstract model by specifying an untimed homomorphism from the circuit level model to the abstract model, in the fashion of Section 4.3[3]. We made extensive use of the assumption-guarantee rule in verifying the abstractions. For each block B, we showed that $\mathcal{A}_B \preceq_L \mathcal{A}_B^{abs}$ in the context given by the abstractions of the rest of the circuit blocks. For instance, for the input stage we showed

$$\mathcal{A}_{inp} \parallel \mathcal{A}_{cl}^{abs} \parallel \mathcal{A}_{cr}^{abs} \parallel \mathcal{A}_{out}^{abs} \quad \preceq_L \quad \mathcal{A}_{inp}^{abs} \parallel \mathcal{A}_{cl}^{abs} \parallel \mathcal{A}_{cr}^{abs} \parallel \mathcal{A}_{out}^{abs} \tag{5.1}$$

The state space associated with the composition of the abstractions $\mathcal{A}_{cl}^{abs} \parallel \mathcal{A}_{cr}^{abs} \parallel \mathcal{A}_{out}^{abs}$ was too large and not all of the detail in this model was needed for showing that $\mathcal{A}_{inp}^{abs}$ is a correct abstraction for $\mathcal{A}_{inp}$. Because of this, instead of checking (5.1) directly, we employed an abstraction of the environment of the input stage, $\mathcal{A}_{inp}^{env}$. We first showed that

$$\mathcal{A}_{cl}^{abs} \parallel \mathcal{A}_{cr}^{abs} \parallel \mathcal{A}_{out}^{abs} \quad \preceq_L \quad \mathcal{A}_{inp}^{env} \tag{5.2}$$

and then we replaced the check in (5.1) with

$$\mathcal{A}_{inp} \parallel \mathcal{A}_{inp}^{env} \quad \preceq_L \quad \mathcal{A}_{inp}^{abs} \parallel \mathcal{A}_{inp}^{env} \tag{5.3}$$

(5.1) can be inferred from (5.3) and (5.2) using the compositionality of $\preceq_L$. The same steps were followed for each of the circuit blocks. In some cases, when checking the equivalent of (5.2) it was possible to "free" some of the automata on the left hand side, i.e., disregard their transition structures and consider them as automata that pose no restriction on their inputs and outputs. This was useful in reducing the computation involved in checking (5.2). The conclusion from the first phase was that

$$\mathcal{A}_{inp} \parallel \mathcal{A}_{cl} \parallel \mathcal{A}_{cr} \parallel \mathcal{A}_{out} \quad \preceq_L \quad \mathcal{A}_{inp}^{abs} \parallel \mathcal{A}_{cl}^{abs} \parallel \mathcal{A}_{cr}^{abs} \parallel \mathcal{A}_{out}^{abs}$$

Observe that assume-guarantee style reasoning is essential for being able to verify the abstract models. None of the abstraction checks would have passed without certain environment assumptions.

In the second phase of the verification, a high level abstraction $\mathcal{A}^{all}$ of the whole circuit was constructed, and it was shown that $\mathcal{A}_{inp}^{abs} \parallel \mathcal{A}_{cl}^{abs} \parallel \mathcal{A}_{cr}^{abs} \parallel \mathcal{A}_{out}^{abs} \preceq_L \mathcal{A}^{all}$ by constructing

[3]The S/R descriptions that were input to COSPAN have not been provided here as they do not provide insight into the proof process. For an example of COSPAN code for timed refinement checking, see the next chapter.

and verifying a homomorphism as before. $\mathcal{A}^{all}$ closely resembles a specification for a FIFO stage with bounds placed on its response time. We will refer to this as the "overall" abstraction. Using $\mathcal{A}^{all}$, it is trivial to verify properties such as no data being lost or requests being responded to within a certain amount of time.

The data from the experiments is listed in Table 5.1. The homomorphism checks were performed within COSPAN using a ordered binary decision diagram based algorithm. COSPAN has the capability to return a counterexample if the desired relation does not hold. This was instrumental in obtaining a correct abstraction by iteration.

	Num. of clocks	Largest timing constant	$\lvert S^{\mathcal{R}} \rvert$ ($\times$1000)	BDD nodes ($\times$1000)	CPU time (seconds)
Input	6	32	.24	15	39
Input (Env)	3	12	6	13	15
Center Left	11	22	18	131	403
Center Right	11	22	19	142	4879
Center (Env)	3	32	.12	7	30
Output	8	22	478	235	1029
Output (Env)	3	12	4.9	15	16
Overall	9	32	101	262	8115

Table 5.1: Tests were run on an SGI machine with 1 GB memory. (Env) denotes the verification corresponding to (5.2) The same environment abstraction was used for both center blocks. Since timing constants must be integers, a 20% variation in a gate delay of k is represented by specifying the delay to be between $5k$ and $6k$.

As a comparison, we attempted to run timed reachability analysis on the complete circuit without using any compositional rules and abstraction. The computation did not complete: the computer ran out of memory (1 GB) after several hours of computation. This is partly due to the following facts: (i) our way of modeling this circuit may not be the most efficient and (ii) our monolithic analysis algorithm did not have many of the improvements in state-of-the-art timing verification tools. However, even if we had addressed (i) and (ii), as long as a monolithic representation for the system in full detail has to be built, algorithms are bound to run out of resources for any electronic system of practical size (say, several cascaded Seitz queue elements). The modular and hierarchical approach, however, enables us to cope with the compositional complexity of verification. With the modular approach, verifying the correctness of an abstract model for 64 FIFO stages requires little more effort

than verifying only one.

Note the following side benefit of the hierarchical verification approach: now that we have an abstract model for the FIFO stage, we can use it to represent the Seitz circuit inside a larger system. By virtue of the hierarchical approach, all properties verified in this manner will be valid at the gate level also.

5.5 Related Research

To be able to support assume-guarantee style reasoning, it is necessary to make an explicit distinction between which variables are updated by the system and which ones are updated by the environment. Our timed automaton model, described in Chapter 2, was constructed with this concern in mind. [LT87]'s timed I/O automata also make this distinction, but this model is geared more towards software applications.

Assume-guarantee style proof rules for untimed systems have been proposed by many researchers (for instance, [GL94, AL93, AH96]). When the rule is symmetric and involves circularity, it is necessary to require that a process is nonblocking [LT87, AH96]. The nonblocking property initially appeared in control theory literature [RW87, RW89]. In the case of timed systems, these issues have been considered for timed I/O automata in [GSSAL94]. By contrast to [GSSAL94], our framework uses synchronous composition and a much simpler definition of nonblocking without resorting to games.

5.6 Conclusions

In the last two chapters, we presented a framework for hierarchical reasoning about real-time systems. Our framework supports modular and compositional verification rules, and provides an algorithm for timed implementation checks. We have demonstrated on the Seitz queue circuit that our techniques can alleviate verification complexity greatly.

In the next chapter, we test our methodology on a larger circuit design: the STARI chip. This chip, designed and implemented by Greenstreet [Gre93], depends for its correct functioning critically on certain timing properties. The experience from this case study provides feedback on our techniques and guides our further research.

Chapter 6

STARI: A Case Study

6.1 Introduction

This chapter describes the application of the techniques and tools presented in Chapters 4 and 5 to the verification of the high-bandwidth communication chip STARI [Gre93].

The STARI circuit (by Greenstreet, [Gre93, Gre]) is a self-timed first-in first-out (FIFO) queue that connects a transmitter and a receiver that operate at the same clock frequency but may have some skew between their clock signals (Figure 6.1). STARI can compensate for large, time varying skews and makes high bandwidth synchronous operation possible by eliminating the need for handshakes between the transmitter and the receiver. However, because the STARI scheme does not use any handshakes, certain timing properties need to be verified to show that the interface functions correctly. In particular, it needs to be shown that no data is duplicated or dropped by the interface.

The FIFO in STARI consists of a cascade of identical stages and thus, the size of the state space representing the FIFO grows roughly as $O(k^n)$, where k is the size of the model for a single stage. If the circuit is modeled at the gate level, k is rather large, and this limits state-space traversal-based automatic verification methods to very small n. As long as the circuit is modeled at this level of detail, improvements to verification algorithms are not likely to have a significant effect, since adding one more stage to the FIFO multiplies the resource requirements by k. Hence, one needs to perform verification on a more abstract representation to be able to handle larger n.

The initial proof of correctness for STARI was performed using a theorem-prover [Gre93, LG95]. An automatic proof was also published [HBAB93]. Both of these stud-

ies operated on simplified, abstract models for STARI which were not proven to be correct and which ignored certain aspects of the circuit implementation. As such, there was no formal assurance that the properties proved on the abstract models were satisfied by the actual circuit implementation. For the approaches mentioned, the use of simplified models was necessary; otherwise the techniques would have become inapplicable or unmanageably complicated.

Instead, using a hierarchical approach, we are able to provide a formal guarantee of correctness for the circuit itself, and model the implementation more faithfully. We proceed as follows:

(i) We construct an abstract model for one stage of STARI, which we prove to be correct in the environment that it operates in.

(ii) By composing n of these abstract models, we obtain an abstract model for STARI on which we prove that the timing properties are satisfied.

The use of multiple levels of abstraction and an inductive argument together with assume-guarantee style reasoning for (i) makes this case study an interesting combination of model checking and theorem proving. The assume-guarantee argument, the abstract model, and the mappings that relate abstract models with the gate level descriptions are constructed manually, whereas the abstraction check and the verification of the timing properties on the abstract model are performed automatically by COSPAN. The automation afforded by COSPAN eliminates the need for having oversimplified abstract models.

This chapter is organized as follows. In Section 6.2 we describe the STARI circuit. Section 6.3 explains how the circuit was modeled using timed automata in COSPAN. Section 6.4 presents the verification of the timing properties and contrasts our method with previous ones. Section 6.5 contrasts our approach with previous ones. Section 6.6 summarizes the experience from this study and suggests further research.

6.2 STARI Operation and Implementation

6.2.1 Overview

Most digital electronic circuits are synchronous, i.e., the state-storing elements make transitions simultaneously as dictated by a clock signal which defines the time step. A high

frequency clock signal can often be safely distributed over relatively large distances, however, it is hard to control the exact phase of the clock at different points in the distribution network. The difference between the phases of the clock signals at two such points is referred to as *skew*. For systems that are not built on a single chip, such as multi-processor systems, ATM networks, etc., skew can be large and time-varying, which makes it a limiting factor on the performance of purely synchronous systems. Even clock distribution across a single chip is becoming a serious concern because of increasing levels of integration. Self-timed systems avoid the synchronization problem by using handshake protocols. For a self-timed system, if no assumptions are made about the delays of circuits and wires, for each data item that is communicated between two parts of a circuit, an acknowledgment needs to be sent back to the transmitter before another data item can be sent. This can limit the communication bandwidth severely, since only one data item can be in transit between these two points at a given time, i.e., two data items need to be separated by the round-trip time between the transmitter and receiver in addition to the response time of the receiver.

STARI (Self-Timed At Receiver's Input) is a hybrid-scheme: it is a self-timed first-in first-out queue (FIFO) that connects a transmitter and receiver. The two communicate as though they were part of a perfectly synchronous system (Figure 6.1 from [Gre93]) despite a possibly time-varying skew.

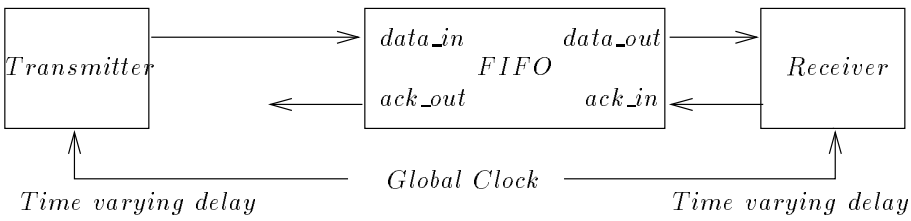

Figure 6.1: The STARI interface

The FIFO is initialized to be roughly half-full, and during each period of the clock, one value is inserted to the FIFO by the transmitter and one value is removed by the receiver. Because data is inserted and removed at the same rate, no control signals are required to prevent underflow and overflow. However, because of variations in clock skew, there can be short term fluctuations in the clock rate at the receiver or transmitter and it can appear that one of them is working faster than the other. STARI responds to these fluctuations by

building up more data in the FIFO when the transmitter is working faster and by supplying data from the FIFO when the receiver is working faster.

For correct operation of the STARI interface, among others, the following two properties need to be proven:

(i) Each data value output by the transmitter must be inserted into the FIFO before the next one is output.

(ii) A new value must be output by the FIFO before each acknowledgment from the receiver.

Intuitively, the longer and faster the FIFO, the more skew it can tolerate. The correctness of the properties above depend on the length of the FIFO, the clock speed, the magnitude of the skew and the speed of operation of FIFO stages. Therefore, it is appropriate to model STARI as a real-time system and perform formal verification. The original correctness proof by [Gre93, LG95] verifies on an abstract real-time model of the FIFO that if a certain relationship holds between these parameters, then properties (i) and (ii) are satisfied.

In the rest of this section we present the operation of the STARI interface and the circuit implementation.

6.2.2 Dual-rail Coding

In the STARI circuit, each Boolean signal `x` that carries data is represented by two Boolean signals `x.t` and `x.f`. `x.t` and `x.f` correspond to wires in the circuit and their values are given by the dual-rail code depicted in Figure 6.2. The "empty" value is needed to distinguish between two consecutive data items of the same value and one data value asserted for a long time.

6.2.3 A High-Level View of STARI

According to the STARI scheme, the transmitter outputs a data stream, updating the value of *data_in* in Figure 6.1 at each rising edge of its clock input. After each time the transmitter outputs a `T` or `F`, it outputs an `E` to separate the current data item from the next one. The receiver samples its input (*data_out* in Figure 6.1) each time its clock signal goes high, and updates its acknowledge signal after some delay (*ack_in* in Figure 6.1).

x.t	x.f	x
0	0	E (empty)
0	1	F (false)
1	0	T (true)
1	1	illegal

Figure 6.2: Dual rail encoding

The FIFO consists of n identical stages, each of which holds a single data value (See Figure 6.3 for an example). Suppose that stage k of the FIFO holds data value d_{old} and that a new data value d_{new} is applied to its inputs by stage $k-1$. Stage k is "enabled" to read and hold the value d_{new} if stage $k+1$ has copied the value d_{old} and has notified stage k. After some delay, stage k takes on the value d_{new} and sends an acknowledgment of this fact to stage $k-1$. Stage k holds this value until a different value is applied to its input by stage $k-1$ and stage $k+1$ has acknowledged reading d_{new}. The same data value in the input sequence can be held by many adjacent stages simultaneously. This is referred to as a *bubble*. For instance, in Figure 6.3, stage 5 initially has a bubble, which gets filled by the data value F. When this happens, the bubble at stage 5 moves back to stage 4. Bubbles enable the FIFO to hold fewer than n data values (Figure 6.3). The FIFO operates by moving data items forward and bubbles backward.

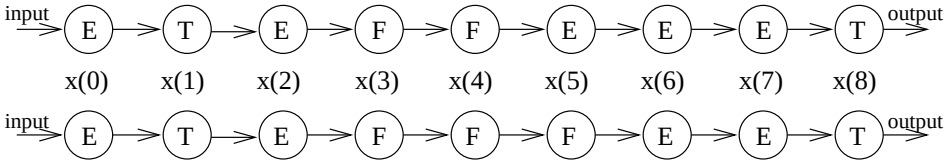

Figure 6.3: The FIFO holds the sequence "E,T,E,F,E,T". Stage 5 is enabled and copies the data value "F".

6.2.4 One Stage of the FIFO

Each stage of the FIFO consists of two Muller C-elements that hold the value of the .t and .f components of a data item, and a NOR gate that computes the acknowledge output signal of the stage (Figure 6.2.4). A Muller C-element works as follows: when the two inputs are the same, the output takes on this value, when the inputs are different,

the output retains its previous value (Figure 6.4). To understand how data flows down

Input 1	Input 2	Output
0	0	0
0	1	unchanged
1	0	unchanged
1	1	1

Figure 6.4: Muller C-element functionality.

the FIFO, first note that stage k is said to have "acknowledged" the data it holds if its `ack_out` output is equal to the `NOR` of `x(k).t` and `x(k).f`. Thus, the copying of data value `E` is acknowledged by asserting `ack_out = 1` and data values `T` and `F` are acknowledged by `ack_out = 0`. Let us consider a situation where stages k and $k+1$ hold the value `E` and stage $k-1$ has the value `T`. Stage k is enabled to copy the new data from stage $k-1$. We have `ack_out(k+1) = ack_in(k) = 1` and `x(k-1).t = 1` and `x(k-1).f = 0`. Therefore, both inputs to the C-element that computes `x(k).t` are 1, while the output of the C-element, `x(k).t = 0`. After some delay, `x(k).t` becomes 1 while `x(k).f` remains unchanged, and, in this way, the value `T` gets copied to stage k. Transitions from a `T` or `F` to `E` happen similarly.

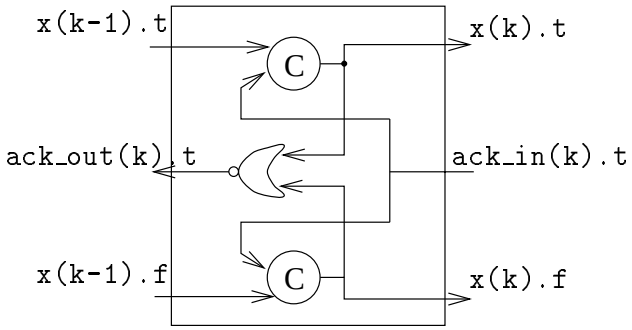

Figure 6.5: Stage k of the FIFO.

Using this circuit description for STARI, we now proceed to verify the two timing properties mentioned earlier.

6.3 Modeling STARI

6.3.1 Specifying Timed Systems in S/R

In the verification of STARI, we use the timed homomorphism checking and state-space exploration algorithms that we implemented in COSPAN. The input language of COSPAN S/R is sufficiently expressive to emulate timed automata. Timing constraints on transitions are specified by using "pause" states. (For a description of the S/R language and pause states, refer to [Gla95] and [Kur94]). Consider the following fragment from S/R code specifying the buffer of Figure 6.6.

one_pending {% { δ_{min}, δ_{max} } go_one}
-> one: (outp=go_one) * (_inp = 1)
-> $: else * (_inp = 1)

The interpretation of this code fragment is follows: When the automaton is at location *one_pending*, the output variable # (referred to as a "selection variable" in S/R) must take on the value "%" for at least δ_{min} time units. "%" is referred to as the "pausing selection". After δ_{min} time units, the value of # is selected non-deterministically from the set $\{\%, go\_one\}$. After δ_{max} time units, # has the value "go_one". By making outgoing transitions conditional on the value of #, one is able to pose timing constraints in the same manner as timed automata.

6.3.2 Modeling Clock Skew

The global clock source has period π and remains low for the initial half of the period and is high during the second half. The delays of the paths that the clock signal travels from the source to the clock inputs of the transmitter and the receiver are modeled using buffers with delays (see Figure 6.6) in the ranges $[\delta_{tr,min}, \delta_{tr,max}]$ and $[\delta_{rx,min}, \delta_{rx,max}]$, respectively. Thus, the rising edge of the transmitter's clock input can be as much as $\delta_{rx,max} - \delta_{tx,min}$ ahead of or $\delta_{tx,max} - \delta_{rx,min}$ behind the rising edge of the receiver's clock input. By choosing these parameters properly, any amount of skew between the clock inputs can be modeled.

The S/R code describing the buffer in Figure 6.6 can be found in Section A.1. In the shaded locations, the input of the buffer is different from its output, and a change in the output value is pending. The buffer moves to location "*Bad*" if the input of the buffer switches back to its starting value while the buffer is in one of the shaded locations. This represents a hazard, and no hazards are expected in the STARI design. However, if one

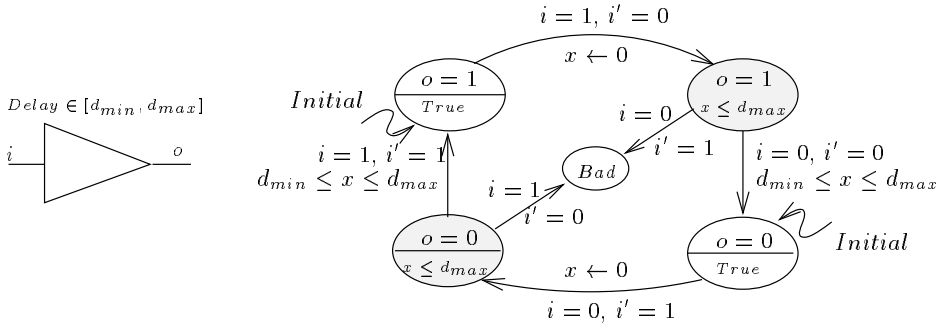

Figure 6.6: A delay buffer with delay in the range $[d_{min}, d_{max}]$. The locations where an output change is pending are shaded.

does occur, we will be aware of this fact, as the homomorphisms of Sections 6.4.3 and 6.4.2 will not map this location to any location in the abstract model. If the location "*Bad*" is reached, the homomorphism checks will not pass.

6.3.3 The Transmitter and the Receiver

The S/R descriptions of the transmitter and the receiver are presented in Section A.4. The transmitter has two locations: `WAIT` and `SEND`. When the clock input is low, the transmitter remains in the `WAIT` location, and moves to `SEND` when the clock rises. The *data_in* signal of Figure 6.1 is updated as the transmitter moves from `SEND` to `WAIT`, i.e., on the falling clock edge. This ensures that *data_in* has a fresh and stable value at each rising clock edge. As described in Section 6.2.3, the transmitter has to output an `E` as a spacer after each `T` and `F`. This is achieved by the use of the variables `out_data` and `old_out_data` in the S/R description of Section A.4. For convenience in stating the desired properties, the variable `acked` is used. This variable takes on the value 1 if the data currently being output by the receiver has been acknowledged by the FIFO; it is not part of the actual transmitter. The transmitter is allowed to modify `out_data` only when `acked` is 1. Later on, to prove the correctness property (i) of Section 6.2.1, we prove the equivalent statement that it is never the case that the transmitter wants to modify the data output and `acked` $= 0$, i.e., the FIFO is not ready to receive new data.

As far as the STARI interface is concerned, the only function of the receiver is to set the signal *ack_in* to the appropriate value after sampling the *data_out* signal at the falling edge of its clock input. This value is the `NOR` of the `.t` and `.f` signals representing *data_out*, as described in Section 6.2.4). The S/R description of Section A.4 performs

exactly this function. At each falling edge of the receiver's clock input, the `.t` and `.f` signals corresponding to *data_in* are sampled and stored in the variables `sampled_t` and `sampled_f` until the next falling edge of the clock input. A `NOR` gate with a delay in the range $[\delta_{rx,min}, \delta_{rx,max}]$ emulates the delay the receiver incurs while it processes the sampled data and computes *ack_in*.

6.3.4 Gate-Level Modeling of the FIFO

A timed automaton model for STARI at the gate level is obtained by composing the timed automata representing each C-element and `NOR` gate. We model the C-elements and the `NOR` gates used as ideal delayless elements followed by delay buffers, in a fashion similar to [MP95]. The S/R code describing an ideal C-element can be found in Section A.2. The timed process modeling an delay buffer is given in Figure 6.6. The S/R code for the FIFO stage describing it as the composition of gates is given in Section A.3.

6.4 Verification of STARI

The verification of STARI consisted of the following two steps:

(I) Constructing an abstraction for a FIFO stage and verifying its correctness within the environment that it operates.

(II) Verifying properties (i) and (ii) of Section 6.2 using the abstract model for the entire circuit.

The latter of these steps was performed using the existing timing verification capabilities of COSPAN [AK96]. The S/R specifications for properties (i) and (ii) are provided in Section A.9[1]. Verifying (I) is the novel part of our approach and is explained in detail in the rest of this section.

6.4.1 The Abstract Model for a Stage.

The abstract model for the FIFO stage describes its behavior at a high level, as in Section 6.2.3, and expresses bounds on certain response times. The S/R description for

[1]Note that the properties are stated in terms of the variables in the abstract model for the STARI interface. These variables will be defined in the rest of this section

the abstract model A is given in Section A.5. Figure 6.7 presents the essential features of the abstract model. A makes use of only one clock variable. Let us focus on stage k. At location `stable`, the FIFO stage has read and acknowledged its current input. If new data arrives at the inputs of stage k, A moves to location `wait_for_ack`, waiting for stage $k+1$ to acknowledge having copied the current data. If stage $k+1$ sends an acknowledgment before the new data arrives at stage k, stage k moves to location `wait_for_data` from location `stable` instead. After new data and an acknowledgment for the old data from stage $k+1$ has been received, A moves to location `out_pend` from where, after some delay, it moves to `ack_out_pend` and copies the new data to its output. Again after some delay, an acknowledgment is sent to stage $k-1$ and A moves to location `stable`.

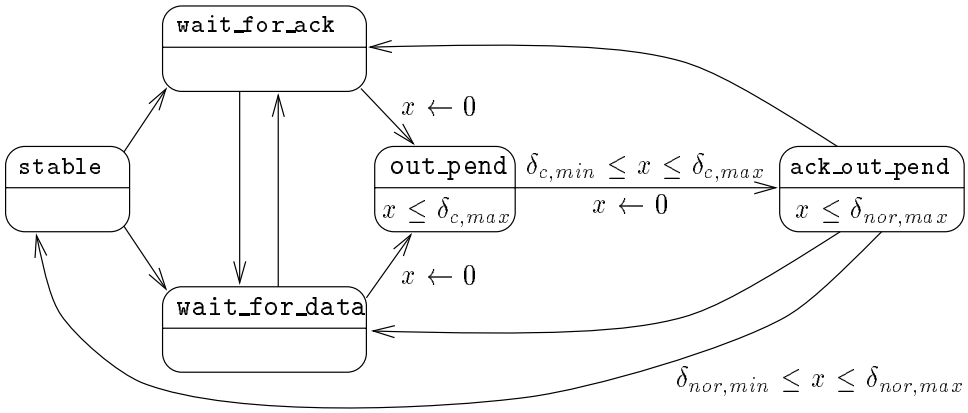

Figure 6.7: The timed process A describing the abstract model for a FIFO stage. The minimum and maximum response times of the C-element and the `NOR` gate are given by $\delta_{c,min}$, $\delta_{c,max}$ $\delta_{nor,min}$ and $\delta_{nor,max}$ respectively. Input predicates labeling the edges and output labels on the locations have been omitted to simplify the figure.

Note that it has been possible to capture the timing information about the stage using one clock only, since only one of the three circuit elements forming the stage can have a pending output change at any given time. Intuitively, this is guaranteed by the fact that the inputs to a stage will not change unless the stage has acknowledged the previous inputs. This assumption about the environment of a stage is crucial for the correctness of the abstraction, and is taken into account in our verification by the use of assume-guarantee reasoning.

6.4.2 Assume-Guarantee Reasoning for One Stage

Let F denote the timed process describing one stage of the FIFO at the gate level. F is the composition of processes representing two Muller C-elements and a NOR-gate as described by Figure 6.5 and Section 6.3.4. Let F_i and A_i denote the detailed and abstract models for the ith FIFO stage. F_i and A_i have structures identical to those of F and A. Also let Tx and Rx be the timed processes describing the transmitter and the receiver. We would like to prove that

$$Tx \parallel F_1 \parallel F_2 \parallel ... \parallel F_n \parallel Rx \preceq_L Tx \parallel A_1 \parallel A_2 \parallel ... \parallel A_n \parallel Rx \tag{6.1}$$

which will enable us to prove properties using the abstract description for the circuit given by the right-hand side. We would like to achieve this by showing that for all i, A_i is a correct abstraction for F_i. As noted above, this is true only within the environment that F_i and A_i operate in. For an arbitrary environment $F_i \leq_L A_i$ does not hold: if the inputs to F are unconstrained, then F_i does not behave like a FIFO element. Therefore, we need to employ assume-guarantee reasoning to carry out the proof. We must prove, for all i, $1 \leq i \leq n$, that

$$Tx \parallel A_1 \parallel ... \parallel A_{i-1} \parallel F_i \parallel A_{i+1} \parallel ... \parallel A_n \parallel Rx \preceq Tx \parallel A_1 \parallel ... \parallel A_{i-1} \parallel A_i \parallel A_{i+1} \parallel ... \parallel A_n \parallel Rx$$

Since the environment of each FIFO stage is different, for each i, the condition to be checked is different. To avoid performing a separate check for each i, we would like to have one set of environment assumptions that is sufficient for showing $F_i \preceq A_i$ and show that this set of assumptions is true for the environment of each module i. Towards this end, we define timed processes E_{left} and E_{right} and show the following:

(I) For all i, $Tx \parallel A_1 \parallel A_2 \parallel ... \parallel A_{i-1} \preceq E_{left}$

(II) For all i, $A_{i+1} \parallel A_{i+2} \parallel ... \parallel A_n \parallel Rx \preceq E_{right}$

(I) and (II) guarantee that E_{left} and E_{right} are correct abstractions for the left and right sides of the environment of stage i. By the assume-guarantee rule, showing that $F \preceq A$ in the environment defined by E_{left} and E_{right}, i.e.,

$$E_{left} \parallel F \parallel E_{right} \preceq E_{left} \parallel A \parallel E_{right} \tag{6.2}$$

suffices to prove the containment in Equation 6.1.

6.4.3 Proving the Correctness of Environment Abstractions.

E_{left} encapsulates the restrictions on the behavior of the left side of F for it to function correctly and for A to be a correct abstraction. The S/R description for E_{left} is provided in Section A.6. The essential features of E_{left} are that it always outputs a valid data value, and that it does not change this value until the next stage acknowledges having copied this value. Also, a time period of $\delta_{c,min}$ must elapse before the data output is modified.

We use induction on i to prove that E_{left} is a correct abstraction for $Tx \parallel A_1 \parallel A_2 \parallel ... \parallel A_{i-1}$. More precisely, we show the following

1. $Tx \preceq E_{left}$
2. Assuming $Tx \parallel A_1 \parallel A_2 \parallel ... \parallel A_{i-1} \preceq_L E_{left}$ we show $Tx \parallel A_1 \parallel ... \parallel A_i \preceq E_{left}$. By the induction assumption, it suffices to prove $E_{left} \parallel A_i \preceq E_{left}$.

For both steps we specify mappings from the locations of the left-hand side process to those of the right-hand side process, and show that this mapping preserves timed behavior using the COSPAN implementation described in Section 4.3. COSPAN code describing the mappings is provided in Section A.8.

The essential feature of E_{right} is that it samples the data at its inputs periodically and after a certain delay, acknowledges having read the data. The S/R description for E_{right} can be found in Section A.7. We proved that E_{right} is a correct abstraction in exactly the same manner as E_{left}. We showed that

1. $Rx \preceq E_{right}$
2. $A_i \parallel E_{right} \preceq E_{right}$.

in order to prove that E_{right} is a correct abstraction for $A_{i+1}... \parallel A_{n-1} \parallel A_n \parallel Rx$ The COSPAN code for the untimed mappings can be found in Section A.8.

Given E_{left} and E_{right}, it is straightforward to prove that $E_{left} \parallel F \parallel E_{right} \preceq E_{left} \parallel A \parallel E_{right}$ (Equation 6.2) as required in Section 6.4.2. COSPAN code for the untimed homomorphism witnessing this equation is given in Section A.8.4.

6.4.4 Time and Memory Consumption.

We report the resource usage for the following checks in Table 6.1:

(I) $Tx \preceq E_{left}$

(II) $E_{left} \parallel A \preceq E_{left}$

(III) $Rx \preceq E_{right}$

(IV) $A \parallel E_{right} \preceq E_{right}$

(V) $E_{left} \parallel F \parallel E_{right} \preceq E_{left} \parallel A \parallel E_{right}$

(VI) $Tx \parallel A_1 \parallel ... \parallel A_8 \parallel Rx$ satisfies timing properties (i) and (ii) of Section 6.2 [2].

For all of these checks the following parameters were used: $\delta_{c,min} = 1$, $\delta_{c,max} = 2$, $\delta_{nor,min} = 1$, $\delta_{nor,max} = 2$, and the clock period $\pi = 12$. The delays from the clock to the transmitter and the receiver (see Figure 6.1) were allowed to be time varying and bounded by 1 time unit, i.e., $\delta_{rx,min} = 0$, $\delta_{rx,max} = 1$, $\delta_{tx,min} = 0$, $\delta_{tx,max} = 1$.

	Num. of clocks	Memory (MB)	CPU time (seconds)
I	3	0.02	1.3
II	3	0.02	4.9
III	3	0.02	13.4
IV	6	3.18	92.7
V	7	6.33	158.6
VI	12	92.4	6014.1

Table 6.1: Run times and memory consumption for the verification of STARI

To serve as a comparison, we tried to verify properties (i) and (ii) of Section 6.2 using the gate level model for STARI, i.e., using F as the model for a FIFO stage. At this level of detail, we ran out of space using 1 GB of memory for a three stage FIFO. Improvements to our timing verification algorithm could enable us to verify larger FIFOs at the gate level, however, any method working on a monolithic description at the circuit level is bound to run out of resources for a FIFO with a large enough number of stages. Using abstractions

[2] See Section A.9 for the statement of these properties in S/R

as we demonstrated above, one is able to handle larger FIFOs, although not an unbounded number of them, since the number of FIFO elements still enters into the computation (VI).

6.5 Related Work

The most important benefit of our approach is the correctness guarantee that it provides for the actual circuit, whereas [LG95] and [HBAB93] based their proofs on oversimplified abstractions of the circuit. Their tools do not provide a mechanism for checking the correctness of abstractions, therefore there is no formal guarantee that the properties they proved at a high level are satisfied by the circuit.

[LG95] and [HBAB93] neglect the time taken by the `NOR` gate (see Figure 6.5) to compute the acknowledgment output and, furthermore, use a delay model that is less realistic than the inertial delay model that we employ. Since verification of properties on the abstract model is performed automatically in our approach, we did not need to resort to such simplifications.

The proof of [LG95] is for all FIFO lengths n, and makes it easier to see the trade-off between circuit parameters, whereas our approach is still limited by n. However, the proof of [LG95] is rather involved, and one needs to have an in-depth understanding of why the properties are satisfied. The abstraction proofs and environment abstractions that we used were rather straightforward and intuitive.

[BM98] is able to verify a 10 stage STARI circuit at the gate level using partial order reductions. The complexity relief offered by traversal methods based on partial order reductions are orthogonal to methods based on abstractions. These reductions can make abstraction checks, as well as the timing verification on abstract models more efficient. However, in general, partial order reductions alone are not sufficient to cope with the combinatorial dependency of the state-space size on the number of components in a system. Moreover, existing reduction methods can not handle certain aspects of asynchronous circuits, such as "true `OR` dependencies" [BM98].

6.6 Conclusions

We demonstrated the use of compositionality and hierarchy on the verification of the STARI communication circuit. By using the timed refinement checking algorithm that we

had implemented as part of COSPAN in a compositional framework, we were able to divide the verification problem into smaller pieces, which enabled us to automatically verify a larger circuit than was previously possible.

This case study demonstrated once more that abstractions are indispensable for verifying large systems, and that compositional and assume-guarantee style reasoning are not only useful techniques for verifying the correctness of abstractions, but are almost always necessary.

The size of our abstract model for STARI, $Tx \parallel A_1 \parallel ... \parallel A_n \parallel Rx$ still has an exponential dependency on n, although less severe than the gate level model. One problem that remains is the construction of an abstract model for STARI that is parametrized with respect to n. One would then prove the correctness of this abstraction using induction. Parametrized real-time systems have been studied before [RA93] and there is indication that verification with parametrized timed automata is rather complex: for more than two parameters, the reachability problem is undecidable.

Chapter 7

Circuit Delay Computation with Timed Automata

While verification of real-time behavior has been an intensely studied topic in recent years, common practice continues to be to design systems so that the logical functionality is correct regardless of timing. For high-speed combinational circuits, however, over-designing for all worst-case circuit effects is not feasible. As feature sizes of circuits shrink, electrical effects such as cross-talk and coupling become more pronounced, and new circuit design styles have made the timing and the logical behavior of a circuit very tightly related.

In this chapter, we present a timed automaton-based method for accurate computation of the delays of combinational circuits. In our method, circuits are represented as networks of timed automata, one per circuit element. The state space of the network represents the evolution of the circuit over time and delay is computed by performing a symbolic traversal of this state space.

Based on the topological structure of the circuit, a partitioning of the network and a corresponding conjunctively decomposed OBDD representation of the state space is derived. The delay computation algorithm operates on this decomposed representation and, on a class of circuits, obtains performance orders of magnitude better than a non-specialized traversal algorithm.

We demonstrate the use of timed automata for accurate modeling of gate delay and cross-talk. We introduce a gate delay model which accurately represents the timing characteristics as derived from a transistor-level model. We also construct a timed automaton

that models delay variations due to cross-talk for two capacitively coupled wires.

On a benchmark circuit, our algorithms achieve accuracy very close to that of a transistor level circuit simulator. We show that our algorithm is a powerful and accurate timing analyzer, with a cost significantly lower than transistor level circuit simulators, and an accuracy much higher than that of traditional timing analysis methods.

7.1 Background and Motivation

The computation of delay for combinational circuits is a well-studied problem which, until recently, was considered solved. Efficient exact methods were devised for computing the delays of acyclic combinational circuits ([DKM93], [LB94b], [MSBSV93], [YH95] among others). However, with the feature sizes of integrated circuits shrinking to sub-micron levels, some of the underlying assumptions of existing delay analysis methods are no longer valid. First, higher clock speeds require more accurate modeling of circuit delay, therefore, more sophisticated gate delay models are needed. For instance, for complex dynamic logic gates, gate delays depend on the relative timing of the input signals, as well as their values. Second, circuit level effects such as capacitive coupling between wires (also referred to as *cross-talk*) need to be taken into account. As a result, it is now commonly accepted that existing methods for modeling and computing delays are inadequate for deep sub-micron circuits (See, for instance, [CWS97, TKB97] and other related papers in the TAU '97 Workshop). Existing timing analysis schemes do not use sufficiently accurate gate delay models, and do not account for cross-talk.

Timed automata have been used to model the delay characteristics of gates and circuits ([FK94], [FK95], [MP95], [TAKB96], [TB97]). Previously, timed automaton-based techniques have been restricted to the analysis of asynchronous circuits because they are generally smaller than synchronous circuits and require a more detailed temporal analysis than delay computation. Today's synchronous circuits with sub-micron feature sizes place stronger demands on timing analysis tools, and this new setting makes the expressiveness of timed automata desirable. Timed automata allow the use of very general gate delay models, including those in which a unique delay can be specified for every pair of input vectors. As a result, effects of cross-talk on delay can be incorporated. One contribution reported in this chapter is the introduction of a new and accurate gate delay model, which efficiently accounts for input sequence dependent delays.

For large portions of a typical design, a less powerful modeling framework suffices. For instance, the topological delay of the circuit has traditionally been used as a conservative but often rough estimate of the true delay of the circuit. However, simple topological delay is not conservative for circuits where electrical coupling is a significant factor. A "worst-case topological analysis" assuming worst-case cross-talk on all wires at all times is likely to be too conservative and not suitable for high-performance circuits . The analysis of certain critical portions of such a design must be performed with great accuracy, which is currently provided only by transistor level simulators. Exhaustive simulation of these sections of the circuit, on the other hand, is not computationally feasible because of the exponential number of possible input patterns. In this study, we present a timed automaton-based delay computation method. We represent input waveforms and circuits by networks of timed automata, with each automaton modeling a circuit element or input waveform. The state space of this network describes the possible behaviors of the circuit over time. Delay computation is then posed as a variant of the state-space traversal problem. Because all input patterns are covered, unlike simulation, the circuit delay computed by this algorithm is guaranteed to be correct.

State-space traversal of timed automata is a PSPACE-complete problem, and, like other traversal-based methods, it suffers from the state-space explosion problem for systems with large numbers of components. However, by exploiting certain topological properties of combinational circuits and the state spaces of the corresponding timed automata, we devise heuristics which enable us to handle systems that are much larger than could be handled with non-specialized traversal methods (cf. Section 7.4.2).

This chapter is organized as follows. Section 7.2 discusses how circuits and waveforms are represented using timed automata. Section 7.3 describes our algorithm for computing maximum delays. Section 7.4 presents experimental results and contrasts our method with other approaches for delay computation. Section 7.6 presents the conclusions from our experience with the methods developed in this chapter, and discusses future research directions.

7.2 Modeling Circuits and Waveforms

7.2.1 Modeling Sets of Waveforms

The maximum delays of combinational circuits are typically computed for the following two cases:

- *Floating-mode delay*: The primary inputs are allowed to change their values arbitrarily until time t_0. After time t_0 all inputs remain stable.
- *Two vector delay*: The primary inputs (and all intermediate nodes of the circuit) are stable until time t_0. At time t_0 the inputs switch to new values and must remain stable thereafter.

The sets of input waveforms for both cases can be represented concisely by timed automata. Figure 7.1 shows the timed automaton for the two-vector delay model. $\vec{\imath}$ is the vector of primary input variables, which are the outputs of this automaton. $\vec{\imath}_{old}$ and $\vec{\imath}_{new}$ represent the primary inputs before and after time t_0 ($t = t_0$) respectively. $\vec{\imath}_{old}$ and $\vec{\imath}_{new}$ are selected non-deterministically, which enables the automaton in the figure to represent all input vector pairs.

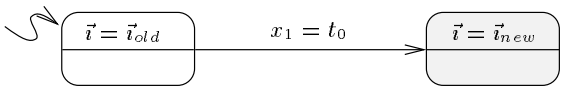

Figure 7.1: The timed automaton producing the primary inputs for the two-vector delay model. All inputs arrive synchronously at time t_0.

It is also straightforward to model different arrival times at different primary inputs, asynchronous inputs, etc., with timed automata using extra clocks. Some examples are shown in Figure 7.2.

7.2.2 Modeling Combinational Gates

Timed automata offer a great deal of expressive power for representing gate delay models. In the past, they have been used to model gates ([MP95],[TAKB96], [TB97], [FK94], [FK95]) in the verification of asynchronous circuits and library elements. For instance, in [TB97] and Chapter 2, timed automaton representations for the inertial delay model were described.

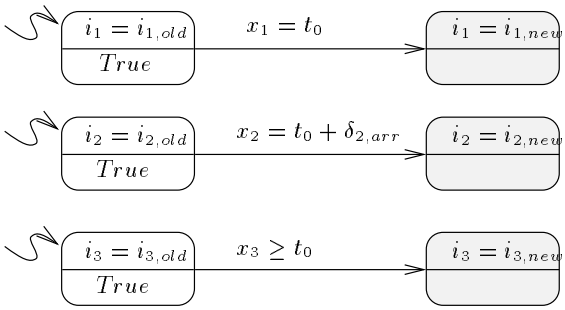

Figure 7.2: Automata generating various primary inputs. i_1 arrives synchronously with the clock at time t_0. i_2 has an arrival time of $\delta_{2,arr}$. i_3 is an asynchronous input that can arrive anytime after the clock signal.

The gate delay models employed by conventional timing analysis methods, on the other hand, have been somewhat rudimentary. In the following, we briefly review them in increasing order of sophistication with which they model the dependency of delay on different factors. The most simplistic one is the *unit delay* model, where each gate is assumed to have a delay of one time unit. The *fixed* delay model, allows gates to have different but constant delays. Both of these models are too coarse as they do not allow for variations in gate delay. The *min-max* delay model addresses this problem to some degree by allowing variations in the delay of a gate: transitions at a gate's input are reflected at its output with a delay in the range $[d_{min}, d_{max}]$. In the *pin-delay* model, different input-to-output delay ranges can be specified for different input pins of a gate. Neither the min-max model nor the pin-delay model are powerful enough for an accurate analysis because they can not express the dependency of gate delay on specific input values.

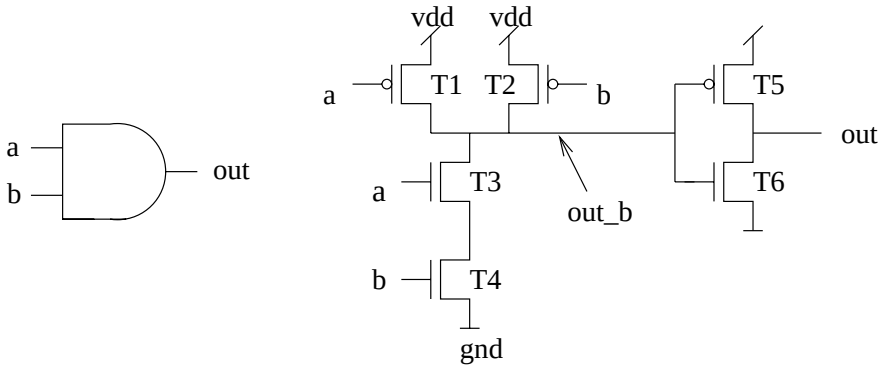

Figure 7.3: `AND` gate: Transistor Level Description

In practice, gate delays depend strongly on the input vectors applied. This is illustrated by means of the static CMOS AND gate shown in Figure 7.3. For this gate, let us denote the delay under the input sequence $a = b = 1 \rightarrow a = 1, b = 0$ by $t^f_{11\rightarrow 10}$. The superscript f (alternatively, r) indicates that the output is falling (alternatively, rising), and the subscript represents the applied input vector sequence. Similar notation is used for the input sequence: for the sequence $a = b = 1 \rightarrow a = 0, b = 0$, the gate delay is $t^f_{11\rightarrow 00}$. In the second case, the delay of the gate is smaller. This is because, in the latter case, after the inputs transition, both transistors T1 and T2 are on, effectively doubling the current to charge the capacitance of node out_b. This results in a faster falling transition. In the first case, only transistor T2 is on after input b has switched, hence the falling transition is slower. The pin and min-max delay models do not distinguish between these two input sequences. High performance combinational circuits make extensive use of complex gates with many inputs, and for these gates, the effect of input vectors on delay is more pronounced. For such circuits the pin and min-max delay models are not accurate enough. Some other conventional delay models [MSBSV93, YH95, Yal97] allow the specification of separate rising and falling delays for the gate output. This is a useful property to model, but in the absence of input sequence dependence of the output delay, this capability alone is not very useful. *State-dependent delay* models (see [Yal97] for a classification of delay models) enable different delays to be specified for different *final* values of the gate inputs, but do not model the effect of the initial input values on gate delay.

Timed automata provide a powerful and uniform framework in which gate delay models of varying sophistication can be specified. One has the flexibility to assign a distinct delay for every possible sequence of input vectors (for an n input gate, there are $2^n \cdot (2^n - 1)$ of these). However, in practice, there are only a few sequences of input vectors that result in distinct delays. Many sequences of input vector transitions result in approximately the same gate delay, and we make use of this property to simplify the timed automata for gates. For example, in Figure 7.3, the delays $t^f_{11\rightarrow 01}$ and $t^f_{11\rightarrow 10}$ have the same value, and so do the delays $t^r_{01\rightarrow 11}$, $t^r_{10\rightarrow 11}$, and $t^r_{00\rightarrow 11}$. We group together these delays in our timed automaton model for the AND gate, resulting in a simpler delay model. In this way, our model incorporates input sequence dependent delays without becoming prohibitively large. A delay model in which different vector pairs are assigned distinct delays was previously introduced by [FK94, FK95]. However, this model requires a significantly larger number of transitions to model delays. For a 3-input NAND gate, the authors of [FK94] state that

109 transitions would be required in their gate model. In our model, only four distinct transitions are required.

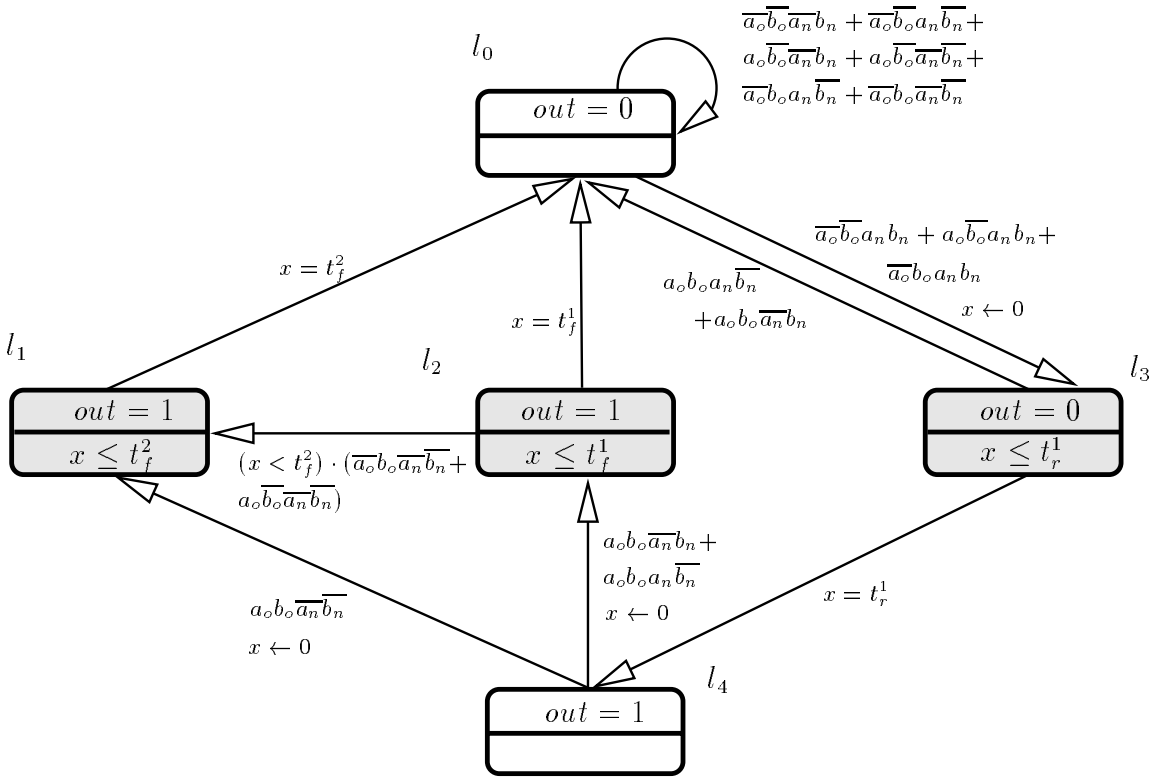

Figure 7.4: Timed automaton for the AND gate.

The timed automaton representing an AND gate is shown in Figure 7.4. The initial location is l_0 or l_4 depending on the initial values of the inputs to the gate. The inputs to the AND gate are a and b. a_o and a_n represent the old and new values of the input a respectively. Similarly for input b. We use the following shorthand: if the input $a = b = 1$ is followed by $a = 1, b = 0$, then this condition is represented by $a_o b_o a_n\overline{b_n}$. "Transient" locations are shaded. These are locations representing a pending output change where the automaton can remain only for a limited time. At any location, if there is an input change that does not require the output of the automaton to change, the automaton remains in the same location. The location l_0 has such a self-loop. Some self-loops are not depicted in order to keep the figure readable. Similarly, if inputs change and then return to their original value while the automaton is still in a transient location (referred to as a *glitch*), the automaton returns to the location that led to the transient location. If, at location

l_0, the condition $\overline{a_o} b_o a_n b_n$ occurs, then clock x is set to 0, and a transition is made to the temporary location l_3. After a delay of t_r^1, a final transition is made to a location where $out = 1$. This corresponds to the gate output changing from 0 to 1 after a delay of t_r^1, when inputs change from $a = 0, b = 1$ to $a = 1, b = 1$. However, if $a = 1, b = 1$ is followed by $a = 1, b = 0$, i.e., a glitch occurs while $x < t_r^1$, the automaton returns to the location with $out = 0$. Again, some edges corresponding to glitches are not drawn in the figure for ease of readability.

When inputs change from $a = 1, b = 1$, depending on whether one or both of the inputs change to 0, the gate can have two distinct delays. If only one of the inputs changes to a 0, then the delay of the gate is t_f^1. If both inputs change to a 0, then the delay of the gate is t_f^2. As discussed earlier in this section, $t_f^1 > t_f^2$, because t_f^2 corresponds to the case where the current charging node out_b of Figure 7.3 is doubled. In either case, the timed automaton makes a transition to the transient location l_2 after setting clock x to 0. From this temporary location, it makes the final transition to l_0 after the appropriate delay. In case the two inputs do not change simultaneously change to 0, the timed automaton makes a transition to l_2, from which it makes a transition to l_1 if the second input arrives before t_f^2.

Timed automata for other gates are constructed similarly. For the transistor level representation of a gate, suppose there are n paths from the output node (In Figure 7.3, out_b is the output node) to the power supply, and m paths from the output node to the ground. Assuming that all paths to the supply and ground have the same effective size of transistors, the timed automaton for the gate will have n distinct rising delays, and m distinct falling delays. In general, for a k input gate, each of the $2^k \cdot (2^k - 1)$ input sequences give rise to distinct delays at the output. However, in practice, the delays are tightly clustered around the $n + m$ distinct values we use.

The process of obtaining a timed automaton delay model for a given CMOS gate proceeds as follows. We first determine the values of m and n. After this, we compute each distinct rise and fall delay using the transistor-level simulator SPICE [Nag95]. All SPICE simulations reported in this chapter use 0.1 μm transistor models. Interconnections are assumed to be made of copper, and are assumed to correspond to a 0.1 μm fabrication process. For complicated gates where it may be difficult to determine the distinct rise and fall times by inspection, the gate can be SPICE simulated for all possible input patterns. This procedure would be impractical for a large circuit, but for gates and other library

elements, it is feasible and, in fact, common practice.

7.2.3 Modeling Cross-Talk between Wires

As the minimum feature sizes of VLSI fabrication processes decrease, certain electrical effects become more pronounced. Figure 7.5 shows a graphical view of two wires on an integrated circuit. In most VLSI processes, $W = S$. W (and S) decrease linearly with minimum feature size, but T, the distance between wires on different metal layers decreases sub-linearly. As a result, the ratio of the capacitance between a wire and its neighboring wire to the capacitance between a wire and wires on other metal layers increases with diminishing feature sizes [NTR97].

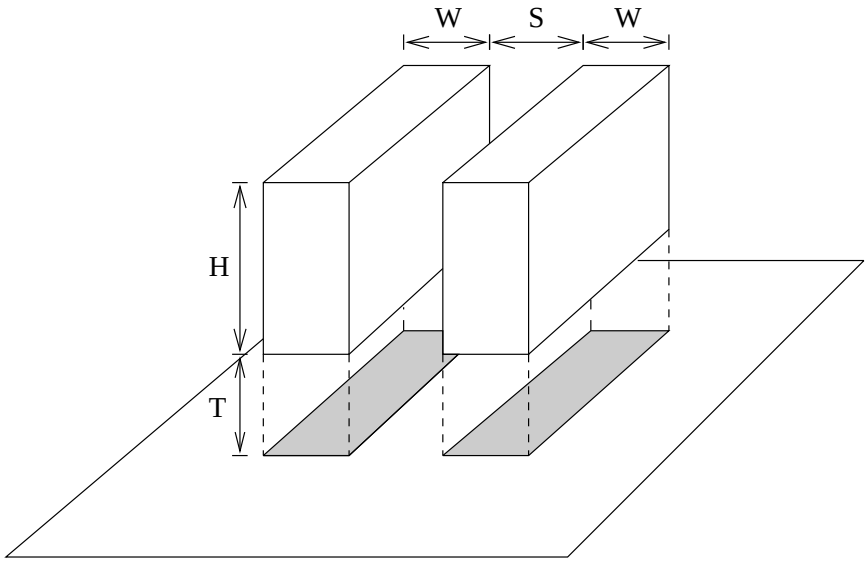

Figure 7.5: Two wires on a VLSI circuit

One of the effects of increased capacitance to neighboring wires is a large variation in the delay of the wire. If the neighboring wires switch in the same direction as the wire of interest, the delay of the wire is decreased, and if they switch in the opposite direction, its delay increases. The effect of the transition activity of a wire on its neighboring wires is referred to as *cross-talk*. A SPICE simulation of three neighboring wires of length 150 μm in a 0.1 μm process shows a 2:1 variation in the delay of the center wire due to cross-talk. In the set-up for this experiment, the wires were driven by static CMOS inverters, whose P devices were 4.5 times larger than a minimum P device, and whose N devices were 3 times larger than a

minimum N device. The parasitics for this configuration of wires were determined using the three-dimensional parasitic extractor SPACE [Pro]. VLSI processes with a 0.1 μm feature size are still a few years away from production, but the above simulation indicates that delay variation due to cross-talk will be a major problem in the future. For this reason, it is becoming increasingly important for circuit timing analyzers to incorporate the effect of cross-talk.

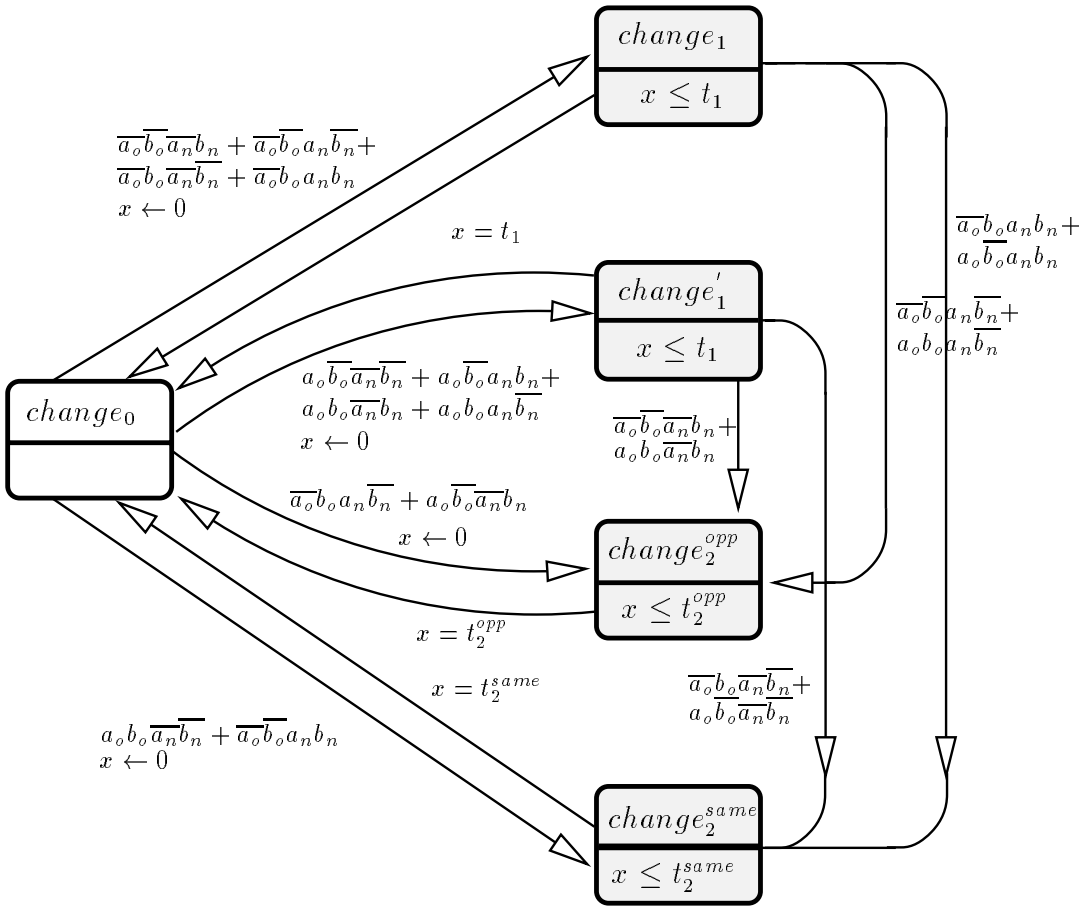

Figure 7.6: Timed automaton model representing the delay of a pair of wires

We model cross-talk between two wires by the timed automaton shown in Figure 7.6. In this figure, the location $change_0$ represents the condition when both wires have a stable value. From this location, if one of the wires switches, a transition is made to one of the temporary locations $change_1$ or $change_1^{'}$. While in these temporary locations, if the other wire switches, a transition is made to a location which models both wires switching. There

are two such locations, $change_2^{same}$, which models the two wires switching in the same direction, and $change_2^{opp}$ which models the two wires switching in opposite directions. In a manner similar to the AND gate model, the automaton returns to $change_0$ whenever a glitch occurs. These edges have not been shown in order to keep the figure simple. The delay associated with $change_1$ and $change_1^{'}$ is t_1. The delays associated with $change_2^{same}$ and $change_2^{opp}$ are t_2^{same} and t_2^{opp} respectively. These delays are determined by running SPICE on the circuit model derived from the layout using SPACE. Clearly, $t_2^{same} \leq t_1 \leq t_2^{opp}$.

7.2.4 Additional Constructs For Circuit Modeling

In this section we introduce two new additions to timed automata: initial state relations and reset functions. These constructs make it more convenient to model combinational circuit, and, as will be shown, do not affect the complexity of, or require significant modifications to verification and delay computation algorithms.

Initial State Relations

When modeling circuit components with timed automata, it is useful to be able to relate the initial location to the initial values of the inputs. For instance, the buffer in Figure 2.1 will start at location l_0 if its input is initially 1, and at location l_3 if its input is initially 0, assuming that it starts at a stable state. To provide this capability, in this chapter we allow the initial location of a timed automaton A to be specified as a relation

$$Init^A \subseteq \{\mathbf{f} |\ \mathbf{f} \text{ is an } I^A \text{ valuation } \} \times S^A$$

$\langle \mathbf{f}, s_0 \rangle \in Init^A$ (equivalently $Init^A(\mathbf{f}, s_0) = True$) if s_0 is one of the possible initial locations when the initial inputs are given by the I^A valuation $\mathbf{f}$. $Init^A$ is called the *initial state relation* of A and is required to be *total*, i.e.,

$$\forall \mathbf{f}\ \exists s_0\ Init^A(\mathbf{f}, s_0)$$

The semantics of timed automata remain unchanged with the understanding that a location is considered initial if and only if it is in *Init* for some initial input $\mathbf{f}$.

A minor change needs to be made to the definition of composition for timed automata. For timed automata A and B with initial state relations $Init^A$ and $Init^B$, the initial state

relation of $A \parallel B$ is given by

$$Init_0^{A\|B} = \Big\{ \langle \mathbf{f}, \langle s_0^A, s_0^B \rangle \rangle \Big| \quad \langle (\mathbf{f}(I^A) \cup \mu^B(s_0^B)), s_0^A \rangle \in Init^A, \text{ and } \langle (\mathbf{f}(I^B) \cup \mu^A(s_0^A)), s_0^B \rangle \in Init^B \Big\}$$

and A and B are said to be composable iff O^A and O^B are disjoint and $Init^{A\|B}$ as defined above is total.

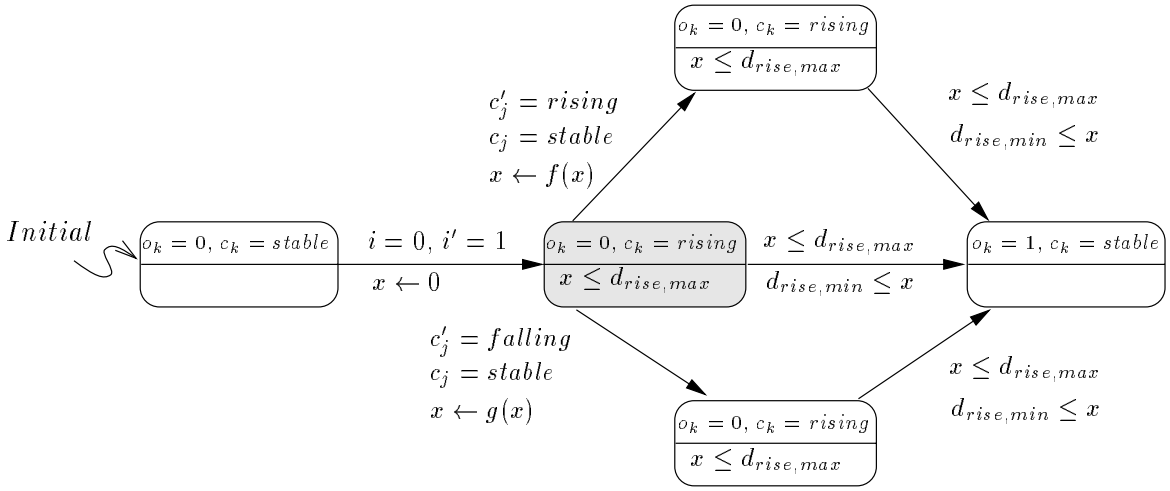

Figure 7.7: Delay model for wire k incorporating cross-talk from wire j. i is the signal that drives wire k.

Reset Functions

For a more precise modeling of cross-talk, we augment timed automata with reset functions in the way depicted in Figure 7.7. This enables us to incorporate simulation results into the delay model without having to devise new analysis algorithms. In this version of timed automata, which will be used in the rest of this chapter, edges e are tuples of the form $\langle s, t, \varphi, \chi, R \rangle$ where a reset function R specifies for each clock x its value after the edge is taken, $R(x)$:

(I) $R(x) = 0$: x is reset to 0.

(II) $R(x) = y$, where $y \in X$: x takes on the value of y right before the transition. The capability to assign the value of one timer to another enables the reuse of timer variables and facilitates minimization of timer variables [DY96]. If $y = x$, x is left unchanged.

(III) $R(x) = f(x)$, where, for each $x \in \mathbb{R}^+ \cup \{0\}$, $f(x)$ is a closed interval of the form $[m_{min}, m_{max}]$ with $m_{min}, m_{max} \in \mathbb{N}$. x can be assigned any value in the interval $[m_{min}, m_{max}]$ after the edge is taken. f is referred to as a "reset function" and has to have the following properties:

(i) For any $n \in \mathbb{N}$ and $x, y \in \mathbb{R}$ such that $x, y \in (n, n+1)$, $f(x) = f(y)$.

(ii) For any $n \in \mathbb{N}$, $f(n) \subseteq f(n-\varepsilon) \cup f(n+\varepsilon)$ for $0 < \varepsilon < 1$.

(iii) There exists some $k_f \in \mathbb{N}$ such that, for all $x \geq k_f$, $f(x) = f(k_f)$.

In words, the value of $f(x)$ depends on the integer part of x for values less than k_f, and whether the fractional part of x is 0. Because of properties (i)-(iii), f can be represented as a look-up table. The purpose of this type of reset is to enable the specification of a new value of x as a function of its current value without affecting the correctness of the constructions presented in Chapter 2. The reasons for this particular set of requirements on f will become clear below.

Region equivalence for this variant of timed automata is defined in the same way as in Section 2.6.2 with the following modification. For each clock x, K_x denotes the maximum among (i) constants that x gets compared within the X-predicates on the edges of A and invariants on the locations of A, (ii) constants k_f such that the reset $x \leftarrow f(x)$ occurs on some edge of A.

Lemma 7.2.1 helps us prove the counterpart of Theorem 2.6.4 on this variant of timed automata.

Lemma 7.2.1 *Let $\langle s, \Phi \rangle \equiv \langle s, \Theta \rangle$. If, for a timed event $\gamma = \langle \delta, \mathbf{f}, \mathbf{f}' \rangle$, $\langle s, \Phi \rangle \xrightarrow{\gamma} \langle s', \Phi' \rangle$, then there exist $\gamma' = \langle \delta', \mathbf{f}, \mathbf{f}' \rangle$ and $\langle s', \Theta' \rangle$ such that $\langle s, \Theta \rangle \xrightarrow{\gamma'} \langle s', \Theta' \rangle$ and $\langle s', \Phi' \rangle \equiv \langle s', \Theta' \rangle$.*

Proof: Let $e = \langle s, s', \varphi, \chi, R \rangle$ be the edge of the automaton associated with the transition $\langle s, \Phi \rangle \xrightarrow{\gamma} \langle s', \Phi' \rangle$. From the region automaton construction, it is known that there exists δ' such that $wait(\langle s, \Theta \rangle, \delta')$ and $\langle s, \Theta + \delta' \rangle \equiv \langle s, \Phi + \delta \rangle$. $\Theta + \delta' \models \varphi$, therefore, the edge e can be taken on the timed event $\gamma' = \langle \delta', \mathbf{f}, \mathbf{f}' \rangle$. We argue that the resets in R can be applied to $\Theta + \delta'$ in such a way that the final result is region equivalent to Φ'. Type (I) and (II) resets pose no difficulty: given two equivalent X-valuations, applying resets of these types leaves them in the same equivalence class. Suppose that on e, a type (III) reset is applied to clock x, which assigns to it non-deterministically a value in the range

$[n_{min}, n_{max}]$. By the definition of region equivalence, we have $\lfloor(\Phi+\delta)(x)\rfloor = \lfloor(\Theta+\delta')(x)\rfloor$, and $Fr((\Phi+\delta)(x)) > 0$ if and only if $Fr((\Theta+\delta')(x)) > 0$. Therefore, for the γ transition as well as the γ' transition, the new value of x is selected from $[n_{min}, n_{max}]$. $\Theta'(x)$ must be picked to have the same integer part as $\Phi'(x)$, and its fractional part must be picked such that the ordering of timers according to their fractional parts is the same for Φ' and Θ'. This is possible, since we have complete freedom in choosing the fractional part of $\Theta'(x)$. With this, it is guaranteed that $\langle s', \Phi'\rangle \equiv \langle s', \Theta'\rangle$. ■

Theorem 7.2.2 *Suppose that starting from $\langle s_0, \Phi_0\rangle$, the following sequence of transitions are possible in a timed automaton: $\langle s_0, \Phi_0\rangle \stackrel{\gamma_0}{\rightarrow} \langle s_1, \Phi_1\rangle \stackrel{\gamma_1}{\rightarrow} \ldots \stackrel{\gamma_{k-1}}{\rightarrow} \langle s_k, \Phi_k\rangle$ Let Θ_0 be such that $\langle s_0, \Phi_0\rangle \equiv \langle s_0, \Theta_0\rangle$. Then there exists a sequence of timed events $\gamma'_0, \ldots, \gamma'_{k-1}$ such that $\langle s_0, \Theta_0\rangle \stackrel{\gamma'_0}{\rightarrow} \langle s_1, \Theta_1\rangle \stackrel{\gamma'_1}{\rightarrow} \ldots \stackrel{\gamma'_{k-1}}{\rightarrow} \langle s_k, \Theta_k\rangle$ and for each i, γ'_i differs from γ_i only in the time increment, and not in the assignments to the input and output variables.*

Proof: Follows by induction on k from Lemma 7.2.1. ■

Theorem 7.2.2 provides the formal basis for region-automaton-based delay computation algorithms operating on augmented timed automata.

Having developed the required formal machinery, let us now examine Figure 7.7 in more closely. Let A_k denote the timed automaton representing wire k. A_k has two outputs o_k and c_k. o_k takes on values from the set $\{0, 1\}$ and represents the digital value of the signal at the output end of wire k, while c_k takes on values from the set $\{rising, falling, stable\}$ and indicates whether wire k is making a transition or has a stable signal on it. If the signal on wire j affects wire k, c_j is an input to the automaton representing wire k. Suppose that i, the signal that drives wire k makes a transition from 0 to 1 and in response to this, A_k moves to the shaded location where, if x has value $\nu(x)$, within $[d_{rise,min} - \nu(x), d_{rise,max} - \nu(x)]$ time units, a transition is taken and the output o_k becomes 1.

A typical scenario is where another wire j runs parallel to wire k and because of this, if j starts rising while k is rising, this reduces the time k takes to rise. Suppose while A_k is in the shaded location c_j changes from *stable* to *rising*. The coupling between the wires is modeled by the reset function f on the marked edge. f will increase the value of x to reflect the fact that now o_k will take less time to rise. x can take on any value from the interval $f(\nu(x)) = [n_{min}, n_{max}]$. Let $\tilde{x}$ be the new value of x. Now a transition to the location where $o_k = 1$ is possible when $d_{rise,min} \leq \tilde{x} \leq d_{rise,max}$, i.e., within $[d_{rise,min} - \tilde{x}, d_{rise,max} - \tilde{x}]$ time units. Thus, f should be such that if the signal on wire k

has been rising for $(\lfloor \nu(x) \rfloor, \lfloor \nu(x) \rfloor + 1)$ time units, the remaining time to rise for o_k with positive coupling from j falls inside the interval $[d_{rise,min} - f(\nu(x)), d_{rise,max} - f(\nu(x))]$. Such $d_{rise,min}$, $d_{rise,max}$ and a look-up table f can either be determined analytically or computed by simulating the two wires as described earlier in this section. By choosing the range $[n_{min}, n_{max}]$ for $\tilde{x}$ large enough, we can ensure conservative modeling of variations in the rise time of x due to cross-talk. Choosing the time unit to be smaller will yield a more accurate model but to larger verification complexity. It is possible to refine the delay model by incorporating more cases for possible interactions between wires. The purpose of the example in Figure 7.5 is to demonstrate that timed automata have the expressiveness to capture cross-talk effects and represent them in a way that captures all behavior obtained from simulation.

In the next section, we describe our delay computation algorithm. The algorithm operates on a network of timed automata representing the primary input waveforms and circuit elements.

7.3 Delay Computation with Timed Automata

We pose the delay computation problem as follows: Given a combinational circuit, described as an interconnection of circuit components, and a set of primary input waveforms, we want to determine the latest time that primary outputs become stable. The set of input waveforms is represented by a timed automaton I as described in Section 7.2.1. The circuit is described as an interconnection of timed automata, $G_1, ..., G_n$. The delay parameters of these are determined by SPICE simulation of the transistor-level circuit description, as described in the example in Figure 7.2.2. The delay computation problem is then formally stated in the following way: Let

$$F = (I \parallel G_1 \parallel G_2 \parallel ... \parallel G_n)$$

represent the evolution of the circuit over time for the primary input waveforms described by I. Let us denote by $G_{o_1}, ..., G_{o_m}$ the circuit components whose outputs are primary outputs of the circuit. For each j, define E_{o_j} to be the set of edges of G_{o_j} for which the primary outputs at the source and target locations are different. Formally,

$$E_{o_j} = \left\{ \langle s_{o_j}, t_{o_j}, \varphi, \chi, R \rangle \,\middle|\, (\mu(s_{o_j}))(O_{prim}) \neq (\mu(t_{o_j}))(O_{prim}) \right\}$$

where O_{prim} denotes the set of primary output variables of the circuit. The delay of the circuit is the latest time that an edge in some E_{o_j} can be traversed in F. Denote by E_{switch} the set of edges of F whose projection onto some G_{o_j} lies in E_{o_j}, and let $S_{switch} \subseteq \Sigma_F$ be the set of states of F that have an outgoing E_{switch} transition. The goal is then to compute the largest time elapsed on paths from initial states of F to S_{switch}. The most straightforward way to obtain this information is to explore the state-space of $Region(F)$. However, the size of this state-space is often very large. Automata networks describing circuits have certain special properties, which enable significant improvements to the traversal techniques. The rest of this section elaborates on these properties and how they are used for making state-space exploration more efficient.

7.3.1 A Region Automaton with Integer Delays

Traversal methods based on difference bound matrices (see Section 2.6.4) are not suitable for delay computation purposes because they represent locations explicitly, and the Boolean component, and therefore the number of locations, of the automaton F is often considerably large. We will employ a region-automaton-based approach instead, as we have done in the previous chapters.

Region equivalence for timed automata as defined in Section 2.6.2 has to distinguish between clock valuations which have different orderings of the fractional parts of clock values. This contributes a factor of $k!$ to the state-space, where k is the total number of timers used in F [AD94]. For a subclass of timed automata, this component of the state space can be eliminated by using Theorem 7.3.1, an equivalent statement of which was proven in [HMP92]. Roughly stated, this theorem stipulates that given any run of a timed automaton, it is possible to construct a run whose transitions occur at integer-valued times, and which traverses the same sequence of locations as the original one.

Theorem 7.3.1 (Henzinger, Manna, Pnueli '92) *Let A be a timed automaton as defined in Section 2.3*[1] *and let $\sigma_0, \sigma_1, \sigma_2, ..., \sigma_k$ be a run of A on the timed event sequence $\overline{\gamma} = \gamma_0, \gamma_1, ..., \gamma_{k-1}$, where $\gamma_i = \langle \delta_i, \mathbf{f}_i, \mathbf{f}'_i \rangle$ and $\sigma_i = \langle s_i, \Phi_i \rangle$ for all i. With $t_i =_{def} \sum_{j=0}^{k} \delta_i$*

[1] Recall that for this type of timed automata, clock predicates must be positive Boolean combination of non-strict inequalities that compare clocks with integers.

let us denote the time that the transition on γ_i *is taken. For arbitrary* $0 < \varepsilon < 1$ *define*

$$\tau_i = \begin{cases} \lfloor t_i \rfloor, \ \textit{if } t_i - \lfloor t_i \rfloor \leq \varepsilon \\ \lceil t_i \rceil, \ \textit{if } t_i - \lfloor t_i \rfloor > \varepsilon \end{cases}$$

and $\theta_i = \langle \Delta_i, \mathbf{f}_i, \mathbf{f}_i' \rangle$ *where*

$$\Delta_i = \begin{cases} \tau_0, \ \textit{for } i = 0 \\ \tau_i - \tau_{i-1}, \ \textit{for } i > 0 \end{cases}$$

Then there exists a run $\rho_0, \rho_1, \rho_2, ..., \rho_k$ *of* A *on the timed event sequence* $\overline{\theta} = \theta_0, \theta_1, ..., \theta_{k-1}$ *where* $\rho_i = \langle s_i, \Theta_i \rangle$ *for all* i.

Note that the clock valuations Θ_i of the run $\rho_0, \rho_1, ...\rho_k$ must all be integer-valued. It follows from Theorem 7.3.1 that there must exist a run of this form which exhibits the largest delay of the circuit.

While performing delay computation, timer predicates refer to the time elapsed between two transitions in node voltages, which are analog waveforms over real-valued time. It does not make any physical sense to specify that the time elapsed between two analog transitions can be any value less than (or greater than) but not equal to $c \in \mathbb{N}$ time units. The set of possible elapsed times is simply rounded to the nearest closed interval with integer endpoints. Therefore, combinational circuits can be modeled without using strict inequalities. This enables the clocks to be treated as integer-valued variables which increase at the same rate. Observe that considering runs with integer transition times may result in $\Delta_i = 0$ in some cases, i.e., a state or a transition may precede another although there seems to be no separation in time between them. We have disallowed events with zero time increments up to now, as they make it difficult to define languages and compositional proof rules. Only in the context of delay computation will we allow such events, in order to be able to use Theorem 7.3.1. The use of zero-time events should be regarded as an internal construct of delay computation algorithms presented in this chapter, and not as part of the formal semantics of timed automata.

With this interpretation, a run of a timed automaton A of the form $\sigma_0 \stackrel{\gamma_0}{\rightarrow} \sigma_1 \stackrel{\gamma_1}{\rightarrow} \sigma_2 \stackrel{\gamma_2}{\rightarrow} ... \stackrel{\gamma_{k-1}}{\rightarrow} \sigma_k$ can be viewed as an interleaving of *time passage transitions* and *control transitions* as follows: $\sigma_0 \stackrel{\delta_0}{\rightarrow} (\sigma_0 + \delta_0) \stackrel{\langle 0, \mathbf{f}_0, \mathbf{f}_0' \rangle}{\rightarrow} \sigma_1 \stackrel{\delta_1}{\rightarrow} (\sigma_1 + \delta_1) \stackrel{\langle 0, \mathbf{f}_1, \mathbf{f}_1' \rangle}{\rightarrow} ... \stackrel{\langle 0, \mathbf{f}_{k-1}, \mathbf{f}_{k-1}' \rangle}{\rightarrow} \sigma_k$, where $\sigma_i + \delta_i$ is shorthand for all clocks in σ_i being incremented by $\delta_i \in \mathbb{N}$ and $\stackrel{\delta}{\rightarrow}$ denotes the stuttering timed event with a time increment of $\delta \in \mathbb{N}$. Note that time passage transitions

with $\delta_i \geq 1$ can be a realized by a sequence of $\delta = 1$ transitions. We have found that taking time steps of one at a time results in more efficient ordered binary decision diagram (OBDD)-based analysis algorithms and makes the formulation of the delay problem easier (see Section 7.3.2). Clearly, the reachability and delay properties of the automaton remain the same.

Using these observations, given a timed automaton A, we construct a variant of the region automaton, denoted by $[A]$, to be used for delay computation purposes. The locations of the finite automaton $[A]$ have the form $\sigma = \langle s, \Phi \rangle$, where s is a location of A and Φ is an integer-valued clock-valuation, where the value of each clock x is limited to the interval $[0, K_x]$. The output function $\mu^{[A]}$ is defined as $\mu^{[A]}(\langle s, \Phi \rangle) = \mu^A(s)$. The edges of $[A]$ are grouped into two. The *time passage edges* $E_\delta^{[A]}$ correspond to stuttering events in A where all variables except clocks remain constant.

$$E_\delta^{[A]} = \{\langle \sigma, \sigma + \delta, \chi \rangle | \quad \sigma \text{ is a location of } [A], \delta = 1, wait(\sigma, \delta), \text{ and } \langle \mathbf{f}, \mathbf{f} \rangle \models \chi \text{ for all observations } \mathbf{f} \}$$

If a clock x has the value K_x, further time increments leave the value of x unchanged. As described above, a transition of the timed automaton A on a timed event $\langle \delta, \mathbf{f}, \mathbf{f}' \rangle$ for $\delta \in \mathbb{N}^+$ is represented in $[A]$ by the traversal of δ time passage edges followed by one *control edge* that is taken on the untimed event $\langle \mathbf{f}, \mathbf{f}' \rangle$. Control edges incorporate the resets of clocks and the location changes of A into $[A]$. The set of control edges $E_c^{[A]}$ is defined as follows.

$$E_c^{[A]} = \{\langle \langle s, \Phi + \delta \rangle, \langle t, \Theta \rangle, \chi \rangle | \quad \delta \in \mathbb{N}^+, \text{ and } \langle s, \Phi \rangle \xrightarrow{\gamma} \langle t, \Theta \rangle \text{ in } A \text{ by taking some edge } e = \langle s, t, \varphi, \chi, R \rangle \text{ of } A \}$$

The set of edges of $[A]$ is then given by $E^{[A]} = E_\delta^{[A]} \cup E_c^{[A]}$. Because a run of a product automaton that has only integer time increments corresponds to a run of the same form in each of its components, the following proposition holds.

Proposition 7.3.2 (Compositionality) *For timed automata A and B, $[A \parallel B] = [A] \parallel [B]$.*

The delay computation algorithm presented in Section 7.3.2 is essentially a breadth-first search of the discrete state-space of the [] construction, using OBDDs to represent sets of locations and edges.

7.3.2 The Algorithm

For uniformity of notation, let us rename the automata comprising F to $C_1, ..., C_m$, such that $F = (C_1 \parallel ... \parallel C_m)$. We will to analyze $[F]$, which, by Proposition 7.3.2 is equivalent to $[C_1] \parallel ... \parallel [C_m]$. In the following, the components whose outputs are inputs to a component C_j are referred to as *fan-ins* of C_j. C_i is said to be in the *transitive fan-in* of C_j if there is a sequence $C_{k_0}, C_{k_1}, ..., C_{k_n}$ where $C_{k_0} = C_i$, $C_{k_n} = C_j$ and $C_{k_{i+1}}$ is a fan-in of C_{k_i} for all i.

In our delay computation algorithm, sets and relations will be represented using characteristic functions, which are defined next.

Definition 7.3.3 (Characteristic function) *Let $\mathcal{S}$ be a finite set, and $S \subseteq \mathcal{S}$. The* **characteristic function** *for S is the following Boolean function:*

$$S(x) = \begin{cases} 1 \text{ if } x \in S, \\ 0 \text{ otherwise} \end{cases}$$

An *implicit representation* for a set is a data structure corresponding to the set's characteristic function. Set operations such as union, complementation, membership checking, as well as operations on relations such as composition and quantification are implemented as logical operations on characteristic functions, each of which, in turn, is realized by an algorithm that operates on implicit representations of Boolean functions. Multi-valued Decision Diagrams (MDDs) [KB91], which are built on an infrastructure based on OBDDs are convenient and efficient implicit representations. All characteristic functions used in this chapter are represented and manipulated using the MDD package of [KB91].

For all j, we let $\boldsymbol{\sigma}_j$ denote a variable whose range is the set of locations $S^{[C_j]}$ of $[C_j]$. $\boldsymbol{\sigma} = \langle \boldsymbol{\sigma}_1, \boldsymbol{\sigma}_2, ..., \boldsymbol{\sigma}_m \rangle$ is a vector-valued variable that has as its range the set of locations of $[F]$. Each assignment to $\boldsymbol{\sigma}$ specifies an assignment to each of the $\boldsymbol{\sigma}_i$'s.

We represent the edges of a timed automaton using characteristic functions as follows: for automaton $[C_j]$, the functions T_j^c and T_j^δ denote the characteristic functions of the control and time increment edges, respectively. T_j^c and T_j^δ's are also referred to as "transition relations". Recall that the outputs $\boldsymbol{o}_j$ of each automaton $[C_j]$ are a (deterministic) function of its location given by $\boldsymbol{o}_j = \mu_j(s_j)$. Therefore, for the purposes of traversing the state-space of $[F]$, it is possible to express all transition relations and initial state relations as functions of the location variables $\boldsymbol{\sigma}_j$'s. For notational ease, we will assume that this is the case in

(1): $k \leftarrow 0$, $t \leftarrow 0$, $S^{(0)} = Init_1 \wedge Init_2 \wedge ... \wedge Init_m$
(2): repeat
(3): repeat
(4): $S^{(k+1)} = \textsc{OneStep}(S^{(k)}, \langle T_1^c, ..., T_m^c \rangle)$
(5): $k \leftarrow k + 1$
(6): while $S^{(k)}(\boldsymbol{\sigma}) \neq S^{(k-1)}(\boldsymbol{\sigma})$
(7): $S^{(k+1)} = \textsc{OneStep}(S^{(k)}, \langle T_1^\delta, ..., T_m^\delta \rangle)$
(8): $t \leftarrow t + 1$
(9): $k \leftarrow k + 1$
(10): while $S^{(k)}(\boldsymbol{\sigma}) \neq S^{(k-1)}(\boldsymbol{\sigma})$

Figure 7.8: Algorithm ComputeDelay

the rest of this chapter. For component j, a transition relation T_j is a function the form $T_j(\boldsymbol{\sigma}_{\rightarrow j}, \boldsymbol{\sigma}_j, \boldsymbol{\sigma}'_{\rightarrow j}, \boldsymbol{\sigma}'_j)$. Here $\boldsymbol{\sigma}_{\rightarrow j}$ denotes the set of location variables that determine the inputs of C_j, i.e., fan-ins of C_j. Primed variables refer to the value of the variable after the transition, i.e., the "new" value, as described in Section 2.1. $T_j(\sigma_{\rightarrow j}, \sigma, \tau_{\rightarrow j}, \tau)$ evaluates to "True" if and only if $\sigma \xrightarrow{\gamma} \tau$ in $[C_j]$ on an input event γ generated by the fan-ins of C_j making a transition from $\sigma_{\rightarrow j}$ to $\tau_{\rightarrow j}$. T_j^δ and T_j^c represent $E_\delta^{[C_j]}$ and $E_c^{[C_j]}$ respectively. Similarly, $Init_j(\boldsymbol{\sigma}_{\rightarrow j}, \boldsymbol{\sigma}_j)$ evaluates to "True" if and only if $\langle \mu^{[C_j]}(\boldsymbol{\sigma}_{\rightarrow j}), \boldsymbol{\sigma}_j \rangle \in Init^{[C_j]}$.

The structure of our delay computation algorithm is described in Figure 7.8. $S^{(k)}$ denotes the characteristic function for the set of states of $[F]$ that can be reached from the initial states by traversing a certain sequence of k edges. Here $Init_j$ is the characteristic function describing the initial location of component C_j as determined by the initial locations of its fan-ins.

$\textsc{OneStep}(S, \langle T_1, ..., T_m \rangle)$ (described in Figure 7.9) computes the states reached from $S^{(k)}$ by traversing one edge in the transition relation given by $T = T_1 \wedge T_2 \wedge ... \wedge T_m$. $\boldsymbol{\sigma}' \leftarrow \boldsymbol{\sigma}$ is shorthand for replacing variable $\boldsymbol{\sigma}'$ with $\boldsymbol{\sigma}$. The relation $S^{(k,k+1)}$ represents the outgoing transitions from the locations $S^{(k)}(\boldsymbol{\sigma})$ as a function of $\boldsymbol{\sigma}$, where $\boldsymbol{\sigma}$ and $\boldsymbol{\sigma}'$ correspond to the present and next locations, respectively. $S^{(k,k+1)}(\boldsymbol{\sigma}, \boldsymbol{\sigma}')$ evaluates to true if $\boldsymbol{\sigma}'$ can be reached from $\boldsymbol{\sigma}$ by the traversal of one edge in the transition relation given by $T_1 \wedge ... \wedge T_m$.

The algorithm ComputeDelay repeats the following loop until the circuit stabilizes: First, control transitions are explored until all states reachable through them are computed, then a time increment of 1 is taken by traversing time passage transitions only. The delay

$$
\begin{array}{l}
\textsc{Onestep}(S^{(k)}, \langle T_1, ..., T_m \rangle) \\
\quad S^{(k,k+1)}(\boldsymbol{\sigma}, \boldsymbol{\sigma'}) = S^{(k)} \wedge \bigwedge_{i=1}^{m} T_i \\
\quad S^{(k+1)}(\boldsymbol{\sigma'}) = (\exists \boldsymbol{\sigma}) S^{(k,k+1)}(\boldsymbol{\sigma}, \boldsymbol{\sigma'}) \\
\quad S^{(k+1)}(\boldsymbol{\sigma}) = (\boldsymbol{\sigma'} \leftarrow \boldsymbol{\sigma})\ S^{(k+1)}(\boldsymbol{\sigma'})
\end{array}
$$

Figure 7.9: Algorithm Onestep.

of the circuit is then the maximum t such that the corresponding $S^{(k,k+1)}$ includes an edge for which there is a change in a primary output. It is a simple OBDD operation to check whether $S^{(k,k+1)}$ contains such an edge. Note that minimum delay computation, as well as checking state invariants[2] can easily be incorporated into this scheme.

The algorithm ComputeDelay is different from a generic breadth-first search in two aspects: (i) a subset of the outgoing edges at the locations are considered at each step (ii) the set of locations explored up to step k ($\bigcup_{j=1}^{k} S^{(j)}$) is not stored. On a general graph, this algorithm would be inefficient and might not terminate. However, certain topological properties of the transition structure of $[F]$ make ComputeDelay suitable for exploring it. This is the topic of the next section.

7.3.3 The Region Automaton is (Virtually) Acyclic

For a properly designed combinational circuit, the transition structure of $[F]$ is acyclic, except for the following two kinds of self-loops, which we will refer to as *trivial cycles*:

(i) Each location has a self-loop on the stuttering event, as described in Section 2.3.

(ii) Locations where all clocks x have reached their upper limits K_x also have self-loops on time passage events.

Let us examine the implications of a non-trivial cycle before arguing that $[F]$ has no such cycles. First observe that, because primary input waveforms are modeled by automata with no inputs, F is a closed system, that is, it has no external inputs. Therefore, any path in $[F]$ is traversable and any cycle in $[F]$ is traversable an unbounded number of times. Now consider the projections of a non-trivial cycle of $[F]$ onto the component automata $[C_j]$. At least one of these projections must be non-trivial. Each component automaton is either a

[2]State invariants are formulas whose truth can be decided by looking at the assignments to state variables only. State invariants are required to hold in every reachable state.

gate or wire delay model, or outputs a primary input waveform. In each case, a non-trivial cycle in a component is possible if only if its inputs or outputs change their values. Thus, a non-trivial cycle in $[F]$ points to an oscillation in some nodes of the circuit that does not die down.

The primary input waveforms that we consider in delay analysis remain constant after a certain point in time (see Section 7.2.1). Combinational circuits are designed to stabilize to a certain final state if the inputs remain constant. For a correctly designed combinational circuit, non-stabilizing oscillations in circuit nodes should not occur, and $[F]$ should not contain any non-trivial cycles.

If the circuit does have unstable behavior and non-trivial cycles, the algorithm described in Section 7.3.2 will not converge. To detect this undesirable case, we observe that the delay of a given acyclic circuit is bounded from above by the delay of the longest topological path. Using this fact, we can place a bound on the maximum number of iterations of ComputeDelay and conclude that if the termination criterion has not been satisfied by then, the circuit must have oscillatory behavior. We will now formalize this argument and, afterwards restrict our attention to combinational circuits that stabilize properly.

Definition 7.3.4 (Topological delay of a circuit component) *Let A be a timed automaton representing a circuit component, and let $\mathcal{L}_{\leq t} \subseteq \mathcal{L}(A)$ denote the traces of A where all of the input variables remain stable after time t. The* topological delay *of A is the largest δ such that, for some t and a trace in $\mathcal{L}_{\leq t}$, an output of A makes a transition at $t + \delta$. Formally, δ is least upper bound of the following set*

$$\left\{ \Delta \;\middle|\; lim_{x \to (t+\Delta)^-} \left(F_{\overline{\gamma}}(O^A)(x) \right) \neq lim_{x \to (t+\Delta)^+} \left(F_{\overline{\gamma}}(O^A)(x) \right) \text{ for some } \overline{\gamma} \in \mathcal{L}_{\leq t},\, t \in \mathbb{N} \right\}$$

The topological delay of A is denoted by $\delta(A)$.

According to this definition, the input variables are allowed to change their values arbitrarily up to time t. As a result, at time t, the timed automaton $[A]$ can be in any of its reachable locations. To determine A's topological delay, we must determine the latest time an output transition can occur, starting from any reachable location. For an arbitrary automaton, this problem is just as complex as the delay computation problem itself and the output of such an automaton may not stabilize at all even though all of its inputs remain constant. However, for the automata that we use as delay models, each clock gets reset only when there is a change in the value of an input variable and edges that correspond

to output changes have non-trivial clock predicates on them. Therefore, $max_{x \in X} K_x$ is an upper bound on the topological delay of each circuit component. This upper bound is accurate enough for our purposes because we will use topological delay only to prove that our algorithms are correct. To be able to guarantee termination, we require of timed automata that will be used as delay models to have finite topological delay.

Definition 7.3.5 (Topological delays of circuit nodes) *For a given circuit, let t_{in} denote the time when the latest primary input becomes stable and let n_j be a circuit node which is an output of automaton C_j. The topological delay δ_j at node n_j is defined recursively as follows:*

- *If C_j is a primary input automaton, $\delta_j = \delta(C_j) - t_{in}$*
- *If C_j corresponds to a circuit component,*

$$\delta_j = \delta(C_j) + max_{C_k \text{ a fan-in of } C_j} \delta_k$$

Observe that for topological delays at nodes to be well-defined, the graph of input-output dependencies between the circuit components C_j's must be acyclic. This would trivially hold for acyclic combinational circuits if there were no cross-talk. Cross-talk may introduce circular dependencies between circuit nodes: the transitions at a node n_j may have an effect on the transitions at n_k, which in turn may lie in the transitive fan-in n_j. Our proof of termination for ComputeDelay assumes that such cyclic dependencies do not exist, in other words, if there is a cyclic dependency between two variables, then they must belong to the same component. For automata modeling coupled elements, this requirement is naturally satisfied. If there are cyclic dependencies that span multiple components, the components that lie on such cycles must be grouped together to form a larger component. The timed automaton representing this larger component is obtained by composing the component automata and hiding the variables representing the internal nodes[3]. The difficulty with this approach is the fact that cross-talk dependencies may require large portions of the circuit to be grouped together and computations on such components may be costly. Unless some of these dependencies can be ruled out by other means, however, this complexity seems unavoidable. In the rest of this chapter, we assume an acyclic dependency graph between the circuit components.

[3]If the circuit is designed and modeled properly, this larger component will have a finite topological delay.

Fact 7.3.6 *It follows from Definition 7.3.4 that each node n_j is stable after $t_{in} + \delta_j$. Also, at time $t_{in} + \delta_j$, $[C_j]$ is at a location of the form $\langle s, \Phi \rangle$ where $\Phi(x) = K_x$ for all $x \in X^A$, i.e., each clock has reached its upper bound. $[C_j]$ does not change its location thereafter.*

While exploring a large state transition graph, the bottleneck is often the size of the representation for the set of reached states. ComputeDelay avoids maintaining a representation for this set by using the fact that $[F]$ has no non-trivial cycles. Because of this, the correctness of ComputeDelay is not immediate. Lemma 7.3.7 and Theorem 7.3.9 address this issue.

Lemma 7.3.7 *The inner loop of* ComputeDelay *terminates each time it is executed, and upon its termination, $S^{(k)}$ consists of all locations of $[F]$ reachable at time t.*

Proof: First note the following: while the inner loop of ComputeDelay is being iterated, because each location has a self-loop on the stuttering event (which is contained in T_j^c for all j), $S^{(k+1)}$ contains $S^{(k)}$ for each k. Thus, the inner loop is essentially a breadth-first search on the sub-graph of $[F]$ consisting of the control edges. It is guaranteed to terminate since $[F]$ has finitely many locations.

Proposition 7.3.8, which we will prove by induction on $t \in \mathbb{N}$, implies the lemma. We first define some shorthand: c represents a control edge and δ represents a time passage transition. We use regular language notation (see, for example, [HU79]) to describe a sequence of edges. For instance, $c(\delta + c)$ represents the sequences of two edges that start with a control edge and that may end in a control or time passage edge.

Proposition 7.3.8 *After the inner loop of* ComputeDelay *terminates $S^{(k)}$ consists of the set of locations that are reachable from an initial location by traversing a sequence of edges of the form $(c^*\delta)^t c^*$, where ^{t} is shorthand for concatenation exactly t times.*

Proof: From the argument preceding the proposition we conclude that for $t = 0$, when the inner loop terminates, $S^{(k)}$ contains all locations reachable from initial locations without traversing a time passage edge. Thus, the proposition holds for $t = 0$. Suppose that the proposition holds for all $t \leq l$ and consider $S^{(k)}$ after the inner-loop has terminated. After line (7) is executed, by the induction assumption $S^{(k+1)}$ contains all locations reached by paths of the form $(c^*\delta)^l c^*\delta = (c^*\delta)^{(l+1)}$. Running the inner loop on this result until

termination computes all locations reached by paths of the form $(c^*\delta)^{(l+1)}c^*$, which proves the proposition and the lemma. ■

In fact, since every non-stuttering edge is associated with either an input or an output change in component automata, the number of iterations of the inner loop is bounded by the longest chain of input-output dependencies, i.e., the largest number of components that lie on the same path in the circuit.

Theorem 7.3.9 (ComputeDelay) *Suppose that* $[F]$ *contain no non-trivial cycles. Then*

(i) If a location σ *of* $[F]$ *is reachable, it is explored by* ComputeDelay, *and*

(ii) ComputeDelay *terminates.*

Proof:

(i) Straightforward from Lemma 7.3.7.

(ii) Let $\Delta = max_j \delta_j$ be the largest topological delay of any node in F. Fact 7.3.6 implies that every component automaton of $[F]$ remains at a constant location for values of $t \geq \Delta + t_{in}$. Thus, the termination condition of line (10) is satisfied at the latest when $t \geq \Delta + t_{in}$.

■

ComputeDelay allows a heuristic memory-time trade-off: Typically, memory is saved by storing a smaller set, but a location may be visited more than once, which results in re-computation. For an arbitrary system, the amount of re-computation could be prohibitively large. However, the delay of a combinational circuit is bounded by the delay of the maximum topological delay of its nodes, which places a polynomial bound on k. Provided that it is not costly, information about the locations in $S^{(0)} \cup ... \cup S^{(k-1)}$ can be used to minimize the size of the representation of $S^{(k-1)}$ or to reduce repeated exploration of the same location.

In practice, we found that this approach results in significant savings in memory (See Section 7.4). Our technique, like many OBDD based methods, is memory limited, and by representing $S^{(k)}$ only, we were able to handle circuits that we could not have handled otherwise. The practical limitation in applying the algorithm of Section 7.3.2 is the size of the OBDD for $S^{(k)}$. In the next section, we present a method for efficiently computing and representing $S^{(k)}$'s.

7.3.4 A Conjunctively Decomposed Representation

$S^{(k)}(\boldsymbol{\sigma})$ typically has a large number of variables in its support. It is well-known that in order to get compact monolithic OBDD representations, it is necessary to choose a good order of the variables [Bry86, ATB93, ATB94]. Heuristically, variables that are strongly related must be close to each other in the order, and the corresponding bits encoding correlated integers must be interleaved [KB91]. For the problem at hand, it is in general not possible to find a total order that satisfies these constraints, especially because all variables corresponding to timers are correlated with each other [ABH+97]. To deal with this difficulty, we employ a conjunctively decomposed representation for the sets of locations $S^{(k)}$. The decomposition corresponds to a "slicing" of the circuit in the following manner: the circuit is partitioned into slices $SL_1, ..., SL_p$, where each slice SL_i consists of a set of circuit elements (C_i's). The slices cover the circuit and no C_i belongs to more than one slice. The SL's are ordered topologically, i.e., if $a < b$ no C_i's in SL_b can be in the transitive fan-in of a C_j in SL_a. (See Figure 7.11 for an example.).

$S^{(k)}$ is then represented by a collection of relations, each corresponding to a slice. We construct $S_1^{(k)}(\boldsymbol{\alpha}_0)$, $S_1^{(k)}(\boldsymbol{\alpha}_0, \boldsymbol{\alpha}_1)$, $S_2^{(k)}(\boldsymbol{\alpha}_1, \boldsymbol{\alpha}_2)$, $\ldots, S_p^{(k)}(\boldsymbol{\alpha}_{p-1}, \boldsymbol{\alpha}_p)$ such that each $S_j^{(k)}$ specifies the set of locations of SL_j at step k of the algorithm. Here $\boldsymbol{\alpha}_i$ is a vector-valued variable the components of which specify the locations of the automata within slice i. We also define

$$\boldsymbol{T}_i(\boldsymbol{\alpha}_{i-1}, \boldsymbol{\alpha}_i, \boldsymbol{\alpha}'_{i-1}, \boldsymbol{\alpha}'_i) = \bigwedge_{C_j \in SL_i} T_j(\boldsymbol{\sigma}_{\to j}, \boldsymbol{\sigma}_j, \boldsymbol{\sigma}'_{\to j}, \boldsymbol{\sigma}'_j)$$

and

$$\boldsymbol{Init}_i(\boldsymbol{\alpha}_{i-1}, \boldsymbol{\alpha}_i) = \bigwedge_{C_j \in SL_i} Init_j(\boldsymbol{\sigma}_{\to j}, \boldsymbol{\sigma}_j)$$

Note that each $S_j^{(k)}$ has $\boldsymbol{\alpha}_{j-1}$ in its support in addition to $\boldsymbol{\alpha}_j$. Intuitively, $S_j^{(k)}$ specifies what assignments to $\boldsymbol{\alpha}_j$ are included in $S^{(k)}$ for each assignment to $\boldsymbol{\alpha}_{j-1}$, that is, the correlation between the location variables of adjacent slices is captured. Since each $S_j^{(k)}$ has much fewer variables in its support, the total size of the $S_j^{(k)}$'s is in general much smaller than a monolithic representation for $S^{(k)}$.

For $j = 1$ to p

$$S_j^{(k,k+1)}(\boldsymbol{\alpha}_{j-1}, \boldsymbol{\alpha}_j, \boldsymbol{\alpha}'_{j-1}, \boldsymbol{\alpha}'_j) \;=\; \left((\exists \boldsymbol{\alpha}_{j-2}, \boldsymbol{\alpha}'_{j-2})\; S_{j-1}^{(k,k+1)}\right) \wedge\; S_j^{(k)} \wedge\; \boldsymbol{T}_j$$

$$S_j^{(k+1)}(\boldsymbol{\alpha}'_{j-1}, \boldsymbol{\alpha}'_j) = (\exists \boldsymbol{\alpha}_{j-1}, \boldsymbol{\alpha}_j)\; S_j^{(k,k+1)}$$

Figure 7.10: Algorithm ONESTEP'

To conform to this new representation, line (1) of algorithm COMPUTEDELAY is replaced by

$$\begin{aligned}(1):\quad & k \leftarrow 0, t \leftarrow 0,\\ & S_1^{(0)} = \boldsymbol{Init}_1(\boldsymbol{\alpha}_1),\\ & S_j^{(0)} = \boldsymbol{Init}_j(\boldsymbol{\alpha}_{j-1}, \boldsymbol{\alpha}_j) \wedge (\exists \boldsymbol{\alpha}_{j-2})\; S_{j-1}^{(0)}(\boldsymbol{\alpha}_{j-2}, \boldsymbol{\alpha}_{j-1}) \text{ for all } j > 1.\end{aligned}$$

and algorithm ONESTEP is replaced by the modified algorithm described in Figure 7.10, which performs one step of reachability computation operating on one slice at a time. For slice j, the transitions in slice $j-1$ provide the input events as expressed by $(\exists \boldsymbol{\alpha}_{j-2}, \boldsymbol{\alpha}'_{j-2}) S_{j-1}^{(k,k+1)}$. $S_j^{(k)}$ relates at step k the locations of slice j to those of $j-1$, therefore, for each location of slice j the corresponding set of possible inputs events are specified by $(\exists \boldsymbol{\alpha}_{j-2}, \boldsymbol{\alpha}'_{j-2}) S_{j-1}^{(k,k+1)}$.

Ideally, for such a decomposed representation, one would like the following equalities to hold: (i) $S^{(k)} = \bigwedge_{j=1}^{p} S_j^{(k)}$, and (ii) $S_j^{(k)} = (\exists \boldsymbol{\tau}_j) S^{(k)}$, where $\boldsymbol{\tau}_j$ is the set of variables not in the support of $S_j^{(k)}$. This would be the case if, at the kth iteration of the algorithm, the location variables of slice k were a function of those of slice $k-1$ only. However, for an arbitrary slicing and arbitrary circuit components, location variables in non-adjacent slices may be correlated. Lemmas 7.3.10 and 7.3.11 prove that ONESTEP' is conservative, i.e., for each slice it computes a superset of the reached locations. Let us first define $\boldsymbol{\rho}_j = \langle \boldsymbol{\alpha}_1, \boldsymbol{\alpha}_2, \text{-} ..., \boldsymbol{\alpha}_{j-2}, \boldsymbol{\alpha}_{j+1}, \boldsymbol{\alpha}_{j+2}, ..., \boldsymbol{\alpha}_p \rangle$, i.e., the location variables that are not in the support of $S_j^{(k)}$, and

$$\mathcal{T}_j^{(k,k+1)}(\boldsymbol{\alpha}_{j-1}, \boldsymbol{\alpha}_j) = (\exists \boldsymbol{\rho}_j, \boldsymbol{\rho}'_j) S^{(k,k+1)}$$

Lemma 7.3.10 *For all j, $S_j^{(0)}$ is a superset of $(\exists \boldsymbol{\rho}_j)\; S^{(0)}$.*

Proof: By the fact that initial state relations must be total, $(\exists \boldsymbol{\rho}_1) S^{(0)} = \boldsymbol{Init}_0 = S_1^{(0)}$, thus the statement holds for $j = 1$. Assuming that it holds for all slices before j, let us

show that it holds for slice j.

$$\begin{aligned}(\exists \boldsymbol{\rho}_j) S^{(0)} &= \boldsymbol{Init}_j \wedge \\ &\quad (\exists \boldsymbol{\alpha}_{j-2})\,(\exists \boldsymbol{\alpha}_1, ..., \boldsymbol{\alpha}_{j-3}, \boldsymbol{\alpha}_{j+1}, ..., \boldsymbol{\alpha}_n) \left(\bigwedge_{1 \le i < j} \boldsymbol{Init}_i \bigwedge_{i > j} \boldsymbol{Init}_i \right) \\ &= \boldsymbol{Init}_j \wedge (\exists \boldsymbol{\alpha}_{j-2})\,(\exists \boldsymbol{\alpha}_1, ..., \boldsymbol{\alpha}_{j-3}) \bigwedge_{1 \le i < j} \boldsymbol{Init}_i \\ &\quad \wedge (\exists \boldsymbol{\alpha}_{j+1}, ..., \boldsymbol{\alpha}_n) \bigwedge_{i > j} \boldsymbol{Init}_i \end{aligned} \tag{7.1}$$

$\boldsymbol{Init}_j$ has been carried outside of the quantifiers as it does not depend on $\boldsymbol{\rho}_j$. By the fact that initial state relations must be total, $(\exists \boldsymbol{\alpha}_{j+1}, ..., \boldsymbol{\alpha}_n) \bigwedge_{i>j} \boldsymbol{Init}_i$ is a tautology. By the induction assumption,

$$(\exists \boldsymbol{\alpha}_1, ..., \boldsymbol{\alpha}_{j-3}) \bigwedge_{1 \le i < j} \boldsymbol{Init}_i \subseteq S^{(0)}_{j-1}$$

since S_{j-1} does not depend on $\boldsymbol{\alpha}_j$. Putting all these together yields

$$(\exists \boldsymbol{\rho}_j) S^{(0)} \subseteq \boldsymbol{Init}_j \wedge \left(\exists \boldsymbol{\alpha}_{j-2} S^{(0)}_{j-1} \right) = S^{(0)}_j$$

■

Lemma 7.3.11 *If* ComputeDelay *is implemented using* Onestep', $S_j^{(k,k+1)}$ *is a superset of* $\mathcal{T}_j^{(k,k+1)}$

Proof: By induction on $l =_{def} pk + j$, the number of times that next location ($S^{(k,k+1)}_{\cdot}$) computation is performed on a given circuit. The base case consists of all $l \le p$ and is implied by Lemma 7.3.10. For $k \ge 1$, let us assume the lemma holds for all $\le l$ and prove that it holds for l also.

$$\begin{aligned}\mathcal{T}_j^{(k,k+1)} &= (\exists \boldsymbol{\rho}_j, \boldsymbol{\rho}'_j)(S^{(k)} \bigwedge_{1 \le i \le p} \boldsymbol{T}_i) \\ &\subseteq (\exists \boldsymbol{\rho}_j, \boldsymbol{\rho}'_j) S^{(k)} \wedge (\exists \boldsymbol{\rho}_j, \boldsymbol{\rho}'_j) \left(S^{(k)} \bigwedge_{1 \le i \le p} \boldsymbol{T}_i \right) \\ &\subseteq S_j^{(k)} \wedge (\exists \boldsymbol{\rho}_j, \boldsymbol{\rho}'_j) \left(S^{(k)} \bigwedge_{1 \le i \le p} \boldsymbol{T}_i \right) \end{aligned} \tag{7.2}$$

where the last $\subseteq$ follows from the induction assumption. We also have

$$\begin{aligned}
&(\exists \boldsymbol{\rho}_j, \boldsymbol{\rho}'_j) \left(S^{(k)} \bigwedge_{1 \le i \le p} \boldsymbol{T}_i \right) \\
&= \boldsymbol{T}_j \wedge (\exists \boldsymbol{\alpha}_{j-2}, \boldsymbol{\alpha}_{j-2})(\exists \boldsymbol{\alpha}_1, \boldsymbol{\alpha}'_1, ..., \boldsymbol{\alpha}_{j-3}, \boldsymbol{\alpha}'_{j-3}, \\
&\qquad\qquad \boldsymbol{\alpha}_{j+1}, \boldsymbol{\alpha}'_{j+1}, ..., \boldsymbol{\alpha}_n, \boldsymbol{\alpha}'_n) \left(S^{(k)} \bigwedge_{1 \le i \le p} \boldsymbol{T}_i \right) \\
&\subseteq \boldsymbol{T}_j \wedge (\exists \boldsymbol{\alpha}_{j-2}, \boldsymbol{\alpha}_{j-2})(\exists \boldsymbol{\alpha}_1, \boldsymbol{\alpha}'_1, ..., \boldsymbol{\alpha}_{j-3}, \boldsymbol{\alpha}'_{j-3}, \\
&\qquad\qquad \boldsymbol{\alpha}_j, \boldsymbol{\alpha}'_j, \boldsymbol{\alpha}_{j+1}, \boldsymbol{\alpha}'_{j+1}, ..., \boldsymbol{\alpha}_n, \boldsymbol{\alpha}'_n) \left(S^{(k)} \bigwedge_{1 \le i \le p} \boldsymbol{T}_i \right) \\
&\subseteq \boldsymbol{T}_j \wedge (\exists \boldsymbol{\alpha}_{j-2}, \boldsymbol{\alpha}_{j-2}) S_{j-1}^{(k,k+1)} \qquad (7.3)
\end{aligned}$$

where, again, the last $\subseteq$ follows from the induction assumption. Putting together (7.2) and (7.3) we obtain

$$\begin{aligned}
\mathcal{T}_j^{(k,k+1)} &\subseteq S_j^{(k)} \wedge (\exists \boldsymbol{\rho}_j, \boldsymbol{\rho}'_j) \left(S^{(k)} \bigwedge_{1 \le i \le p} \boldsymbol{T}_i \right) \\
&\subseteq S_j^{(k)} \wedge \boldsymbol{T}_j \wedge (\exists \boldsymbol{\alpha}_{j-2}, \boldsymbol{\alpha}_{j-2}) S_{j-1}^{(k,k+1)} \\
&= S_j^{(k,k+1)} \qquad (7.4)
\end{aligned}$$

which is what we sought. ■

Lemma 7.3.11 proves that ComputeDelay implemented using Onestep' is an over-approximation, however, in practice, we have not found any cases where the results of the slicing method is different from the results of the monolithic but exact method. We offer the following intuitive explanation for this fact: Combinational circuit elements have bounded memory, and thus, their locations are only strongly correlated with the locations of other elements close-by. Let us call a circuit component "active" if it is in a transient location. Suppose that the circuit is sliced in such a way that the topological delay through each slice is roughly equal. Then, at any given time, the active elements are contained in a portion of the circuit consisting of two adjacent slices. Our algorithm performs traversal by sweeping the circuit using a window consisting of two slices. If transitions at any point in time are confined to two adjacent slices, then next-location computation as performed by Onestep' is exact. An exact algorithm based on this intuition will be part of our future research. The algorithm dynamically focuses only on the active portion of the circuit.

7.4 Experimental Results

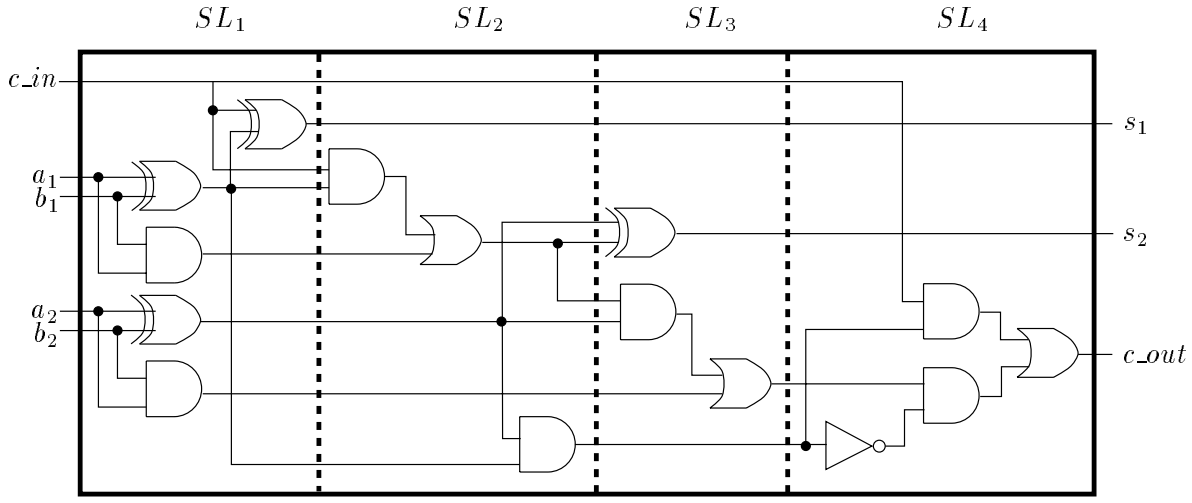

Figure 7.11: Two-bit carry-skip adder block. The dashed lines indicate the boundaries between the slices.

We conducted two sets of experiments. The first set, described in Section 7.4.1 compares the performance and accuracy of conventional delay analysis methods, a circuit simulator and the algorithm described in this paper. The second set, described in Section 7.4.2 demonstrates that our algorithm for traversing the state-space scales much better than a non-specialized OBDD based method.

As our benchmark circuits, we use n-bit adders built by cascading two-bit Carry Skip Adder (CSA) blocks (Figure 7.11). These circuits are known to have false paths, therefore, even in the absence of cross-talk, a topological analysis over-approximates delay.

7.4.1 Comparison with conventional methods

For this experiment, we constructed a 4-bit adder using two two-bit CSAs (Figure 7.12). Two configurations of this circuit were used:

K_1: The c_{out} output of CSA1 and primary input A3 were neighboring wires in the circuit layout, resulting in a possible delay variation for both due to cross-talk.

K_2: No cross-talk exists between any wires.

These configurations were designed to be "proof-of-concept" circuits, meant to highlight the difference between various algorithms. For an industrial strength tool, cross-talk infor-

mation needs to be extracted for the whole circuit from the layout. Here, we performed this only for the pair of wires c_{out} and A3.

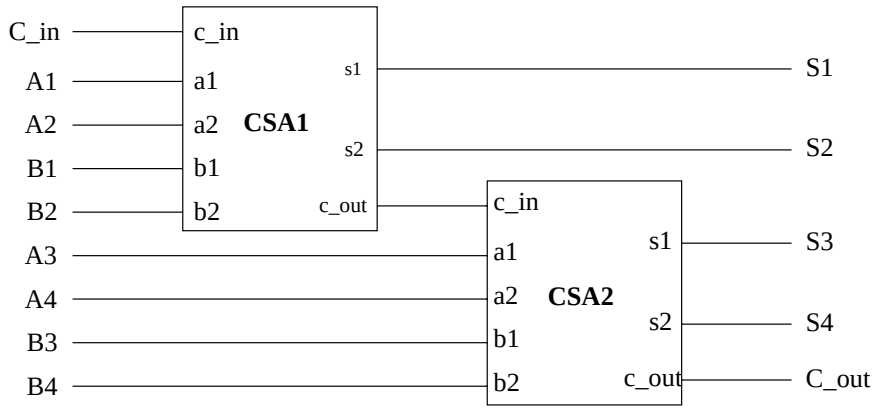

Figure 7.12: A 4-bit CSA

Four different algorithms were tested on this circuit:

A_1: *Circuit Simulation:* The circuit was modeled at the transistor level and analyzed using SPICE. The parameters of the wire cross-talk model were derived using SPACE. Exact delays were computed by simulating the input vector pair that causes largest delay. The run-time for exhaustive simulation of all input vector pairs was estimated by multiplying the run-time for one vector pair by the number of possible pairs. For this circuit, it was possible to determine by inspection the input vector pairs causing worst-case behavior. The one we used in our simulation was

$$\begin{aligned} (\text{C\_in, A1, A2, B1, B2, A3, A4, B3, B4}) &= (0,0,0,0,0,1,0,0,0) \\ (\text{C\_in', A1', A2', B1', B2', A3', A4', B3', B4'}) &= (0,1,1,0,1,0,1,0,1) \end{aligned}$$

This is in general not possible, and exhaustive simulation needs to be performed.

A_2: *Our Approach:* We first created timed automata for each gate using the ideas described in Section 7.2.2. This was done for each of the 4 gate types in the circuit of Figure 7.11. Similarly, a timed automaton model for cross-talk between two wires was constructed as discussed in Section 7.2.3. The value of different delays was determined by a transistor level simulation, using SPICE. For the gate delay models, the numbers obtained were rounded to the nearest multiple of 5 ps. The maximum delay of the

circuit was computed using the algorithm described section 7.3. Our algorithms were implemented on the verification platform MOCHA [AHM+98].

A_3: *"Exact timing analysis"*: This is the algorithm described in [MSBSV93], implemented on the SIS platform. This algorithm does not account for cross-talk, but reports the delay of the longest true path. Although this method is not truly exact, it is referred to as "exact timing analysis" in the delay computation literature. The implementation we used did not allow input-dependent delays, so worst-case delay parameters obtained from SPICE simulation were used for gates and wires.

A_4: *Topological delay analysis:* This algorithm computes topological delay in a way similar to that described in Section 7.3.3. The implementation in SIS [SSL+92] was used. This method does not model cross-talk, and does not account for false paths. Again, worst-case delay parameters obtained from SPICE simulation were used for gates and wires.

Observe that the input vector sequence which results in the maximum delay for the 4-bit CSA also results in the c_{out} output of CSA1 switching in the opposite direction as the primary input A3. However, these signals are significantly separated in time, so there is no increase in circuit delay due to cross-talk between the wires. An algorithm which is not cross-talk-aware will assign a worst-case delay for each of the wires, and hence estimate a larger delay for configuration K_1 than for configuration K_2, whereas, in reality, the two configurations result in the same delay.

	With cross-talk (K_1)		Without cross-talk (K_2)	
Method	run-time(s)	max delay (ps)	run-time(s)	max delay (ps)
A_1	2.936×10^6	611	3.09×10^6	616
A_2	602	660	617	660
A_3	1.745	770	1.619	740
A_4	0.1	920	0.1	890

Table 7.1: Experimental Results

The results of these runs are described in table 7.4.1. SPICE models circuit behavior most closely, and we take the delay that SPICE computes as our reference. Note that the

maximum delay that SPICE computes for the two configurations is almost the same, as expected.

The results obtained from our algorithm (A_2) are promising:

- Our method correctly determines that the delay of the circuit is identical whether configuration K_1 or K_2 is used.
- The computed delay is within 10% of the true circuit delay as computed by SPICE. A_3 and A_4 deviate from SPICE by approximately 20% and 50% respectively.
- The run-time of our algorithm is three orders of magnitude less than the estimated run-times for SPICE.

The discrepancy of our delay figures from those obtained by SPICE is partly due to the fact that in the timed-automaton models for gates, delays are rounded up to the nearest multiple of 5 ps. The more important reason, however, is the fact that we compute gate delays assuming a nominal loading. In the circuit, each instance of any gate drives a different load. This fact is naturally taken into account by SPICE. If we had incorporated this factor into our delay models, we would have had a larger number of models, but our results would have been closer to SPICE.

Since algorithm A_3 cannot detect that cross-talk does not actually take place in K_1, it reports a larger delay than for K_2. This is due to the fact that cross-talk in this case was modeled using buffers whose delays are the worst-case delay under cross-talk (t_2^{opp} in Figure 7.6). Also, the value of delay computed by this scheme for configuration K_2 is larger than that computed by timed automata. This is because the delay model used by A_3 is a pin-delay model, and does not consider input pattern dependency of delays.

The topological timing analysis scheme (algorithm A_4) has the lowest run-times, but gives the most inaccurate results. False paths in the circuit are not detected, and further, cross-talk is not modeled.

These results demonstrate that the method developed in this chapter is powerful, significantly faster than transistor-level simulation, and more accurate than conventional timing analysis methods.

7.4.2 Comparison with non-specialized traversal methods

To quantify the improvement brought about by our traversal heuristics, we compared their performance with a non-specialized traversal algorithm. The non-specialized algorithm made use of a partitioned representation of the transition relation while keeping the representation of $S^{(k)}$'s monolithic. The algorithms were run on a family of n-bit carry-skip adders as described earlier. For this set of experiments, we used the same delay models with a coarser time discretization (1 time unit = approx. 14 ps) in order to demonstrate how the algorithm's performance scales with circuit size[4]. The results presented in Table 7.2 show that our algorithm scales remarkably well. In contrast, the non-specialized algorithm ran out of space (1GB) for the 4-bit CSA and was not able to complete even with the help of dynamic variable reordering heuristics. Observe that, with the help of our heuristics, we were able to handle models with thousands of OBDD variables.

no. of CSAs	no. of timers	circuit delay (ps.)	no. of BDD vars	BDD mem.(MB)	CPU time
2	30	634	526	29	2 min
3	45	767	820	58	8 min.
4	60	900	1114	78	13 min.
5	75	1034	1408	85	21 min.
6	90	1167	1702	102*	30 min.
8	120	1440	1180	63†	15 min.

Table 7.2: Performance of ComputeDelay on a family of n-bit carry-skip adders. Experiments were run on a DEC Alpha 8400 5/625 with 2Gb of memory. * denotes dynamic OBDD variable reordering. † denotes a coarser time discretization (1 time unit = approx. 35 ps.). The second column gives the number of timer variables used in modeling each circuit. The fourth and fifth column indicate the number of OBDD variables in the circuit representation and the total memory used by OBDDs.

Table 7.3 presents the performance of ComputeDelay where inertial gate delay models with non-determinism (Figure 2.1) were used instead of the delay models described in Sections 7.2.3 and 7.2.2. The performance and the scaling behavior of the algorithm are essentially the same, which indicates that our method does not derive its efficiency from the particular delay models used.

[4]The slicing heuristic addresses the computational complexity that is due to the circuit size, but not the exponential dependency on the timing constants used. To be able to demonstrate the former, we de-emphasized the latter by reducing the sizes of delay parameters.

no. of CSAs	no. of clocks	$[\delta_{min}, \delta_{max}]$ (approx.)	circuit delay	no. of BDD vars	BDD mem.(MB)	CPU time
2	30	$[23, 69]$	617	364	40	5.3 min.
3	45	$[23, 69]$	754	540	56	13 min.
4	60	$[23, 69]$	891	716	72	21 min.
5	75	$[23, 69]$	1029	892	85	24 min.
6	90	$[23, 69]$	1166	1068	$85^{\star}$	75 min.
8	120	$[0, 69]$	1440	1180	$40^{\dagger}$	7.7 min.
16	240	$[0, 69]$	2537	2362	$78^{\star,\dagger}$	80 min.

Table 7.3: Experimental results for n-bit CSAs. * denotes dynamic OBDD variable reordering. † denotes a coarser time discretization (1 time unit = approx. 35 ps.). The second column gives the number of timer variables used in modeling each circuit. The fourth and fifth column indicate the number of OBDD variables in the circuit representation and the total memory used by OBDDs.

While the results reported in Tables 7.2 and 7.3 are a significant improvement over previous methods based on timed automata, they also indicate that circuits with several thousand gates are still beyond the reach of our algorithm. As is, the algorithm ComputeDelay is too complex for analyzing practical circuits in their entirety. Therefore, we propose that it be applied to determine the delay of critical portions and circuit nodes in their electrical neighborhoods, for which an accurate formal analysis is crucial. We further conjecture that it is possible in many cases to decompose the delay computation problem and make it amenable to formal analysis with timed automata. Our conjecture is based on the following observation: to achieve higher clock speeds, there is a trend in high performance designs to employ a larger number of pipeline stages and consequently fewer levels of logic between latches. As a result, the transitive fan-ins of a primary output[5] contains a smaller number of circuit nodes and primary inputs than earlier designs. This allows the delay computation problem to be naturally decomposed by considering a (small) subset of primary outputs to analyze at a time. By choosing the subsets in a way that maximizes the overlap of their transitive fan-ins, it may be possible to focus on a manageably small subcircuit at a time and apply ComputeDelay. These subcircuits will have overlaps, but since our method is memory limited, it is justified to reduce the memory requirements at the expense of time for re-computation. In our future research, we intend to test this conjecture

[5]For the purposes of delay analysis, we consider latch inputs as primary outputs of, and latch outputs as primary inputs to the combinational circuit.

on recent industrial circuits.

To be able to analyze "deep and narrow" designs, which make use of a large number of levels of logic and fewer latches, we propose a hierarchical delay computation method. Hierarchical computation, while conservative, sacrifices some accuracy to achieve computational efficiency. For designs with many levels of logic, the design goal is more often to meet a certain clock speed than to determine the delay of the circuit exactly. To be able to provide a formal guarantee that the clock speed is met, however, it is still essential to account for effects such as cross-talk and to be able to model complex gates accurately. Therefore, for such circuits the use of timed automata based hierarchical methods is justified. In the next section, we propose such an approach. The key ingredients of this approach have been explored in this and earlier chapters. The implementation of an efficient hierarchical delay computation framework is a major undertaking and is beyond the scope of this dissertation.

7.5 Delay Computation on a Hierarchy

7.5.1 A Hierarchical View of Combinational Circuits

We propose the following hierarchical representation for combinational circuits: Each leaf-level component in the hierarchy is a connected (or coupled) group of gates, transistors and wires. Higher level modules consist of interconnections of modules from lower levels. The functional and timing characteristics of each leaf level model is represented by timed automata as described in Section 7.2. Automata representing higher level modules are obtained from lower level ones by composition and hiding of variables corresponding to internal nodes. To keep the automata of higher level modules small, abstraction is used, in which case, the high level automaton over approximates the behavior of the interconnection of lower level models. By controlling the amount of abstraction used, one can trade-off accuracy and efficiency of analysis. Previous research on performing minimization of products of timed automata [DY96, LPY95] reports encouraging results from algorithms related to this problem.

Example 7.5.1 Figure 7.13 illustrates the hierarchical representation described above. The circuit consists of four identical `NOR` gates, each of which is modeled by an ideal `NOR` gate followed by an inertial delay buffer depicted in Figure 7.14.

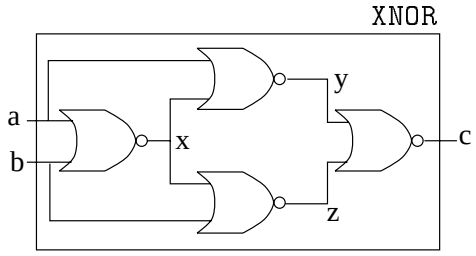

Figure 7.13: A hierarchical view of a combinational circuit block.

The four NOR gates constitute the leaf-level modules in the hierarchy. Composing these four models we obtain a timed automaton that represents the behavior of the nodes a, b, c, x, y, and z. This automaton has $4^4 = 256$ locations and uses 4 clock variables. x, y and z are internal to the XNOR gate and are not observable from outside. Thus, if the XNOR gate is part of a larger circuit, the representation for this gate need not include information about x, y and z.

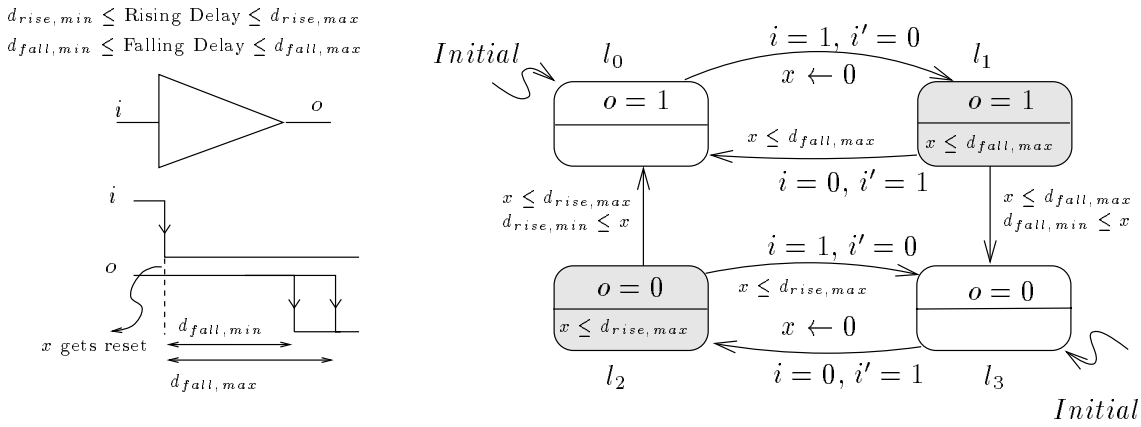

Figure 7.14: An inertial delay buffer with rising delay in the range $[d_{rise,min}, d_{rise,max}]$ and falling delay in the range $[d_{fall,min}, d_{fall,max}]$.

We smooth these variables to obtain a timed automaton model for the module XNOR. This timed automaton specifies exactly what waveforms are possible at c for each pair of input waveforms at a and b. Since no abstraction has been performed at this point, there is no reduction in the number of locations and clocks. If we insist on an exact model at a higher level of the hierarchy, often little or no further reduction is possible in the number of clocks and locations. Suppose, however, that for the delay computation problem we are interested in, modeling XNOR at this level of detail is unnecessary and that it is sufficient to

specify the earliest possible time c becomes unstable and the latest possible time it takes on a stable value. The behavior of c in the unstable intervals does not need to be specified exactly. Then we can model XNOR by the automaton shown in Figure 7.15. The number of locations and clocks has been reduced significantly at the expense of allowing more behavior at c than is possible in the original circuit. By picking d_{min}, d_{max} and the look-up table f properly, we can ensure that the higher level model is conservative but is accurate enough for our application. ■

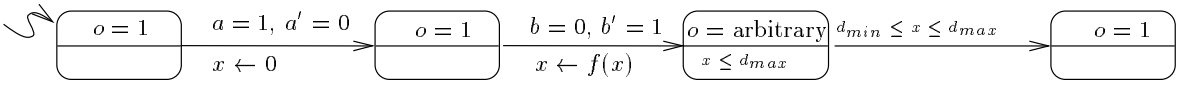

Figure 7.15: Part of the automaton representing the higher level model for XNOR. The reset function f models the dependency of delay on the relative arrival times of a and b. The rest of the automaton is constructed similarly using a total of two clocks.

7.5.2 Image Computation on a Hierarchy

Recall the expression for the timed automaton F representing the evolution of the circuit over time: $F = (C_1 \parallel ... \parallel C_m)$. Observe that F represents the waveforms at the internal nodes and primary inputs of the circuit. To obtain a timed automaton 0 representing the waveforms at the primary outputs (and latch inputs) only, we must existentially quantify in F the variables representing the internal nodes ($\boldsymbol{in}$) and primary inputs ($\boldsymbol{pi}$):

$$0 = (\exists \vec{pi}, \vec{in})\ F = (\exists \vec{pi}, \vec{in})\ (C_1 \parallel ... \parallel C_m)$$

The computation method proposed is based on the following interpretation of the computation of F: Each circuit component transforms the set of waveforms at its inputs (represented by a timed automaton) to a set of waveforms at its output (again represented as a timed automaton), that is, it computes as its output the "image" of the set of its input waveforms. Starting with the primary input waveforms, and by propagating sets of waveforms across the circuit in topological order, we arrive at the set of resulting primary output waveforms, again represented by timed automata. Given a representation of this set, the maximum delay of the circuit can be deduced, as well as other useful information such as whether the set-up and hold times of latches are met.

The analogy with image computation across a Boolean network manifests itself in the equation for O. As is the case in taking the image of inputs across a Boolean network,

there is freedom in the order that the compositions ($\|$ operations) and the smoothing ($\exists$ operations) are performed. Heuristics for ordering these operations can keep intermediate results small. Otherwise, for instance, if the product of all the C_j's is taken first, much more intermediate information is represented than is necessary for calculating the maximum delay. The particular choice for the ordering of operations for computing O is formally expressed in the form of a "hierarchy tree"[6] as alluded to in Section 7.5.1. We will index the nodes of this tree by some prefix-closed set $\Gamma \subseteq \mathbb{N}^*$. A_0 is the root of the tree and corresponds to 0. If $n \in \Gamma$ is a leaf of Γ, then A_n is one of the C_i's. Each C_i occurs once and only once as a leaf. The children of a node A_n are given by $A_{n0}, A_{n1}, ..., A_{nk_n}$ and the A's are computed using the following recursive formulation:

$$A_n = (\exists \boldsymbol{\kappa}_n) \ \|_{l=1}^{k_n} A_{nl} \tag{7.5}$$

Here $\boldsymbol{\kappa}_n$ denotes the set of all variables that are internal to A_n

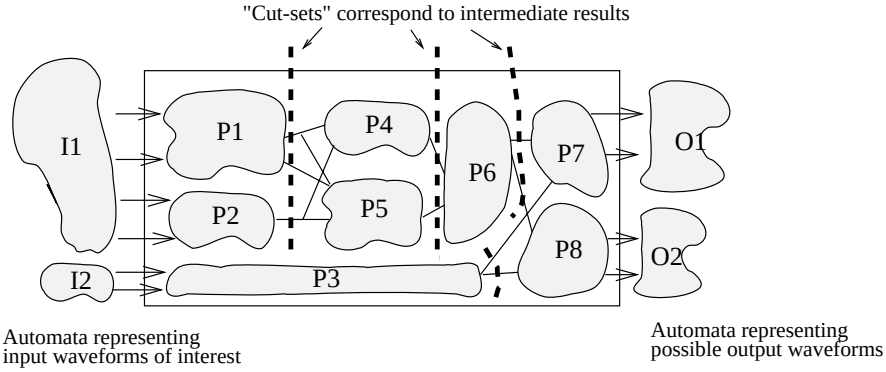

Figure 7.16: Waveform propagation across partitions.

The practical success of this approach depends most critically on the partitioning of the circuit into the hierarchy tree. Hierarchical verification works well when the high level models encapsulate just the right amount of information needed to carry out the analysis. Without proper partitioning, high level models may have to represent information about many intermediate signals, which would make analysis costly. We propose the following heuristics:

- Image computation should be performed by "propagating wavefronts" across the partitions. This corresponds to performing the composition of the C_i's in topological

[6] This hierarchy need not correspond to the design hierarchy.

(i) Take the product of automata and hide intermediate variables.

(ii) Perform "untimed reachability analysis" on the product automaton. Delete locations that are unreachable.

(iii) Apply the clock minimization algorithm of [DY96] to the reachable part of the product.

Figure 7.17: Algorithm COMPOSE-AND-SMOOTH

order. To achieve this, whenever one child of a node in the tree lies in the transitive fan-out of another, it should in fact be in its immediate fan-out, i.e., they should be contiguous in the circuit topology. Also, if a partition P precedes Q in the topological order of the circuit, then P should be placed to the left of Q in the hierarchy tree. As depicted in Figure 7.16, if the set of waveforms at a given cut-set in the topological order has been characterized, all variables to the left of the cut-set can be smoothed. Heuristically, fewer variables in the support of an automaton result in a smaller representation for the automaton.

- Whenever possible, partitions should have disjoint support. For such partitions, only the set of possible waveforms at the output nodes need to be stored, the input variables can be existentially quantified (see partitions P1 and P2 in Figure 7.16 for instance).

- Long and narrow partitions (such as P3 in Figure 7.16) are desirable, because they can essentially be treated separately, they have few input and output variables in their support, and the cut-sets that such partitions give rise to are small.

- The sizes of the partitions can be determined in a way similar to the selection of intermediate points in implicit image computation. When timed automata for intermediate results get too large, it is time to smooth out some variables and perform some abstraction.

We refer to the computation on the right hand side of Equation 7.5 as the "Compose-Smooth" operation. This operation needs to be implemented efficiently and must compute a small representation for its output. We propose the algorithm in Figure 7.17. Note that this algorithm avoids computation on the "timed state space" involving sets of clock valuations. Computations involving this space often have exponential complexities, whereas algorithms that operate on a syntactic description of the timed automaton have their complexities expressed in terms of the number of locations. Therefore, even if the latter have high

degree polynomial complexities, they are desirable if they can achieve reductions in the size of the timed state space. We now elaborate on steps (ii) and (iii).

(ii) The "untimed reachability analysis" performs reachability analysis on the $Untime(.)$'d (see Section 2.5.3) automaton and may declare locations reachable whereas analysis using timing information would have concluded that they are not. Untimed reachability analysis provides an overestimate of the set of reachable locations that can be computed quickly. When analyzing combinational circuits, many locations in the product automata encountered are unreachable because of the intended Boolean behavior of the circuit.

(iii) [DY96] presents an algorithm that minimizes the number of clock variables used by a timed automaton. It operates on a textual description of a timed automaton and outputs a timed automaton that is language equivalent to the input automaton.

Existentially quantifying the variables simplifies the input predicates on the edges of the automaton. Since all internal variables are quantified out by COMPOSE-AND-SMOOTH, many edges will have "*True*" as the edge predicate. It may be possible to further minimize the number of locations and clocks of this automaton while leaving its language unchanged using, for instance, bisimulation equivalence [KS90, LY92, ACH+92]

However, as remarked in the previous section, if the input-output behavior of each A_n is represented exactly, the minimization algorithms referred to above may not be able to reduce the resultant automaton to a manageable size. In general, to obtain a simpler representation for A_n, the right hand side of Equation 7.5 needs to be overapproximated. The form of this overapproximation is technology and application dependent. The investigation of abstraction templates for the delay analysis of deep sub-micron circuits is left as a topic for future research.

An important feature of the hierarchical approach described in this section is the fact that it provides many degrees of freedom for a trade-off between efficiency and accuracy. For instance, to obtain simpler representations, waveform sets can be over-approximated. Suppose that an automaton A represents exactly the sets of waveforms at two circuit nodes, x and y and that A is too big. This automaton can be replaced by $(\exists x)A \parallel (\exists y)A$ to obtain a conservative analysis. This amounts to ignoring the correlation between x and y: if α is a possible waveform at x and β is a possible waveform at y, then the latter automaton

states that (α, β) is a possible waveform at (x, y) while this may be impossible in A. The representation for $(\exists x)A \parallel (\exists y)A$ is likely to be smaller than A. Further, it is possible to partition such intermediate results when they become too large. The sets of waveforms at a cut-set can be divided into separate sets represented by separate automata and image computation can be performed on each of them individually. The underlying formalism of timed automata ensures that while we make such modifications to our algorithms to gain efficiency, the analysis remains conservative.

7.6 Conclusions

In this chapter we investigated timed-automaton based methods for delay computation of combinational circuits. We addressed issues that arise due to decreasing minimum feature sizes of VLSI circuits. The performance of our algorithms show a significant advance in the applicability of timed automaton-based methods, and make them a more viable option than exhaustive circuit simulation when accuracy and full coverage are desired. To be able to handle large practical circuits we proposed the use of hierarchical modeling and analysis, and heuristics that mimic image computation on Boolean networks.

An important benefit of the approach presented is the decoupling of modeling issues from the algorithmic issues of analysis and verification by employing timed automata as a clean interface between the two. In this way, the algorithms developed remain applicable for different delay models and at different levels of the hierarchy. Moreover, more efficient techniques developed for analyzing timed automata can be incorporated immediately. In addition to computing delay, other timing aspects of a circuit, such as the absence of spikes and short-circuited paths from the power supply to the ground node, can be verified.

A distinct conceptual advantage of this framework is the fact that it provides a precise formulation for delay computation, with delay models that can be made as accurate as desired. The exact solution can then be approximated in a formally correct way. In contrast to other delay models and delay computation methods, our approach benefits from a uniform representation of delay models that can be used at all levels of the hierarchy. Common current industrial practice is to exploit hierarchy in an ad hoc way by using look-up tables for delay simulation. Our method is distinguished by the facts that it enables coverage of all possible inputs, and that it provides a formal guarantee that the hierarchical models are conservative abstractions of lower level models.

Chapter 8

Concluding Remarks

Validation of the correct timing and functionality has become the most costly and demanding portion of the design of an electronic system. The conceptual appeal of algorithmic formal verification as a means for validation is largely outweighed by its prohibitive computational complexity, around which most formal verification research centers. For real-time systems, there are two sources of this complexity. The first one is associated with the component of the state-spaces that represents timing information. As a rule of thumb, verification algorithms are exponential in the number of timing constraints as well as the sizes of numbers used to express the timing constraints. The second source of complexity is the exponential dependency of the state-space size in the number of state variables, or, put differently, the textual description of the system. The goal of the techniques presented in this dissertation has been to make formal verification a viable means for validation by the countering two sources of exponential complexity. With the exception of Chapter 3, the focus has been on the second, "compositional" kind.

We have built our modeling formalism around a variant of the timed automaton model which facilitates compositional arguments. Timed automata have become the canonical representation for real-time systems. Several verification tools for real-time systems have been based on timed automata, including the algorithms presented in this dissertation which have been implemented in COSPAN and MOCHA.

In Chapter 4 we studied abstraction relations for real-time systems. We proved that it is decidable to check whether one system timed simulates another. Our algorithm was EXPTIME, so we opted for an algorithm for checking homomorphisms that preserve timed behavior. This algorithm, implemented in COSPAN was our workhorse for verifying abstrac-

tions of real-time systems.

When modeled at the implementation level in their entirety, the state-spaces of all but the smallest systems is beyond the capacities of algorithms based on state-space traversal. With this conviction, Chapter 5 was dedicated to the study of modularity in verifying abstractions of real-time systems.

The STARI chip, verified in Chapter 6, demonstrated that the techniques of Chapters 4 and 5 can make the verification of real designs viable. It was also our conclusion that hierarchical techniques, particularly assume-guarantee style reasoning, are indispensable for carrying out modular proofs.

In Chapter 7, we tailored our techniques to the problem of computing the delays of combinational circuits accurately. This application domain is interesting not only because it is a critical problem in the design cycle but also because here functionality and timing are closely intertwined. This study benefited a lot from the expressiveness as well as the formal power of the timed automaton framework.

Despite the intensive research that has taken place in the recent years, the gap between the available formal verification technology and the requirements of designers still remains wide. The research problems that are immediate extensions of our work have been pointed out in the related chapters. Below we list[1] more general research directions which we believe will be important for providing more convenient and powerful formal validation tools.

8.1 Making Refinement Checking More Efficient

A variety of techniques exist for making state-space traversal more efficient, partial-order reductions [GW90, Rok93, BM98], partitioned transition relations ([HKB94] for instance), over- and under-approximation techniques [DW95, HHM+96], iterative techniques (Chapter 3, [WPJ+96, Bal95b]), partitioned representations (Chapter 7, [McM96]) to name a few. In the implementation of the refinement checking algorithm of Section 4.3, we have made use of the implicit OBDD representations for sets and computing reachability, but have not made an further effort to optimize the performance beyond the capabilities offered by standard OBDD packages. In principle, all of the aforementioned optimizations can be incorporated in some form into refinement checking algorithms. The improvement brought about by a combination of these techniques can be significant [ABH+97] and may pro-

[1] in decreasing order of direct relevance to the work presented in this dissertation

vide sufficient complexity reduction to enable refinement checking on certain large systems. Data structures also factor significantly into the efficiency of timing verification algorithms. While MDDs and OBDDs provide a much needed improvement over explicit representations for representing the control component of the state space, they are not well suited for representing sets of clock valuations. Difference bound matrices, on the other hand, are efficient for representing sets of clock valuations but inefficient for representing the control component. An efficient data structure for representing Boolean and timing constraints uniformly is a topic of active research [LB94b, Bal95a, ABK+97].

As was demonstrated in Chapter 4, the definition of the implementation preorder determines the computational complexity of checking it. For certain applications, complete preservation of timed behavior may not be required, or may not necessarily be the correct definition of refinement. For instance, a clock cycle accurate refinement of a high-level specification for a microprocessor is often not desired. In such cases, our definition of correct refinement is too restrictive. Depending on the application domain, it may make sense to define timed refinement differently and, possibly, more loosely. This may make it easier to check refinement and to specify abstraction mappings.

8.2 Reducing the User Effort in Hierarchical Verification

A top-down, modular design methodology facilitates the hierarchical approach to formal verification. However, many designs are not carried out in this fashion. If this is the case, the user must first come up with an abstract model, verify that the abstraction is correct, and then make sure that the abstract model still contains enough information to prove the properties of interest. This is an iterative, time-consuming process. One way to alleviate this difficulty is to check refinement automatically without requiring an abstraction map, as was the case with the homomorphisms of Section 4.3. Chapter 4 demonstrated that completely automatic algorithms can be very complex. Another way to reduce the user effort is to provide computer support for the iterations by incorporating "abstraction templates" which have been found to be useful in a particular domain. Ideally, such templates capture the designer's high level view of the system. Iterative techniques (in the style of Chapter 3) for checking refinement can also be valuable especially in constructing environment abstractions while checking refinement on a per module basis.

Part of the difficulty in performing hierarchical verification with timed automata stems

from the fact that timed automata are a rather low level representation. Translations of system descriptions and abstraction mappings from high level languages to timed automata may make the refinement checking process easier for the user.

8.3 Multiple Abstract Views of One System

It is both natural and more efficient to use separate abstract models to represent different aspects of a system, such as timing and logical functionality, or data-flow and control. These abstract models necessarily make assumptions about each other. We plan to investigate assume-guarantee style verification techniques which enable the use of multiple abstract views. Such techniques are especially needed for systems with both hardware and software components and for design styles mixing synchronous and asynchronous circuits. For these kinds of systems, a complete monolithic representation for the whole system may not exist or may be impractical to build. The capability to work with several abstract representations, which may involve different mathematical models of computation, is essential in these cases.

8.4 Integration of Formal Verification and Simulation

Abstractions and hierarchical verification are not currently part of the design and validation process in the electronics industry. Verification tools supporting such methodologies are fairly recent, and are not robust enough to be used on the critical path of a design. In fact, to the best of our knowledge, no commercially available verification tool supports refinement checks and modular verification.

Furthermore, even with proper tool support, the nature of a design may make it difficult to use the kinds of techniques investigated in this dissertation. While a designer is implementing her portion of the design, the specifications of the neighboring portions may not be well-defined, may change or the designer may simply not have this knowledge[2]. Also, what is considered as one "component" of a large design may still be too large for verification tools, which may require further decomposition. This may be very time consuming and error prone. These, and many other reasons make software simulation an indispensable part of the validation process. Simulation is also necessary as a sanity check for formal

[2]One promising way to address this issue is "interface-based design" [RS97]

methodologies. However, much of the resources spent on simulation are wasted because of the lack of a formal approach, as discussed in Chapter 1. We believe that software simulation integrated with formal verification techniques (as suggested in [Dil98, FDK98]) and implicit representations will be an important topic of research because of the pressing need for formal but still computationally viable techniques.

Bibliography

[ABB+94] A. Aziz, F. Balarin, R. K. Brayton, S.-T. Cheng, R. Hojati, T. Kam, S. C. Krishnan, R. K. Ranjan, A. L. Sangiovanni-Vincentelli, T. R. Shiple, V. Singhal, S. Taşıran, and H.-Y. Wang. HSIS: A BDD-Based Environment for Formal Verification. In *Proc. of the Design Automation Conf.* 1994.

[ABH+97] R. Alur, R.K. Brayton, T.A. Henzinger, S. Qadeer, and S. Rajamani. Partial order reduction in symbolic state space exploration. In O. Grumberg, editor, *CAV 97: Computer Aided Verification*, Lecture Notes in Computer Science 1254. Springer-Verlag, 1997.

[ABK+97] E. Asarin, M. Bozga, A. Kerbrat, O. Maler, A. Pnueli, and A. Rasse. Data structures for the verification of timed automata. In O. Maler, editor, *HART 97: Hybrid and Real-time Systems*, Lecture Notes in Computer Science 1201, pages 346–360. Springer-Verlag, 1997.

[ACH+92] R. Alur, C. Courcoubetis, N. Halbwachs, D.L. Dill, and H. Wong-Toi. Minimization of timed transition systems. In R.J. Cleaveland, editor, *CONCUR 92: Theories of Concurrency*, Lecture Notes in Computer Science 630, pages 340–354. Springer-Verlag, 1992.

[AD90] R. Alur and D.L. Dill. Automata for modeling real-time systems. In M.S. Paterson, editor, *ICALP 90: Automata, Languages, and Programming*, Lecture Notes in Computer Science 443, pages 322–335. Springer-Verlag, 1990.

[AD94] R. Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[AH96] R. Alur and T.A. Henzinger. Reactive modules. In *Proceedings of the 11th Annual Symposium on Logic in Computer Science*, pages 207–218. IEEE Computer Society Press, 1996.

[AHM+98] R. Alur, T.A. Henzinger, F.Y.C. Mang, S. Qadeer, S.K. Rajamani, and S. Taşıran. MOCHA : Modularity in model checking. In A. Hu and M. Vardi, editors, *CAV 98: Computer Aided Verification*, Lecture Notes in Computer Science. Springer-Verlag, 1998.

[AIKY92] R. Alur, A. Itai, R.P. Kurshan, and M. Yannakakis. Timing verification by successive approximation. In G. von Bochmann and D.K. Probst, editors, *CAV 92: Computer Aided Verification*, Lecture Notes in Computer Science 663, pages 137–150. Springer-Verlag, 1992.

[AK96] R. Alur and R.P. Kurshan. Timing analysis in COSPAN. In R. Alur, T.A. Henzinger, and E.D. Sontag, editors, *Hybrid Systems III*, Lecture Notes in Computer Science 1066, pages 220–231. Springer-Verlag, 1996.

[AL88] M. Abadi and L. Lamport. The existence of refinement mappings. In *Proceedings of the Third Annual Symposium on Logic in Computer Science*, pages 165–175. IEEE Computer Society Press, 1988.

[AL91] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

[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.

[Alu91] R. Alur. *Techniques for Automatic Verification of Real-time Systems.* PhD thesis, Stanford University, 1991.

[ATB93] A. Aziz, S. Taşıran, and R. K. Brayton. A Communication Complexity Based Approach to BDD Variable Ordering for systems of Interacting Finite State Machines. In *Workshop Notes of the Intl. Workshop on Logic Synthesis*, Tahoe City, CA, May 1993.

[ATB94] A. Aziz, S. Taşıran, and R. K. Brayton. BDD Variable Ordering for Interacting Finite State Machines. In *Proc. of the Design Automation Conf.*, June 1994.

[Bal95a] F. Balarin. Approximate reachability analysis of timed automata. In *1995 ACM/IEEE Intl. Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, pages 270–281. 1995.

[Bal95b] F. Balarin. *Iterative Methods for Formal Verification of Digital Systems*. PhD thesis, University of California at Berkeley, 1995.

[BCH+91] R.K. Brayton, M. Chiodo, R. Hojati, T. Kam, K. Kodandapani, R.P. Kurshan, S. Malik, A.L. Sangiovanni-Vincentelli, E.M. Sentovich, T.S. Shiple, K.J. Singh, and H.-Y. Wang. BLIF-MV: An interchange format for design verification and synthesis. Technical Report UCB/ERL Memo M91/97, University of California, Berkeley, 1991.

[BCM+92] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[BHSV+96] R.K. Brayton, G.D. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S.-T. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R.K. Ranjan, S. Sarwary, T.R. Shiple, G. Swamy, and T. Villa. VIS: A System for Verification and Synthesis. In R. Alur and T.A. Henzinger, editors, *CAV 96: Computer Aided Verification*, Lecture Notes in Computer Science 1102, pages 428–432. Springer-Verlag, 1996.

[BLL+96] J. Bengtsson, K.G. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL: a tool-suite for automatic verification of real-time systems. In R. Alur, T.A. Henzinger, and E.D. Sontag, editors, *Hybrid Systems III*, Lecture Notes in Computer Science 1066, pages 232–243. Springer-Verlag, 1996.

[BM98] W. Belluomini and C. J. Myers. Verification of timed systems using POSETS. In A. Hu and M. Vardi, editors, *CAV 98: Computer Aided Verification*, Lecture Notes in Computer Science 1427, pages 403–415. Springer-Verlag, 1998.

[BMP81] M. Ben-Ari, Z. Manna, and A. Pnueli. The temporal logic of branching time. In *Proceedings of the Eighth Annual Symposium on Principles of Programming Languages*, pages 164–176. ACM Press, 1981.

[Bry86] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[BSV95] Felice Balarin and Alberto L. Sangiovanni-Vincentelli. An iterative approach to verification of real-time systems. *Formal Methods in System Design: An International Journal*, January 1995.

[Čerāns92] K. Čerāns. Decidability of bisimulation equivalence for parallel timer processes. In G. von Bochmann and D.K. Probst, editors, *CAV 92: Computer Aided Verification*, Lecture Notes in Computer Science 663, pages 302–315. Springer-Verlag, 1992.

[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.

[CGH+93] E.M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D.E. Long, K.L. McMillan, and L.A. Ness. Verification of the Futurebus+ cache coherence protocol. In *Proceedings of the 11th Symposium on Computer Hardware Description Languages and their Applications*, 1993.

[ČGL93] K. Čerāns, J.C. Godskesen, and K.G. Larsen. Timed modal specification: Theory and tools. In C. Courcoubetis, editor, *CAV 93: Computer Aided Verification*, Lecture Notes in Computer Science 697, pages 253–267. Springer-Verlag, 1993.

[CLM89] E.M. Clarke, D.E. Long, and K.L. McMillan. Compositional model checking. In *Proceedings of the 4th Annual Symposium on Logic in Computer Science*, pages 353–362. IEEE Computer Society Press, 1989.

[CR83] J.E. Coolahan and N. Roussopoulos. Timing requirements for time-driven systems using augmented petri nets. *IEEE Transactions on Software Engineering*, SE-9(9):603–616, 1983.

[CWS97] V. Chandramouli, J. Whittemore, and K. Sakallah. AFTA: A delay model for functional timing analysis. In *1997 ACM/IEEE Intl. Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, pages 5–14. 1997.

[DH93] D.L. Dill and A.J. Hu. Efficient verification with BDDs using implicitly conjoined invariants. In C. Courcoubetis, editor, *CAV 93: Computer Aided Verification*, Lecture Notes in Computer Science 697, pages 3–14. Springer-Verlag, 1993.

[DHWT91] D. Dill, A. Hu, and H. Wong-Toi. Checking for language inclusion using simulation relations. In K.G. Larsen and A. Skou, editors, *CAV 91: Computer Aided Verification*, Lecture Notes in Computer Science 575. Springer-Verlag, 1991.

[Dil98] D. Dill. What's between simulation and formal verification? In *Proc. of the Design Automation Conf.*, June 1998.

[DKM93] S. Devadas, K. Keutzer, and S. Malik. Computation of floating mode delay in combinational circuits: Theory and algorithms. *IEEE Transactions on Computer-Aided Design*, 12(12):1913–1923, 1993.

[DOTY96] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In R. Alur, T.A. Henzinger, and E.D. Sontag, editors, *Hybrid Systems III*, Lecture Notes in Computer Science 1066, pages 208–219. Springer-Verlag, 1996.

[DOY94] C. Daws, , A. Olivero, and S. Yovine. Verifying ET-LOTOS programs with KRONOS. In *Formal Description Techniques VII, Proceedings of FORTE'94*, pages 227–242, 1994.

[DW95] D.L. Dill and H. Wong-Toi. Verification of real-time systems by successive over- and underapproximation. In P. Wolper, editor, *CAV 95: Computer Aided Verification*, Lecture Notes in Computer Science 939, pages 409–422. Springer-Verlag, 1995.

[DY96] C. Daws and S. Yovine. Reducing the number of clock variables of timed automata. In *Proceedings of the 17th Annual Real-time Systems Symposium (RTSS '96)*, pages 66–75. IEEE Computer Society Press, 1996.

[EC82] E.A. Emerson and E.M. Clarke. Using branching-time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2(3):241–266, 1982.

[Eme90] E.A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 995–1072. Elsevier Science Publishers, 1990.

[Eme92] E.A. Emerson. Logics and models of real time: a survey. In J.W. de Bakker, K. Huizing, W.-P. de Roever, and G. Rozenberg, editors, *Real Time: Theory in Practice*, Lecture Notes in Computer Science 600, pages 74–106. Springer-Verlag, 1992.

[FDK98] F. Fallah, S. Devadas, and K. Keutzer. Occom: Efficient computation of observability-based code coverage metrics for functional verification. In *Proc. of the Design Automation Conf.*, June 1998.

[FK94] J. Frößl and T. Kropf. A new model to uniformly represent the function and timing of MOS circuits and its application to VHDL simulation. In *Proceedings of the European Design and Test Conference (EDTC)*, pages 343–348, 1994.

[FK95] J. Frößl and T. Kropf. Verifying real-time properties of MOS-transistor circuits. In *Proceedings of the European Design and Test Conference (EDTC)*, pages 314–319, 1995.

[GL94] O. Grumberg and D.E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, 1994.

[Gla95] A. Glaser. S/R reference manual. Technical report, AT&T Bell Laboratories, 1995.

[GPV94] A. Gollu, A. Puri, and P. Varaiya. Discretization of timed automata. In *Proceedings of the 33rd IEEE Conference on Decision and Control*, pages 957–958. ACM Press, 1994.

[Gre] M.R. Greenstreet. STARI: Skew tolerant communication. *(To appear) IEEE Transactions on Computers.*

[Gre93] M.R. Greenstreet. *STARI: A Technique for High-Bandwidth Communication.* PhD thesis, Princeton University, 1993.

[GSSAL94] R. Gawlick, R. Segala, J.F. Sogaard-Andersen, and N.A. Lynch. Liveness in timed and untimed systems. In S. Abiteboul and E. Shamir, editors, *ICALP 94: Automata, Languages, and Programming*, Lecture Notes in Computer Science 820, pages 166–177. Springer-Verlag, 1994.

[Gup91] A. Gupta. Formal hardware verification methods: a survey. *Formal Methods in System Design*, 1(2/3):151–238, 1991.

[GW90] P. Godefroid and P. Wolper. Using partial orders to improve automatic verification methods. In R.P. Kurshan and E.M. Clarke, editors, *CAV 90: Computer Aided Verification*, Lecture Notes in Computer Science 531. Springer-Verlag, 1990.

[GW91] P. Godefroid and P. Wolper. A partial approach to model checking. In *Proceedings of the Sixth Annual Symposium on Logic in Computer Science*, pages 406–415. IEEE Computer Society Press, 1991.

[HBAB93] H. Hulgaard, S. M. Burns, T. Amon, and G. Borriello. Practical applications of an efficient time separation of events algorithm. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 322–325, November 1993.

[HHK95] M.R. Henzinger, T.A. Henzinger, and P.W. Kopke. Computing simulations on finite and infinite graphs. In *Proceedings of the 36th IEEE Symposium on Foundations of Computer Science*, pages 453–462, 1995.

[HHK96] R.H. Hardin, Z. Har'El, and R.P. Kurshan. COSPAN. In R. Alur and T.A. Henzinger, editors, *CAV 96: Computer Aided Verification*, Lecture Notes in Computer Science 1102, pages 423–427. Springer-Verlag, 1996.

[HHM+96] C. Hyunwoo, G.D. Hachtel, E. Macii, B. Plessier, and F. Somenzi. Algorithms for approximate fsm traversal based on state space decomposition. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(12):1465–1478, 1996.

[HKB94] R. Hojati, S. Krishnan, and R.K. Brayton. Heuristic algorithms for early quantification and partial product minimization. Technical Report UCB/ERL Memo M94/11, University of California, Berkeley, 1994.

[HMP92] T.A. Henzinger, Z. Manna, and A. Pnueli. What good are digital clocks? In *Proceedings of the 19th International Colloquium on Automata, Languages, and Programming (ICALP 1992)*, Lecture Notes in Computer Science 623, pages 545–558. Springer-Verlag, 1992.

[Hoa78] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[HQRT98] T. Henzinger, S. Qadeer, S. Rajamani, and S. Taşıran. An assume-guarantee rule for checking simulation. In *FMCAD 98*, Lecture Notes in Computer Science. Springer-Verlag, 1998.

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

[KB91] T. Kam and R. K. Brayton. Multi-valued decision diagrams. Technical Report UCB/ERL Memo M90/125, University of California, Berkeley, 1991.

[KR97] T. Kropf and J. Ruf. Using MTBDDs for discrete timed symbolic model checking. In *Proceedings of the European Design and Test Conference (EDTC)*, pages 182–187. IEEE Comput. Soc. Press, 1997.

[KS90] P.C. Kanellakis and S.A. Smolka. CCS expressions, finite-state processes, and three problems of equivalence. *Information and Computation*, 86:43–68, 1990.

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

[LA90] N.A. Lynch and H. Attiya. Using mappings to prove timing properties. In *Proceedings of the Ninth Annual Symposium on Principles of Distributed Computing*, pages 265–280. ACM Press, 1990.

[LA92] N.A. Lynch and H. Attiya. Using mappings to prove timing properties. *Distributed Computing*, 6:121–139, 1992.

[LB93] W. K. C. Lam and R. K. Brayton. Alternating RQ timed automata. In C. Courcoubetis, editor, *CAV 93: Computer Aided Verification*, Lecture Notes in Computer Science 697, pages 237–252. 1993.

[LB94a] W. K. C. Lam and R. K. Brayton. Criteria for the simple path property in timed automata. In D.L. Dill, editor, *CAV 94: Computer Aided Verification*, Lecture Notes in Computer Science 818, pages 237–252. Springer-Verlag, 1994.

[LB94b] W.K.C. Lam and Robert K. Brayton. *Timed Boolean Functions- A Unified Formalism for Exact Timing Analysis*. Kluwer Academic Publishers, 1994.

[LG95] C. Leung and M. Greenstreet. A simple proof checker for timing verification. In *ACM Intl. Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, pages 294–305. November 1995.

[LP85] O. Lichtenstein and A. Pnueli. Checking that finite-state concurrent programs satisfy their linear specification. In *Proceedings of the 12th Annual Symposium on Principles of Programming Languages*, pages 97–107. ACM Press, 1985.

[LPY95] K.G. Larsen, P. Pettersson, and W. Yi. Compositional and symbolic model checking of real-time systems. In *Proceedings of the 16th Annual Real-time Systems Symposium*, pages 76–87. IEEE Computer Society Press, 1995.

[LT87] N.A. Lynch and M.R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the 6th Annual Symposium on Principles of Distributed Computing*, pages 137–151. ACM Press, 1987.

[LY92] D. Lee and M. Yannakakis. Online minimization of transition systems. In *Proceedings of the 24th Annual Symposium on Theory of Computing*, pages 264–274. ACM Press, 1992.

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

[McM93] K.L. McMillan. *Symbolic Model Checking: An Approach to the State-Explosion Problem*. Kluwer Academic Publishers, 1993.

[McM96] K.L. McMillan. A conjunctively decomposed boolean representation for symbolic model checking. In R. Alur and T.A. Henzinger, editors, *CAV 96: Com-*

puter Aided Verification, Lecture Notes in Computer Science 1102, pages 13–25. Springer-Verlag, 1996.

[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.

[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.

[Mil80] R. Milner. *A Calculus of Communicating Systems*. Lecture Notes in Computer Science 92. Springer-Verlag, 1980.

[MP95] O. Maler and A. Pnueli. Timing analysis of asynchronous circuits using timed automata. In *ACM Intl. Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, pages 249–257. November 1995.

[MSBSV93] P. McGeer, A. Saldanha, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Delay models and exact timing analysis. In T. Sasao, editor, *Logic Synthesis and Optimization*, pages 167–189. Kluwer Academic Publishers, 1993.

[Nag95] L. Nagel. SPICE: A computer program to simulate computer circuits. Technical Report UCB/ERL Memo M520, University of California, Berkeley, 1995.

[NTR97] NTRS97. *The National Technology Roadmap for Semiconductors* `http://notes.sematech.org/97melec.html`. 1997.

[OL82] S. Owicki and L. Lamport. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems*, 4(3):455–495, 1982.

[Ost90] J.S. Ostroff. *Temporal Logic of Real-time Systems*. Research Studies Press, 1990.

[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.

[Pro] SPACE Project. *Physical Design Modeling and Verification Project:* `http://cas.et.tudelft.nl/research/space.html`.

[RA93] M.Y. Vardi R. Alur, T.A. Henzinger. Parametric real-time reasoning. In *Proceedings of the 25th ACM Symposium on Theory of Computing*, pages 592–601. IEEE Computer Society Press, 1993.

[Ram74] C. Ramchandani. Analysis of asynchronous concurrent systems by petri nets. Technical Report MAC TR-120, Massachussetts Institute of Technology, 1974.

[Rei85] W. Reisig. *Petri Nets: An Introduction.* Springer-Verlag, 1985.

[Rok93] T. Rokicki. *Representing and Modeling Digital Circuits.* PhD thesis, Stanford University, 1993.

[RS97] J.A. Rowson and A.L. Sangiovanni-Vincentelli. Interface-based design. In *Proc. of the Design Automation Conf.*, pages 178–183, 1997.

[RW87] P.J. Ramadge and W.M. Wonham. Supervisory control of a class of discrete-event processes. *SIAM Journal of Control and Optimization*, 25(1):206–230, 1987.

[RW89] P.J. Ramadge and W.M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81–98, 1989.

[Sha92] A.U. Shankar. A simple assertional proof system for real-time systems. In *Proceedings of the 13th IEEE Real-Time Systems Symposium*, pages 167–176, 1992.

[SSL+92] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Sis: A system for sequential circuit synthesis. Technical Report UCB/ERL M92/41, University of California, Berkeley, 1992.

[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.

[TAKB96] S. Taşıran, R. Alur, R.P. Kurshan, and R.K. Brayton. Verifying abstractions of timed systems. In U. Montanari, editor, *CONCUR 96: Concurrency Theory*, Lecture Notes in Computer Science, pages 546–562. Springer-Verlag, 1996.

[Tas95] S. Tasıran. Language Containment Using Non-deterministic ω-automata. Master's thesis, University of California at Berkeley, Berkeley, California, 1995.

[TB95] S. Taşıran and R.K. Brayton. On iterative verification with timed automata. In *1995 ACM/IEEE Intl. Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, pages 258–269. 1995.

[TB97] S. Taşıran and R.K. Brayton. STARI: a case study in compositional and hierarchical timing verification. In O. Grumberg, editor, *CAV 97: Computer Aided Verification*, Lecture Notes in Computer Science 1254, pages 191–201. Springer-Verlag, 1997.

[THB95] S. Taşıran, R. Hojati, and R.K. Brayton. Language containment of non-deterministic ω-automata. In *Proceedings of the IFIP Working Conference on Correct Hardware Design and Verification Methods, CHARME*, 1995.

[Tho81] W. Thomas. A combinatorial approach to the theory of ω-automata. *Information and Control*, 48(3):261–283, 1981.

[TKB97] S. Taşıran, Y. Kukimoto, and R.K. Brayton. Computing delay with coupling using timed automata. In *1997 ACM/IEEE Intl. Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, pages 232–244. 1997.

[TKY+98] S. Taşıran, S. Khatri, S. Yovine, R.K. Brayton, and A. Sangiovanni-Vincentelli. A timed-automaton-based method for accurate computation of circuit delay in the presence of cross-talk. In *FMCAD 98*, Lecture Notes in Computer Science. Springer-Verlag, 1998.

[TSL+90] H. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Implicit State Enumeration of Finite State Machines using BDD's. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 130–133, November 1990.

[TY96] S. Tripakis and S. Yovine. Analysis of timed systems based on time-abstracting bisimulations. In *CAV 96: Computer Aided Verification*, Lecture Notes in Computer Science 1102, pages 232–243. Springer-Verlag, 1996.

[WPJ+96] L. Woohyuk, A. Pardo, J. Jae-Young, G. Hachtel, and F. Somenzi. Tearing based automatic abstraction for CTL model checking. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 76–81, November 1996.

[Wu94] E.C. Wu. Formal timing verification with simple-path timed automata. Master's thesis, University of California at Berkeley, Berkeley, California, 1994.

[Yal97] H. Yalcin. *Hierarchical Timing Analysis of Digital Circuits*. PhD thesis, University of Michigan,, 1997.

[YH95] H. Yalcin and J. P. Hayes. Hierarchical timing analysis using conditional delays. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 371–377, November 1995.

Appendix A

S/R Descriptions for STARI

A.1 A Buffer with Delay

```
proctype DB( _inp, _in_st : boolean; _lb, _ub : integer )
import _inp

selvar outp : (%, go_one, go_zero)
selvar # : boolean
stvar $ : (zero, one, zero_pending, one_pending, bad)

init $:= zero ? (_in_st=0) | one

asgn # := (($ = one) + ($ = zero_pending) * (outp=%) + (outp = go_one))

trans
zero {%}
      -> one_pending: (_inp = 1)
      -> $ : else;

one_pending {% {_lb, _ub} go_one}
      -> one: (outp=go_one) * (_inp = 1)
      -> $: else * (_inp = 1)
      -> bad: else;

one {%}
      -> zero_pending: (_inp = 0)
      -> $ : else;

zero_pending {% {_lb, _ub} go_zero}
      -> zero: (outp=go_zero) * (_inp = 0)
      -> $: else * (_inp = 0)
      -> bad: else;

bad {%}
      -> bad: true;

end DB( )
```

A.2 Ideal Muller C-element

```
proctype DELAYLESS_MULLER_C( _inp_1, _inp_2, _in_st1, _in_st2, _init_out: boolean)
import _inp_1, _inp_2

selvar # : boolean
stvar $ : ( i1_0_i2_0, i1_1_i2_1, i1_0_i2_1_o0, i1_0_i2_1_o1, i1_1_i2_0_o0, i1_1_i2_0_o1 )

init $:= i1_0_i2_0     ? ((_in_st1 = 0) * (_in_st2 = 0)) |
         i1_1_i2_1     ? ((_in_st1 = 1) * (_in_st2 = 1)) |
         i1_0_i2_1_o0 ? ((_in_st1 = 0) * (_in_st2 = 1) * (_init_out=0)) |
         i1_0_i2_1_o1 ? ((_in_st1 = 0) * (_in_st2 = 1) * (_init_out=1)) |
         i1_1_i2_0_o0 ? ((_in_st1 = 1) * (_in_st2 = 0) * (_init_out=0)) |
         i1_1_i2_0_o1 ? ((_in_st1 = 1) * (_in_st2 = 0) * (_init_out=1)) |
         i1_0_i2_0

asgn # := ( ((_inp_1 = 1) * (_inp_2 = 1)) +
            (($=i1_1_i2_1) + ($=i1_0_i2_1_o1) + ($=i1_1_i2_0_o1) ) *
            ((_inp_1 = 1) + (_inp_2 = 1)) )

trans
i1_0_i2_0
     -> i1_0_i2_1_o0: ((_inp_1=0) * (_inp_2=1))
     -> i1_1_i2_0_o0: ((_inp_1=1) * (_inp_2=0))
     -> i1_1_i2_1 : ((_inp_1=1) * (_inp_2=1))
     -> $ : else;

i1_1_i2_1
     -> i1_0_i2_1_o1: ((_inp_1=0) * (_inp_2=1))
     -> i1_1_i2_0_o1: ((_inp_1=1) * (_inp_2=0))
     -> i1_0_i2_0 : ((_inp_1=0) * (_inp_2=0))
     -> $: else;

i1_0_i2_1_o0
     -> i1_0_i2_0 : ((_inp_1=0) * (_inp_2=0))
     -> i1_1_i2_1 : ((_inp_1=1) * (_inp_2=1))
     -> i1_1_i2_0_o0: ((_inp_1=1) * (_inp_2=0))
     -> $: else;

i1_1_i2_0_o0
     -> i1_0_i2_0 : ((_inp_1=0) * (_inp_2=0))
     -> i1_1_i2_1 : ((_inp_1=1) * (_inp_2=1))
     -> i1_0_i2_1_o0: ((_inp_1=0) * (_inp_2=1))
     -> $: else;

i1_0_i2_1_o1
     -> i1_0_i2_0 : ((_inp_1=0) * (_inp_2=0))
     -> i1_1_i2_1 : ((_inp_1=1) * (_inp_2=1))
     -> i1_1_i2_0_o1: ((_inp_1=1) * (_inp_2=0))
     -> $: else;

i1_1_i2_0_o1
     -> i1_0_i2_0 : ((_inp_1=0) * (_inp_2=0))
     -> i1_1_i2_1 : ((_inp_1=1) * (_inp_2=1))
     -> i1_0_i2_1_o1: ((_inp_1=0) * (_inp_2=1))
     -> $: else;

end DELAYLESS_MULLER_C( )
```

A.3 The Gate level model F

proctype FIFO_STAGE(_inp_t, _inp_f, _ack_in: boolean;
$\delta_{c,min}$, $\delta_{c,max}$, $\delta_{nor,min}$, $\delta_{nor,max}$: integer;
_init_t, _init_f, _init_ack, _init_t_out, _init_f_out: boolean)

import _inp_t, _inp_f, _ack_in

selvar _ack_out, _out_t, _out_f : boolean

asgn _ack_out := NOR.#
asgn _out_t := TC.#
asgn _out_f := FC.#

proc TC: MULLER_C(_inp_t, _ack_in, _init_t, _init_ack, _init_t_out, $\delta_{c,min}$, $\delta_{c,max}$)
proc FC: MULLER_C(_inp_f, _ack_in, _init_f, _init_ack, _init_f_out, $\delta_{c,min}$, $\delta_{c,max}$)
proc NOR: DB(∼(_out_t + _out_f), 1, $\delta_{nor,min}$, $\delta_{nor,max}$)

end FIFO_STAGE()

A.4 The Transmitter and the Receiver

proc TRANSMITTER
import TR_CLOCK, ACKER /* ACKER is a dummy proc that provides the .ack signal */

selvar out_t, out_f: boolean
stvar $: (WAIT, SEND)

selvar out_data: (zero, one, empty)
stvar prv_data: (zero, one, empty)
selvar acked: boolean

init $:= SEND@, prv_data := empty

asgn out_data := zero, one, empty ? ($=SEND) * (TR_CLOCK.#=0) * acked * (prv_data=empty) |
empty ? ($=SEND) * (TR_CLOCK.#=0) * acked * ∼(prv_data=empty) |
prv_data

asgn acked := 1 ? ((prv_data = empty)*(ACKER.ack = 1) + (∼(prv_data = empty))*(ACKER.ack = 0)) |
0 ? ((prv_data = empty) * (ACKER.ack = 0) + (∼(prv_data = empty))*(ACKER.ack = 1)) |
0
asgn prv_data -> out_data
asgn out_t := (out_data = one)
asgn out_f := (out_data = zero)

trans
WAIT
-> SEND: (TR_CLOCK.#=1)
-> $: (TR_CLOCK.#=0);

```
SEND
    -> WAIT: (TR_CLOCK.#=0)
    -> $: else;

end TRANSMITTER

monitor REC
import SENDER, REC_CLOCK
selvar ack: boolean
selvar sampled_t, sampled_f: boolean
stvar old_sampled_t, old_sampled_f: boolean

init old_sampled_t := 0, old_sampled_f := 0

asgn sampled_t := SENDER._out_t ? (REC_CLOCK.# = go_zero) | old_sampled_t
asgn sampled_f := SENDER._out_f ? (REC_CLOCK.# = go_zero) | old_sampled_f

asgn old_sampled_t -> sampled_t
asgn old_sampled_f -> sampled_f
asgn ack := A.#

proc A: DB( ~(REC.sampled_t + REC.sampled_f), 1, δ_nor,min, δ_nor,max)

end REC
```

A.5 The Abstract Model A for One FIFO Stage

```
monitor FIFO_STAGE_ABS
import LEFT_ENV, RIGHT_ENV

selvar # : (%, go)
selvar inp_t, inp_f, ack_in: boolean
selvar out_t, out_f : boolean
selvar t_switch, f_switch : boolean
selvar ack_out, acked: boolean
selvar inputs_changed : boolean

stvar old_out_t, old_out_f, prev_ack_out : boolean
stvar $: (stable, wait_for_ack, wait_for_data, wait_out_chg, wait_ack_out)

init old_out_t := 0, old_out_f := 0, prev_ack_out := 1, $:= wait_for_data

asgn inp_t := LEFT_ENV.out_t
asgn inp_f := LEFT_ENV.out_f
asgn ack_in := RIGHT_ENV.ack
asgn inputs_changed := ( ~(inp_t=old_out_t) + ~(inp_f=old_out_f) )

asgn acked := (ack_in = 0) ? ((old_out_t = 1) + (old_out_f = 1)) |
              (ack_in = 1) ? ((old_out_t = 0) * (old_out_f = 0)) |
              0
```

```
asgn t_switch := acked*($=wait_out_chg)*(#=go)
asgn f_switch := acked*($=wait_out_chg)*(#=go)

asgn ack_out := ~(old_out_t + old_out_f) ? ($=wait_ack_out)*(#=go)
                                         | prev_ack_out
asgn prev_ack_out -> ack_out

asgn out_t := inp_t ? t_switch | old_out_t
asgn out_f := inp_f ? f_switch | old_out_f

asgn old_out_t -> out_t
asgn old_out_f -> out_f

trans
stable {%}
      -> wait_for_ack: ~acked * inputs_changed
      -> wait_out_chg: acked * inputs_changed
      -> wait_for_data: acked * ~inputs_changed
      -> $: else;

wait_for_ack {%}
      -> wait_out_chg: acked
      -> $: else;

wait_for_data {%}
      -> wait_out_chg: inputs_changed
      -> $: else;

wait_out_chg {% { δ_c,min , δ_c,max } go}
      -> wait_for_ack: ~acked
      -> wait_ack_out: #=go
      -> $: else;

wait_ack_out {% { δ_nor,min , δ_nor,max } go}
      -> stable          : (#=go) * ~acked * ~inputs_changed
      -> wait_for_ack    : (#=go) * ~acked * inputs_changed
      -> wait_out_chg    : (#=go) * acked * inputs_changed
      -> wait_for_data   : (#=go) * acked * ~inputs_changed
      -> $: else;
```

end FIFO_STAGE_ABS

A.6 The Environment Abstraction E_{left}.

monitor LEFT_ENV

monitor LEFT_ENV_MAIN
import TIMER, TR_CLOCK, TRANSMITTER, ACKER

selvar acked, timer_ok, out_t, out_f : boolean
selvar #: (constant, update_both)
selvar out_data: (zero, one, empty)
stvar prev_out_data: (zero, one, empty)
stvar prev_out_t, prev_out_f: boolean

asgn # : { constant, update_both }
asgn timer_ok := ~(TIMER.$ = lt_lb)

asgn out_data := zero, one, empty ? (((#=update_both) * timer_ok * acked)) * (prev_out_data= empty) |
empty ? (((#=update_both) * timer_ok * acked)) * ~(prev_out_data= empty) |
prev_out_data

asgn prev_out_data -> out_data

asgn out_t := 1 ? timer_ok * ((#=update_both)) * (out_data=one) |
0 ? timer_ok * ((#=update_both)) * ((out_data=zero) + (out_data=empty)) |
prev_out_t

asgn out_f := 1 ? timer_ok * ((#=update_both)) * (out_data=zero) |
0 ? timer_ok * ((#=update_both)) * ((out_data=one) + (out_data=empty)) |
prev_out_f

asgn prev_out_t -> out_t
asgn prev_out_f -> out_f

asgn acked := 1 ? ((prev_out_data = empty) * (ACKER.ack = 1) +
((prev_out_data = zero) + (prev_out_data = one)) * (ACKER.ack = 0)) |
0 ? ((prev_out_data = empty) * (ACKER.ack = 0) +
((prev_out_data = zero) + (prev_out_data = one)) * (ACKER.ack = 1)) |
0

init prev_out_data := empty, prev_out_t := 0, prev_out_f := 0

end LEFT_ENV_MAIN

monitor TIMER
import LEFT_ENV

selvar #: (%, go)
selvar start: boolean
stvar $: (idle, lt_lb, gt_lb_lt_ub)

init $:= idle

trans
idle { % }
    -> lt_lb: ∼(LEFT_ENV_MAIN.# = constant) * (LEFT_ENV_MAIN.acked)
    -> $: else;

lt_lb { % { $\delta_{c,min}$, $\delta_{c,min}$ } go }
    -> gt_lb_lt_ub: (#=go)
    -> $: else;

gt_lb_lt_ub { % }
    -> lt_lb: ∼(LEFT_ENV_MAIN.# = constant) * (LEFT_ENV_MAIN.acked)
    -> $: else;

end TIMER
end LEFT_ENV

A.7 The Environment Abstraction E_{right}.

monitor RIGHT_ENV
import SENDER

selvar ack: boolean
selvar $: (first, sample_even, sample_odd)
selvar sampled_t, sampled_f: boolean
stvar old_sampled_t, old_sampled_f: boolean

init old_sampled_t := 0, old_sampled_f := 0

asgn old_sampled_t -> sampled_t
asgn old_sampled_f -> sampled_f
asgn sampled_t := SENDER._out_t ? (SAMPLER.#=go) — old_sampled_t
asgn sampled_f := SENDER._out_f ? (SAMPLER.#=go) — old_sampled_f
asgn ack := A.#

proc A: DB((RIGHT_ENV.sampled_t + RIGHT_ENV.sampled_f), 1, $\delta_{nor,min}$, $\delta_{nor,max}$)

monitor SAMPLER
import REC_CLOCK

selvar #: (stvar $: (first, sample_even, sample_odd)
init $:= first@

trans
first { % , go }
    -> sample_even: (#=go)
    -> $: else;

sample_even { % { $\delta_{c,min}$, * } go }
    -> sample_odd: (#=go)
    -> $: else;

sample_odd { % { $\delta_{c,min}$, * } go }
    -> sample_even: (#=go)
    -> $: else;

end SAMPLER
end RIGHT_ENV

A.8 The Untimed Homomorphisms

A.8.1 E_{left}: Basis Step

```
hom $ := HOLD
hom out_data := TRANSMITTER.out_data
hom prev_out_data := TRANSMITTER.prev_out_data
hom out_t := TRANSMITTER.out_t
hom out_f := TRANSMITTER.out_f
hom acked := TRANSMITTER.acked

hom # := update_both ? (TRANSMITTER.$ = SEND) * (TR_CLOCK.#=0) * (TRANSMITTER.acked=1) |
         constant
```

A.8.2 E_{left}: Inductive Step

```
hom $ := HOLD
hom out_t := (FIFO_ABS.out_t)
hom out_f := (FIFO_ABS.out_f)
hom acked := FIFO_ABS.acked

hom out_data := one ? (FIFO_ABS.out_t=1) * (FIFO_ABS.out_f=0) |
                zero? (FIFO_ABS.out_t=0) * (FIFO_ABS.out_f=1) |
                empty

hom old_out_data := one ? (FIFO_ABS._old_out_t = 1) * (FIFO_ABS._old_out_f = 0) |
                    zero? (FIFO_ABS._old_out_t = 0) * (FIFO_ABS._old_out_f = 1) |
                    empty

hom # := update_both ? (FIFO_ABS.t_switch + FIFO_ABS.f_switch) |
         constant
```

A.8.3 E_{right}: Basis Step

```
hom sampled_t := REC.sampled_t
hom sampled_f := REC.sampled_f
hom ack := REC.ack
hom SAMPLER.# := go ? (REC_CLOCK.DEL.outp = go_zero) | %
```

A.8.4 E_{right}: Inductive Step

```
hom RIGHT_ENV.sampled_t := FIFO_ABS._out_t
hom RIGHT_ENV.sampled_f := FIFO_ABS._out_f
hom RIGHT_ENV.ack := FIFO_ABS._ack_out
hom RIGHT_ENV.# := go ? ((FIFO_ABS.t_switch=1) + (FIFO_ABS.f_switch=1)) | %

hom A.# := go_one ? ( FIFO_ABS.$ = wait_ack_out) *
                    ( ~(FIFO_ABS.out_t + FIFO_ABS.out_f) ) *
                    (FIFO_ABS.#=go) * ~(FIFO_ABS.prev_ack_out=1) |
           go_zero ? ( FIFO_ABS.$ = wait_ack_ out) *
                    ( (FIFO_ABS.out_t + FIFO_ABS.out_f) ) *
                    (FIFO_ABS.#=go) * ~(FIFO_ABS.prev_ack_out=0) |
           %
```

A.8.5 $F \preceq_L A$

In the following, we use FFS as shorthand for FIFO_STAGE.

```
hom out_t := FFS._out_t
hom out_f := FFS._out_f
hom prev_ack_out := ( (FFS.NOR.$= one) + (FFS.NOR.$ = zero_pending) )
hom ack_out := FFS.NOR.#

hom t_switch := (FFS.TC.DL.outp = go_zero) + (FFS.TC.DL.outp = go_one) +
          (FFS.FC.DL.outp = go_zero) + (FFS.FC.DL.outp = go_one)
hom f_switch := (FFS.TC.DL.outp = go_zero) + (FFS.TC.DL.outp = go_one) +
          (FFS.FC.DL.outp = go_zero) + (FFS.FC.DL.outp = go_one)


macro OT := ((FFS.TC.DL.$ = one) + (FFS.TC.DL.$ = zero_pending) )
macro OF := ((FFS.FC.DL.$ = one) + (FFS.FC.DL.$ = zero_pending) )
macro RACK :=((RIGHT_ENV.A.$ = one) + (RIGHT_ENV.A.$ = zero_pending))
macro FIFO_acked := ((OT+OF) * ~RACK + ~OT * ~OF * RACK)
macro tgo := ((FFS.TC.DL.outp = go_one) + (FFS.TC.DL.outp = go_zero))
macro fgo := ((FFS.FC.DL.outp = go_one) + (FFS.FC.DL.outp = go_zero))
macro tstable := ((FFS.TC.DL.$ = one) + (FFS.TC.DL.$ = zero))
macro fstable := ((FFS.FC.DL.$ = one) + (FFS.FC.DL.$ = zero))


hom $ := wait_ack_out     ? ( (FFS.NOR.$ = zero_pending) + (FFS.NOR.$ = one_pending)) |
         wait_out_change ? ( (FFS.TC.DL.$ = zero_pending) + (FFS.TC.DL.$ = one_pending) +
                             (FFS.FC.DL.$ = zero_pending) + (FFS.FC.DL.$ = one_pending) ) |
         wait_for_ack     ? ( ((FFS.TC.DMC.$ = i1_1_i2_0_o0) + (FFS.TC.DMC.$ = i1_0_i2_1_o1)) +
                              ((FFS.FC.DMC.$ = i1_1_i2_0_o0) + (FFS.FC.DMC.$ = i1_0_i2_1_o1)) ) |
         wait_for_data    ? ( ((FFS.TC.DMC.$ = i1_1_i2_0_o1) + (FFS.TC.DMC.$ = i1_0_i2_1_o0)) +
                              ((FFS.FC.DMC.$ = i1_1_i2_0_o1) + (FFS.FC.DMC.$ = i1_0_i2_1_o0)) ) *
                              (FIFO_acked) |
         stable
hom # := go ? ((FFS.NOR.outp = go_zero) + (FFS.NOR.outp = go_one)) +
               tgo * fgo + tgo * fstable + fgo * tstable |
          %
```

A.9 The Correctness Properties

```
monitor INPUTS_OK: STOP( ( (TRANS.$=SEND) *
                    (TR_CLOCK.#=0) * ( TRANS.acked)) )
monitor OUTPUTS_OK: STOP( ( (REC.sample_now) *
                    (FIFO.out_t=REC.old_sampled_t) * (FIFO.out_f = REC.old_sampled_f) ) )
```