Copyright © 1995, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

ITERATIVE METHODS FOR FORMAL VERIFICATION OF DIGITAL SYSTEMS

by

Felice Balarin

Memorandum No. UCB/ERL M95/1

4 January 1995

ITERATIVE METHODS FOR FORMAL VERIFICATION OF DIGITAL SYSTEMS

Copyright © 1994

by

Felice Balarin

Memorandum No. UCB/ERL M95/1

4 January 1995

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

ITERATIVE METHODS FOR FORMAL VERIFICATION OF DIGITAL SYSTEMS

Copyright © 1994

by

Felice Balarin

Memorandum No. UCB/ERL M95/1

4 January 1995

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Abstract

Iterative Methods for Formal Verification of Digital Systems

by

Felice Balarin

Doctor of Philosophy in Engineering–Electrical Engineering and Computer Sciences

University of California at Berkeley

Professor Alberto L. Sangiovanni-Vincentelli, Chair

Complexity management is the key to applying formal verification methods to real-life digital designs. Abstractions are a powerful tool to manage complexity, but finding a useful abstraction is a difficult task requiring significant designer's effort. In this work I propose techniques for automatic abstraction for three classes of systems: networks of communicating finite-state machines, real-time systems and arrays of identical components.

I show that ignoring communication in a network of finite-state machines can be used to simplify the representation of the system. Ignoring communication can prove to be too simplistic. In that case communication is selectively restored. The process is repeated until a suitable abstraction is found. I show that the process terminates in finitely many iterations.

We also develop a similar approach for real-time systems. All timing constraints are initially relaxed, and if that abstraction is proven too simplistic, then some of them are enforced. Again, the process is iterated. I consider two models of real-time systems: a basic model of real-time systems, called *timed automata* and propose an extended one called *timed automata with decrements* (TAD's) that allows modeling some high-level features of systems (e.g. interrupts) that can not be modeled accurately with timed automata. I show that the proposed iterative process always terminates for timed automata, while it may not terminate for arbitrary TAD's. I also show that this limitation is intrinsic, because the verification problem is undecidable for TAD's. Finally, I will show how to select automatically a subset of timing constraints necessary to verify a real-time system. Since typically only a small fraction of timing constraints is relevant to any given property, eliminating the rest of the

constraints can improve the efficiency of the verification process dramatically.

Finally, for arrays of identical components I address the problem of finding an abstraction which is independent of the actual number of components. Such an abstraction can then be used to verify a whole class of arrays of different sizes. I show that the problem is undecidable in general, and propose some search strategies that can find such an abstraction in special cases.

Professor Alberto L. Sangiovanni-Vincentelli
Dissertation Committee Chair

Contents

List of Figures

List of Tables

Acknowledgements

Many have contributed to shaping this work (and myself), and without any of them this work would be much different, or would never happen, but none was so influential as my advisor Prof. Alberto Sangiovanni-Vincentelli. It all started much before I came to Berkeley, when as a fresh graduate I started my first research steps, and while reading my first papers, more often than not, Alberto was involved. Those papers made me want to do research in CAD, and those papers made me want to come to Berkeley. During my graduate studies, Alberto was there to put an enormous body of work (including my own) into the right perspective and to constantly encourage me to take my ideas as far as they can go. Many times it was his *"You got it!"* that made me realize, that I am on the right track, and that I should try to develop what was just a foggy idea at a time, to theorems, algorithms and methods. For all he has given to me and this work, I am deeply grateful to Alberto.

Prof. Robert Brayton has been a member of my graduation committee, but my gratitude to him goes much further than that. I learned from him that it is worthwhile to take some time and effort and get to the bottom line, and I hope that this is reflected in this thesis. Thanks to Prof. John Rice for being in my committee and teaching me to interpret statistical data carefully. This lesson will be useful long after my thesis is done. I am also grateful to Robert Kurshan, who sparked my interest in formal verification with several illuminating lectures that he gave at Berkeley in 1991. On many occasions after that it was his comments that helped me choose the right direction in research.

Almost everything I know about formal verification I learned together with Berkeley verifiers: Adnan Aziz, Bob Brayton, S.-T. Cheng, Ramin Hojati, Tim Kam, Sriram Krishnan, Yuji Kukimoto, Rajeev Murgai, Rajeev Ranjan, Alberto Sangiovanni-Vincentelli, Tom Shiple, Vigyan Singhal, Paul Stephan, Gitanjali Swamy, Serdar Tasiran, Tiziano Villa, and H.-Y. Wang. To all of them: thank you for sharing your thoughts and for all the seminars that were always longer than expected, but for a good reason.

Several people directly contributed to the contents of this thesis. I thank Karl Petty and Prof. Pravin Varaiya who are responsible for designing PATHO operating system and who initiated its formal verification. I also thank Anuj Puri for sharing with me the automated factory example, and providing a lot of interesting insights into real-time systems.

Many people contributed to this work by making sure that I can fully concen-

trate on it. For that, my warmest "thank you" goes to Flora Oviedo, who has been helping me on a daily bases, often beyond her call of duty, and also to Kia Cooper, Elise Mills, Irena Stanczyk-Ng, Tim Castro, Heather Brown, Brad Krebs, and last but not least DARPA(contract JFBGI90-073) and SRC (contract DC-008-008) who have been funding me for all these years.

It is impossible to thank individually all the people who helped me in doing this work (and I apologize to all whom I have not mentioned), but it is not hard to decide who shared most of the burden with me: Marin and Verica. My gratitude to them is beyond words. I hope they feel it in every precious moment we spent together.

Chapter 1

Introduction

Verification is an important component of any design methodology. A design error that is discovered late in the design process can significantly increase the cost of a project. A design error discovered after a system is put in use is usually even costlier, either in the form of expensive product recalls and customer dissatisfaction, or in the worst case, in form of failure of life-critical systems such as flight controllers.

Presently, simulation is a prevailing method for verifying digital systems. But as technology advances and systems become more complex, verification by simulation is becoming increasingly insufficient. An illustrative example was reported by Chen, Yamazaki and Fujita [CYF94]. A data-switching chip (assumed to be correct by the designers) was exhibiting incorrect behavior, but only sporadically, and only after several seconds of operation at $156MHz$. Simulating such long input patterns is clearly very expensive, and the probability of selecting one that exhibits faulty behavior is quite low. However, by using a formal verification tool Chen et al. were able to identify the fault. The tool also generated a short (50 cycles) input pattern that exhibits the fault.

In the future, simulation is likely to remain an important verification method, but it is also likely that it will be more and more supplemented by formal verification methods. Formal verification is particularly effective in early design phases, when the system is described at a high level of abstraction, hence is less complex then later refinements, and must satisfy properties that are typically simple and easy to specify formally.

The phrase "formal verification" is broadly used to indicate any technique where one, with a rigor of a mathematical proof, establishes (or disproves) a *satisfaction* relation between a *design* and a *specification*. Formal verification paradigms vary with a choice of

formalisms for the design and specification, and a choice of a satisfaction relation. The basic tradeoff is between expressiveness and efficiency. On one side of the spectrum are general theorem proving techniques which are very expressive, but are provably hard, and can be only partially automated. On the other side are finite-state techniques that suffer from somewhat limited expressiveness, but can be completely automated.

1.1 Automatic verification of finite-state systems

Two approaches to automatic verification of finite-state systems have emerged in the last decade: *model checking* based on temporal logics, and *language containment* based on automata theory. In the model checking approach a system is verified if it represents a model of a given formula in some temporal logic. Most of the logics proposed for specification and verification of finite-state systems can be classified as either *branching time* or *linear time* temporal logics (see [Eme90] for excellent survey). The precise formulation of the model checking problem is somewhat different for each class.

Models of branching time temporal logics are trees, but their subformulas reason about paths in a tree. Typically, a subformula may postulate that another formula holds in every (or some, or the first, or the second, ...) state along a path. A formula is then closed by existentially (or universally) quantifying over paths. In other words, a formula is true of the tree rooted in a state, if the subformula is true on some (or all) paths from that state. With every finite-state system we associate an infinite tree with the initial state as its root, and children of every state being its possible next states. A system is verified, if a formula is true of the associated tree. The first efficient automatic model checking procedure for any temporal logic was given Clarke, Emerson and Sistla [CES86] for the branching time logic CTL, which still remains the most widely used logic in verification. For example, verification systems SMV [McM93], and HSIS [ABB+94] are based on it.

Formulas of linear time temporal logics reason about paths, and the system is verified if the formula is true for all paths from the initial states. The use of linear time temporal logics for specification and verification of finite-state systems dates back to late seventies and early eighties when in a trailblazing contribution Pnueli and Manna [Pnu77, MP81] use such a logic to specify many interesting properties of computer programs. Linear time temporal logics are still an interesting research topic, but their inherent complexity [SC85] has limited their use in practice.

Linear temporal logics are closely related to another popular verification paradigm known as *language containment*, pioneered among the others by Vardi and Wolper [VW86]. Here, systems are model as *automata*, and the *language* accepted by an automaton is assumed to be its behavior. In other words, the behavior of a finite state system is a set of input-output sequences that can be observed from the outside. The verification problem is to check whether every sequence in the language is acceptable, i.e. we need to check that the language of a system is contained in the language consisting of all acceptable sequences. That language is specified by another automaton, so the verification problem reduces to checking language containment between two automata.

Model checking of linear temporal logics can be reduced to language containment [VWS83, VW86]. First, we construct an automaton that accepts all the sequences that satisfy the formula. Then, we check whether the language of that automaton contains the language of the system.

1.2 Real-time systems

In all of the approaches mentioned so far time is modeled qualitatively. One can check properties about the ordering of events, but not about exact times of occurrences. However, for many systems (especially embedded real-time controllers) the precise quantitative timing constraints are essential.

The complexity of timing constraints rises with a level of abstraction at which a system is described. At the gate level, assigning fixed delays to gates may be sufficient to capture timing constraints. Then, simple longest path techniques can be used to verify the system. At a higher level of abstraction, constraints are typically more complex. For example, the speed by which a software task progresses might change in time depending on the load of the system. Many formalisms that can capture complex timing constraints have been proposed (e.g. see [Vyt91, dBHdRR91, AH92] for a compilation of many approaches), but most are not suitable for automatic verification, because the associated verification problems are undecidable. This is true even for some straightforward extensions of standard finite-state models (e.g. [AL91]). The problem is that if real-valued time becomes a state component, then the system is no longer finite-state, and the state space cannot be searched exhaustively in finite time.

Alur and Dill [AD90, Dil89] have proposed *timed automata* as a model of real-time

systems, and showed that they have a finite-state representation. They propose adding timing information to automata through a real-valued variables called *timers*. Timers can be used to bound elapsed time between any two transitions. Even though the (real) values of timers are state components of a timed automaton, Alur and Dill have shown that many of these states are equivalent, and that in fact the number of equivalence classes is finite.

In a similar fashion, temporal logics can be extended with real-time operators, and then the infinite-state model checking can be reduced to a finite state one [ACD90, Alu91, AH92]. Unfortunately, the number of equivalence classes in the Alur-Dill's construction is exponential both in the number of timers, and in the sizes of time constants used in a system (or property) specification.

1.3 Complexity management

Even though the verification of finite-state systems can be completely automated, it is still not widely used in practice, mainly due to issues related to specification of correct behavior, design methodology, and complexity of the associated algorithms.

Before using a formal verification tool, a user must first specify acceptable behaviors (in case of language containment, the acceptable behaviors are represented by a language, and in case of model checking, it is represented by a formula). In the worst case, specifying acceptable behaviors is as hard and as error-prone as actually designing a system. Fortunately, specification is often much simpler than design for at least the following reasons:

- It is often possible to express *what* a system must do (a specification) much more concisely than *how* it is doing it (a design).
- Acceptable behaviors can usually be specified as a list of *properties* that a system must satisfy. Even though properties are typically simple, designing a system that satisfies all of them is not.

To reduce the possibility of errors, the properties must not only be simple, but also expressed in a way that is natural to designers and consistent with other system documentation. Unfortunately, neither finite-state automata, nor CTL formulas satisfy these conditions. A promising approach to this problem is to develop translators that provides formal interpre-

tation (in terms of automata or temporal logic formulas) of specification methods used by designers, such as HDL annotations [NJK94, ALG+91, BBC+] or timing diagrams [SD93].

Another issue that has to be addressed before automatic formal verification is widely accepted is the development of a verification based design methodology. This includes engineering issues like providing a common design representation to be used by synthesis, simulation and verification tools, theoretical issues like identifying properties that are preserved under a given set of synthesis operations, as well as organizational issues such as deciding at which points in the design cycle simulation or verification tools should be used to optimize the overall design process. Some of these issues have been addressed in the literature [ABB+94, BBC+, Kur94], but many practical questions have yet to be resolved.

Last but not least, a formal verification method can be successful only only if accompanied with elaborate complexity management techniques. At first glance, it might seem that complexity is not a significant issue because algorithms polynomial in the number of states exist both for CTL model checking and for language containment. However the problem is that in practice, systems to be verified are never specified by explicitly enumerating all states. Typically, they are specified as a composition of several components which are specified either as software (a program in some language), or as hardware (combinational logic plus latches). In this case, the number of states can be exponential in the size of the description, and in fact both CTL model checking and language containment of systems consisting of interacting components are PSPACE-complete [AB93]. This is the well known *state explosion problem.* As we have seen, the problem is even more severe for real-time systems, where adding precise timing information adds another layer of complexity to the system.

Attacking the state explosion problem is the focus of a wide range of research. Two approaches dominate: *symbolic computation* where one try to manipulate a large number of states efficiently by representing sets of states symbolically (rather than by enumeration), and *simplification* where one argues about the correctness of complex systems by verifying their simplified versions. Those two approaches are independent, and in fact several researcher have proposed methods that combine both approaches.

By far the most widely used approach to symbolic computation is to represent sets of states with their characteristic functions. If states are encoded with binary variables, then the characteristic function of a set is just a Boolean function, and is typically represented by a *binary decision diagram* (BDD) [BRB90]. This approach is successful because in

many practical cases large sets of states have quite small BDD representations, and thus it is possible to manipulate sets of states that are too large to enumerate with existing computing resources. BDD-based algorithms (and verification systems) are available both for model checking [McM93] and language containment [ABB+94] The success of BDD-based approaches have made them almost synonymous to symbolic computation. Still, not all symbolic approaches are BDD-based. For example, in the verification of real-time and hybrid systems, sets of linear inequalities are used to represent convex polyhedra bounded by them [HNSY92, ACHH93].

Recent advancements in symbolic computation techniques have significantly increased the capabilities of automatic formal verification, but simplifications are still necessary to handle most real-life systems. We make a distinction between two kinds of simplifications: *exact* and *conservative*. Exact simplifications (or *reductions*) preserve all aspects of system behavior, therefore the original system is verified *if and only if* the simplified system is. They can be applied to virtually any verification formalism, but it is often hard to find an exact simplification of reasonable size.

On the other hand, conservative simplification (or *abstractions*) preserve enough behavior of a system to guarantee that the original system is verified *if* the simplified system is. However, if the simplified system is not verified, the original system might or might not satisfy the required property. Conservative approximations can often lead to much larger savings than exact simplifications, but they exist only for some formalisms. In particular, if the set of properties expressible in a formalism is closed under complementation (i.e. if for every property P there exists a property $\overline{P}$ such that a system S satisfies P if and only if it does not satisfy $\overline{P}$), then every abstraction is also a reduction. To see this, assume towards a contradiction that S' is an abstraction but not a reduction of a system S, i.e. assume that there exists a property P such that S satisfies P but S' does not. This implies that S' satisfies $\overline{P}$ even though S does not, contradicting the assumption that S' is an abstraction of S.

Most of branching time temporal logics (including CTL) are closed under complementation, hence they allow only exact simplification. However, if the logic is restricted to universal path quantifiers (and of course no negation), then conservative simplifications are possible. Roughly speaking, any system with more paths is an abstraction of the original system. For example, Grümberg and Long [GL91] have studied such a restriction of CTL (called ACTL), and showed that one system is an abstraction of another, if so-called

simulation relation holds between them.

The language containment (hence also model checking of linear time temporal logics) clearly allows conservative approximations: an automaton is an abstraction if its language contains the language of the original system. But that means that checking whether one system is an abstraction of the other is just another instance of the original verification problem (and hence just as hard). Still, a careful use of abstractions can be beneficial. For example, one might check abstractions of (usually small) components, and deduce from that an abstraction of the (usually large) complete system. These and other strategies for simplification of communicating automata were studied by Kurshan [Kur90, Kur94].

The problem of checking automatically whether one system is an abstraction of another has received a wide attention in the last decade, but generating abstractions is currently mostly a manual task performed by users of verification systems. Since almost all real-life systems require at least some abstractions, it follows that completely automatic verification is not achieved even for finite-state systems.

A good abstraction must balance two (often conflicting) requirements:

- it must be simple enough so that a verification tool can handle it efficiently,
- it must preserve the behavior of the original system in many aspects, particularly those that are relevant to the property to be verified.

To satisfy the first requirement a user (or an automatic method) must take into account a measure of complexity that is reasonable for the underlying verification algorithm. For example, a number of reachable states may be a good measure of complexity for algorithms based on explicit state enumeration, while communication complexity between subsystems may be a better measure for BDD-based algorithms. Therefore, algorithm-specific methods are likely to yield better results than general purpose ones.

To satisfy the second requirement a user must somehow capture the essence of the behavior of the system with respect to the property at hand. This is clearly not a task that could ever be completely automated. The best we can hope for is to develop some heuristic approaches targeted at certain classes of systems.

Balancing these requirements is hard, even for an experienced designer with a deep understanding of a system being verified. Most of the times, the first attempt does not result in an abstraction that satisfies the property. In that case, a user have to analyze

a failure report from a verification tool and decide whether the failure is inherent in the original system, or a consequence of some over-simplification. If the latter applies, a user needs to modify the current abstraction and try again until the final answer is reached. Automating this process yields the following generic iterative abstraction algorithm:

INITIALIZE: Choose an initial abstraction.

VERIFY: If the the current abstraction of the system satisfies the property to be verified, then return $PASS$, otherwise proceed with the next step.

ANALYZE: If a failure trace reported by a verification tool is a valid example of the behavior of the original system, then return $FAIL$, otherwise proceed with the next step.

MODIFY: Refine the current abstraction of the system such that it can no longer exhibit the reported faulty behavior. Proceed with the **VERIFY** phase.

To make this approach useful, a careful choice of abstractions in the **MODIFY** and **INITIALIZE** phase has to be made, and efficient failure analysis methods have to be developed. For example, if an initial abstraction is too weak, the system might not be simplified enough for a verification tool to handle. Also, if modifications are too small the number of iterations may be large or even infinite. Finally, the failure analysis requires checking whether a given trace is a valid behavior of the original system. In general, this problem may be as hard as the original verification problem, but if a choice of abstraction is restricted it may be significantly simpler.

1.4 Outline of this work

The language containment verification framework is introduced in details in chapter 2. We describe the approach both for automata on finite and infinite strings. Automata on infinite strings are somewhat more expressive, so whenever possible we use them. However, some of our results are valid only for automata on finite sequences. We develop all our results for language containment, but basic ideas extend also to similar formalisms such as ACTL model checking.

In chapter 3 we specialize the generic iterative abstraction algorithm of section 1.3 to the verification of communicating finite state machines. The basic idea is to abstract

communication between components. Doing this typically increases the number of reachable states, but reduces the BDD representation. Thus, the approach can be efficient only if applied with BDD-based verification system.

Chapters 4–6 are dedicated to formal verification of real-time systems. First, we introduce a model of real-time system in chapter 4. We base this model on timed automata and extend it with some additional capabilities. Those extensions allow more faithful modeling of systems, but make the verification problem undecidable in general. Therefore, in chapter 4 we first focus on timed automata. We propose an iterative algorithm for such systems, and show how user guidance can be used to improve efficiency. We deal with the extended model later (in chapter 6).

A substantial example, formal verification of the PATHO real-time operating system is presented in chapter 5. We introduce several models of PATHO, and report experimental results using iterative abstraction algorithm both in automatic and guided modes. In chapter 6, we present a more accurate model that cannot be modeled with timed automata without the extensions we have proposed in chapter 4. To be able to verify such systems, we extend the iterative algorithm of chapter 4. The result is a semi-decision procedure. It always returns a correct result if it terminates, but in general it might not terminate.

In chapter 7 we propose an approach to abstraction of arrays of identical (untimed) components. This approach is also iterative, but it is quite different from approaches in chapters 3 and 4. In the latter approaches we maintain an abstraction of the system in every iteration, while in the former, the simplification becomes conservative only in the last iteration. We show that the problem is undecidable in general, hence there may not be the last iteration, i.e. our procedure may not terminate.

Finally, in chapter 8 we summarize our results, and discuss possible future research trends that could bring formal verification techniques closer to the design practice.

Chapter 2

Formal Verification of Finite-State Systems

Any formal approach to verification has three components: a formalism for describing systems, a formalism for describing properties to be verified, and a notion of correctness, i.e. a precisely defined relation that must hold between a system model and a property model, for a system to be considered correct. We focus on the approach where both the system and the property are modeled as finite-state *automata*. The notion of correctness is that of *language containment*: a system is correct if its language is contained in the language of the property. In this chapter we present both rigorous definitions and the intuition behind these choices. But first we review the basic mathematical tools used in this approach.

2.1 Sets, characteristic functions and BDD's

Let $V = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\}$ be some set of variables, where $\mathbf{x}_i$ takes values from some set U_i. The set of *Boolean formulas* over V is defined recursively as follows:

1. **1** and **0** are Boolean formulas,
2. $\mathbf{x}_i = a$ is a Boolean formula for all $\mathbf{x}_i \in V$ and all $a \in U_i$,
3. if **F** and **G** are Boolean formulas, then so are also $\mathbf{F} * \mathbf{G}$ (conjunction), $\mathbf{F} + \mathbf{G}$ (disjunction), and $\overline{\mathbf{F}}$ (complementation).

We interpret Boolean formulas as *Boolean functions* from $U_1 \times \ldots \times U_n$ to $\{0,1\}$ as follows:

- $\mathbf{1}(a_1,\ldots,a_n) = 1$ for all $(a_1,\ldots,a_n) \in U_1 \times \ldots \times U_n$,
- $\mathbf{0}(a_1,\ldots,a_n) = 0$ for all $(a_1,\ldots,a_n) \in U_1 \times \ldots \times U_n$,
- $(\mathbf{x}_i = a)(a_1,\ldots,a_n) = 1$ if and only if $a_i = a$,
- $(\overline{\mathbf{F}})(a_1,\ldots,a_n) = 1$ if and only if $\mathbf{F}(a_1,\ldots,a_n) = 0$,
- $(\mathbf{F} * \mathbf{G})(a_1,\ldots,a_n) = 1$ if and only if $\mathbf{F}(a_1,\ldots,a_n) = 1$ *and* $\mathbf{G}(a_1,\ldots,a_n) = 1$,
- $(\mathbf{F} + \mathbf{G})(a_1,\ldots,a_n) = 1$ if and only if $\mathbf{F}(a_1,\ldots,a_n) = 1$ *or* $\mathbf{G}(a_1,\ldots,a_n) = 1$.

If some total order $\leq$ is defined on U_i, then we also allow $\mathbf{x}_i \leq a$ to be a Boolean formula, and interpret it naturally by:

- $(\mathbf{x}_i \leq a)(a_1,\ldots,a_n) = 1$ if and only if $a_i \leq a$.

The *support* of some formula $\mathbf{F}$ (denoted by $supp(\mathbf{F})$) is the set of variables that actually appear in $\mathbf{F}$, i.e.:

$$\begin{aligned} supp(\mathbf{1}) &= supp(\mathbf{0}) &&= \emptyset\ , \\ supp(\mathbf{x}_i = a) &= supp(\mathbf{x}_i \leq a) &&= \{\mathbf{x}_i\}\ , \\ supp(\mathbf{F} * \mathbf{G}) &= supp(\mathbf{F} + \mathbf{G}) &&= supp(\mathbf{F}) \cup supp(\mathbf{G})\ , \\ & supp(\overline{\mathbf{F}}) &&= supp(\mathbf{F})\ . \end{aligned}$$

Note that to specify a Boolean function we need to specify (i) a Boolean formula, and (ii) its domain, i.e. a set of variables V (which includes at least its support, but may also include some additional variables), and the ranges of all variables in V. Often, we specify only a formula, while a domain can be inferred from the context. In fact, this feature significantly simplifies our notation, because we often interpret the same formula over different domains. In particular, unless stated otherwise, we assume that some formulas $\mathbf{F}$ and $\mathbf{G}$ are defined over $supp(\mathbf{F})$ and $supp(\mathbf{G})$ respectively, but when combined (in $\mathbf{F} * \mathbf{G}$ or $\mathbf{F} + \mathbf{G}$), we will assume that they are both defined over $supp(\mathbf{F}) \cup supp(\mathbf{G})$.

Every Boolean function $\mathbf{F} : U_1 \times \ldots \times U_n \rightarrow \{0,1\}$ characterizes a set $F \subseteq U_1 \times \ldots \times U_n$ defined by:

$$F = \{(a_1,\ldots,a_n) \mid \mathbf{F}(a_1,\ldots,a_n) = 1\}\ .$$

We say that $\mathbf{F}$ is a *characteristic function* of F. Given some set F we use $\mathcal{X}(F)$ to denote its characteristic function, and inversely, given some Boolean function $\mathbf{F}$ we use $\mathcal{S}(\mathbf{F})$ to denote the set it characterizes.

Let $\mathbf{F}$ and $\mathbf{G}$ be two Boolean formulas over the same domain. It is straightforward to relate the operations on formulas to the operations on the sets they characterize:

$$\begin{aligned}
\mathcal{S}(\mathbf{F} * \mathbf{G}) &= \mathcal{S}(\mathbf{F}) \cap \mathcal{S}(\mathbf{G}) \quad \text{(conjunction means intersection)}, \\
\mathcal{S}(\mathbf{F} + \mathbf{G}) &= \mathcal{S}(\mathbf{F}) \cup \mathcal{S}(\mathbf{G}) \quad \text{(disjunction means union)}, \\
\mathcal{S}(\overline{\mathbf{F}}) &= \{s | \, s \notin \mathcal{S}(\mathbf{F})\} \quad \text{(complementation means complementation)}.
\end{aligned}$$

Another useful interpretation of the conjunction applies when $\mathbf{F}$ and $\mathbf{G}$ have disjoint support and we interpret $\mathbf{F}$, $\mathbf{G}$, and $\mathbf{F} * \mathbf{G}$ as functions over their respective support. In this case, conjunction corresponds to the Cartesian product, i.e.:

$$\mathcal{S}(\mathbf{F} * \mathbf{G}) = \mathcal{S}(\mathbf{F}) \times \mathcal{S}(\mathbf{G}) = \{(s,q) | \, s \in \mathcal{S}(\mathbf{F}), q \in \mathcal{S}(\mathbf{G})\} \ .$$

An important operation in manipulation of Boolean functions is *existential quantification.* Given some function $\mathbf{F}$ over variables V and some $\mathbf{x}_i \in V$, to *"existentially quantify* $\mathbf{x}_i$ *out of* $\mathbf{F}$*"* means to compute a function over $V - \{\mathbf{x}_i\}$, denoted by $\exists \mathbf{x}_i \, . \, \mathbf{F}$, and defined naturally as follows:

$$(\exists \mathbf{x}_i \, . \, \mathbf{F})(a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n) = \begin{cases} 1 & \text{if there exists } a_i \in U_i \text{ such that:} \\ & \quad \mathbf{F}(a_1, \ldots, a_{i-1}, a_i, a_{i+1}, \ldots, a_n) = 1 \ , \\ 0 & \text{otherwise.} \end{cases}$$

Finally, given some Boolean formula $\mathbf{F}$ and some variables $\mathbf{x}$ and $\mathbf{y}$, we use $[\mathbf{F}]_{\mathbf{x} \mapsto \mathbf{y}}$ to denote the formula obtained from $\mathbf{F}$ by replacing all occurrences of $\mathbf{x}$ in $\mathbf{F}$ with $\mathbf{y}$.

Usually, the range of all the variables in systems we consider is finite.[1] In that case, Boolean functions are usually represented with *binary decision diagrams* (BDD's)[BRB90]. Details of BDD representation will not be of interest here, thus we will only list some properties BDD's that are useful in analyzing complexity of our algorithms:

- a BDD representation is canonical, thus if implemented properly, comparing two functions for equivalence can be as simple as comparing two pointers,

[1] A notable exception are real-time systems introduced in chapter 4 which include real-valued variables, but as we shall see BDD representation can be used even in this case.

- representations of functions **1**, **0**, and $\mathbf{x}_i = a$ are small and do not depend on the number of variables in V,
- the BDD size for $\overline{\mathbf{F}}$ is the same as the size for **F**,
- the BDD size for $\mathbf{F} * \mathbf{G}$ or $\mathbf{F} + \mathbf{G}$ is in the worst case equal to the product of BDD sizes for **F** and **G**,
- the size of the BDD for $\exists \mathbf{x}_i \,.\, \mathbf{F}$ can in the worst case be exponentially larger than the size of the BDD for **F**, but the worst case is achieved only in degenerate cases, and in most cases the BDD for $\exists \mathbf{x}_i \,.\, \mathbf{F}$ is smaller than the BDD for **F**,
- computing $\overline{\mathbf{F}}$ is a constant time operation,
- the time needed to compute $\mathbf{F} * \mathbf{G}$, and $\mathbf{F} + \mathbf{G}$ is proportional to the size of the result; for $\exists \mathbf{x}_i \,.\, \mathbf{F}$ the size of the result is a lower bound.

In summary, checking for equivalences, constructing basic functions, and complementing is cheap, but in the worst case even after small (polynomial) number of conjunctions, disjunctions, and quantifications, the BDD representation can grow (exponentially) large.

These properties illustrate difficulties in complexity analysis of BDD-based algorithms. Judging by worst case performance only, BDD's are not very useful, but in many interesting cases the performance is much better than in the worst case. Also for many operations, computation time is proportional to the space needed to store the data, thus the size of the BDD representation is often the limiting factor.

2.2 Automata and languages

To model digital systems, we use finite state *finite-state automata.* These are finite structure that can represent possibly infinite sets of sequences of inputs and outputs of a system. There exists a wide body of knowledge concerning automata both on finite (e.g. see [HU79]) and infinite sequences (e.g. [Tho90, Eme90]). Here, we present only fragments of the developed theory that are relevant to our research results, and to the implementation of an efficient verification tool based on language containment.

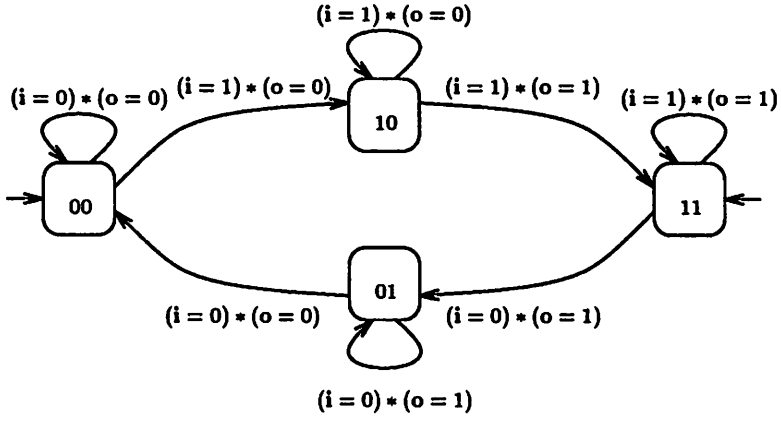


Figure 2.1: A model of a buffer.

2.2.1 Automata on finite strings

Let Σ denote some set of *I/O values*. An *automaton* A over finite strings of values in Σ is a 4-tuple $(S, \mathbf{I}, \mathbf{T}, \mathbf{F})$, where S is some *set of states*, $\mathbf{I} : S \to \{0,1\}$ is (a characteristic function of) a set of *initial states*, $\mathbf{T} : S \times \Sigma \times S \to \{0,1\}$ is (a characteristic function of) a *transition relation*, and $\mathbf{F} : S \to \{0,1\}$ is (a characteristic function of) a set of *final states*.

As a matter of convention, we assume that the formula $\mathbf{T}$ is defined over the variables $\mathbf{ps}_A$ (representing the present state), $\mathbf{ns}_A$ (representing the next state), and σ_A (representing the I/O values), where $\mathbf{ps}_A$ and $\mathbf{ns}_A$ range over S, and σ_A ranges over Σ. When no confusion can arise we will drop the subscripts.

We say that a sequence of states $s_0, s_1, \ldots, s_n$ is a *run* of a string $\sigma_1\sigma_2\ldots\sigma_n \in \Sigma^*$ in the automaton $A = (S, \mathbf{I}, \mathbf{T}, \mathbf{F})$ if $\mathbf{T}(s_{i-1}, \sigma_i, s_i) = 1$ for all $i = 1, \ldots, n$. A run is *initialized* if $\mathbf{I}(s_0) = 1$ (s_0 is an initial state). A run is *accepting* if it is initialized and $\mathbf{F}(s_n) = 1$ (s_n is a final state). The language of A (denoted by $\mathcal{L}(A)$) is the set of all strings that have an accepting run in A.

Consider for example a model of a one-bit buffer with an arbitrary delay shown in Figure 2.1. As long as the input and the output values (represented by variables $\mathbf{i}$ and $\mathbf{o}$, respectively) are the same, the buffer remains in one of the stable states (00 or 11). When the input value changes, the output changes only after the buffer spends some time in a transient state (01 or 10). We complete the definition by stating that only stable states are initial and that all state are final. An example of a string in the language of the buffer is

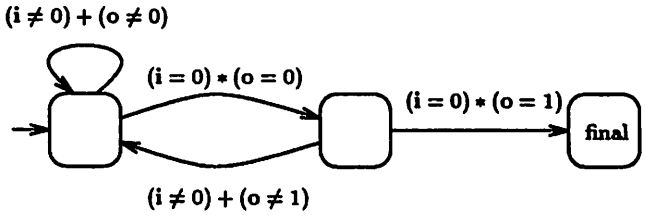


Figure 2.2: An automaton for a safety property.

(0/0, 1/0, 1/0, 1/1), which has an accepting run (00, 00, 10, 10, 11).

Automata on finite strings are sufficient to verify properties that can always be disproved by a finite sequence of events (also called *safety* properties). Assume that we want to verify that if the input and the output of the buffer are both zero, then the output cannot change before the input, i.e. we want to show that the I/O value 0/0 can never be followed by the value 0/1. To disprove this property, we need to show a finite string in the language of the buffer that ends with the I/O value 0/0 followed by 0/1. Thus, to check the property we:

1. construct an automaton that accepts exactly those strings that disprove the property; such an automaton is shown in Figure 2.2,

2. check whether the languages of the automata in Figures 2.1 and 2.2 intersect.

Typically, the first step is performed manually, and the second step is done automatically by a verification tool.

We say that an automaton $A = (S, \mathbf{I}, \mathbf{T}, \mathbf{F})$ is *complete* if for every $s \in S$ and every $\sigma \in \Sigma$ there exists at least one $q \in S$ such that $\mathbf{T}(s, \sigma, q) = 1$. If for every $s \in S$ and every $\sigma \in \Sigma$ there exists at most one $q \in S$ such that $\mathbf{T}(s, \sigma, q) = 1$, then we say that A is *deterministic*. Note that notions of completeness and determinism depend only on the transition relation, and thus are equally applicable to automata of infinite sequences.

It is easy to see that if A is complete, then every string in Σ^* has a run in A. If A is also deterministic, then such a run is unique. This property reduces the problem of complementing the language to switching accepting and non-accepting runs. Thus, given a deterministic and complete automaton $A = (S, \mathbf{I}, \mathbf{T}, \mathbf{F})$, we can easily construct the automaton $(S, \mathbf{I}, \mathbf{T}, \overline{\mathbf{F}})$ which accepts all strings not in $\mathcal{L}(A)$.

The completeness is easily obtained. Given any automaton A we can construct a complete automaton A' as follows:

- states of A' are those of A augmented by a special *"dead"* state @,
- initial and final states of A' are those of A,
- for all $s, q \in S_A$ and all $\sigma \in \Sigma$: $s \xrightarrow{\sigma} q$ is in the transition relation of A' if and only if it is in the transition relation of A,
- for all $s \in S_A$ and all $\sigma \in \Sigma$: $s \xrightarrow{\sigma} @$ is in the transition relation of A' if and only if there does not exist $q \in S_A$ such that $s \xrightarrow{\sigma} q$ is in the transition relation of A,
- $@ \xrightarrow{\sigma} @$ is in the transition relation of A' for all $\sigma \in \Sigma$.

In other words, incomplete runs in A are completed in A' by moving to the dead state and remaining there forever. It is easy to check that A' is deterministic if A is. It is also easy to check that languages of A and A' are the same.

The second requirement for an easy complementation (determinism) is not so easily obtained. In fact, examples are known where determinization and complementation must incur an exponential blow-up.

2.2.2 Automata on infinite sequences

While there is only one widely accepted sort of automata on finite sequences, several variants of automata on infinite sequences (distinguished mainly by their acceptance conditions) have been used extensively both in theory and in practice. In our development we use a version due to Streett [Str82], which we introduce next in detail. We will review other kinds suggested for formal verification in section 2.3.

An *automaton* over *infinite* sequences of values in Σ is a 4-tuple $(S, \mathbf{I}, \mathbf{T}, F)$, where S, $\mathbf{I}$, and $\mathbf{T}$ are as in finite case, and

$$F = \{(\mathbf{L}_1, \mathbf{U}_1), (\mathbf{L}_2, \mathbf{U}_2), \ldots, (\mathbf{L}_{|F|}, \mathbf{U}_{|F|})\}$$

is some set of *fairness constraints*, where each fairness constraint $(\mathbf{L}_i, \mathbf{U}_i)$, $i = 1, \ldots, |F|$, is a pair of (characteristic functions of) some subsets of states.

We say that a sequence of states $s_0, s_1, \ldots$ is a *run* of a sequence $\sigma_1 \sigma_2 \ldots \in \Sigma^\omega$ in the automaton $A = (S, \mathbf{I}, \mathbf{T}, F)$ if $\mathbf{T}(s_{i-1}, \sigma_i, s_i) = 1$ for all $i \geq 1$. Again, a run is

initialized if $\mathbf{I}(s_0) = 1$. Let $inf(s_0, s_1, \ldots)$ denote the set of states that occur infinitely often in $s_0, s_1, \ldots$, i.e. let:

$$inf(s_0, s_1, \ldots) = \{s \in S | \forall n \,.\, \exists m > n \,.\, s_m = s\} \ .$$

We say that a run $s_0, s_1, \ldots$ satisfies a fairness constraint $(\mathbf{L}, \mathbf{U})$ if either:

- $\mathbf{L}(s) = 0$ for all $s \in inf(s_0, s_1, \ldots)$, or
- $\mathbf{U}(s) = 1$ for some $s \in inf(s_0, s_1, \ldots)$.

A run if *fair* if it satisfies all fairness constraints in F. A run is accepting if it is initialized and fair. As in the finite case, the language of A (denoted by $\mathcal{L}(A)$) is the set of all sequences that have an accepting run in A.

Both automata on finite and infinite strings can be used to verify safety properties, but only automata on infinite strings can verify properties that cannot be disproved with finite strings (also called *liveness* properties). For example, assume that we want to verify that the output of the buffer always eventually follows the input, i.e. that whenever the input 1 at some time in the future the output is also 1. To disprove that property we need to show an infinite sequence in the behavior of the buffer where at some point the input is 1 and the output is 0 at all times thereafter. The automaton in Figure 2.3 accepts exactly those sequences, because the fairness constraint require that the "unfair" state is visited only finitely many times.

To interpret Figure 2.1 as an automaton on infinite sequences we need to augment it with fairness constraints. A natural requirement is that the buffer cannot stay in the transient states 01 and 10 forever. In other words, only runs that visit 00 or 11 infinitely often are fair, i.e.:

$$F = \{(1, (\mathbf{ps} = 00) + (\mathbf{ps} = 11))\} \ .$$

One can check that under these constraints the property in Figure 2.3 is satisfied.

2.3 Other approaches

In our approach the verification process consists of the following two steps:

1. the user constructs an automaton (say P) that accepts exactly those sequences that disprove the property,

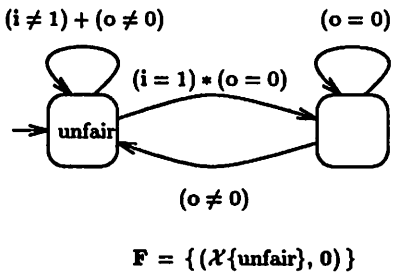


Figure 2.3: An automaton for a liveness property.

2. a tool checks whether $\mathcal{L}(P)$ intersects the language of the system to be verified.

Here, we differ slightly from the traditional approach (e.g. [VW86, Kur90]) where:

1. the user specify an automaton (say P) that accepts exactly those strings representing acceptable behavior, i.e. all strings that *do not* violate the property,

2. the tool check whether the language of the system (henceforth denoted by $\mathcal{L}(A)$) is contained in $\mathcal{L}(P)$.

In practice, the difference between two paradigms is mostly in interpretation, not in computation. The language containment is typically checked by first constructing an automaton P' that accepts sequences not in $\mathcal{L}(P)$, and then checking the emptiness of $\mathcal{L}(A) \cap \mathcal{L}(P')$. Constructing P' is a hard problem in general, but existing tools are restricted to special cases where constructing P' requires only simple syntactic modification of P. We will review three such approaches: one suggested by Vardi and Wolper [VW86], one by Kurshan [Kur90] on which the verification tool COSPAN [HK88] is based, and finally one by Hojati and Brayton [HB95] on which the tool HSIS [ABB+94] is based. In the following we assume that all definitions form section 2.2.2 are still valid except that of a fair run.

2.3.1 COSPAN approach

An L-process [Kur90] is a 5-tuple (S, $\mathbf{I}$, $\mathbf{T}$, R, F), where S, $\mathbf{I}$, $\mathbf{T}$, are as in previous definitions, $R \subseteq S \times S$ is some set of transitions called *recur edges*, and $F = \{Z_1, \ldots, Z_{|F|}\}$ is some set of subsets of states called *cycle sets*. A run $s_0, s_1, \ldots$ is fair in an L-process if:

- none of the recur edges appear in $s_0, s_1, \ldots$ infinitely often, *and*

- $inf(s_0, s_1, \ldots)$ is not contained in any of the cycle sets.

An L-automaton [Kur90] is syntactically the same as an L-process. However, a run is fair in an L-automaton if:

- some recur edge appears in $s_0, s_1, \ldots$ infinitely often, *or*
- $inf(s_0, s_1, \ldots)$ is contained in some of the cycle sets.

Obviously, a run is fair in an L-process (S, I, T, R, F) if and only is it is not fair in the L-automaton (S, I, T, R, F). Thus, if an L-automaton (S, I, T, R, F) is deterministic and complete, then its language is the complement of the language of the L-process (S, I, T, R, F).

Kurshan [Kur90] proposed to use L-processes to model systems and to use deterministic and complete L-automata to express properties. The language of the property is then complemented at no cost, by interpreting the corresponding deterministic and complete L-automaton as an L-process. The verification problem is thus reduced to checking language emptiness of the intersection of L-processes. This problem is essentially of the same complexity as the language emptiness of the intersection of Street automata [CDK89].

We can embed the COSPAN approach into ours. More precisely, we show that for any L-process $A = (S, \mathbf{I}, \mathbf{T}, R, F)$ there exists a Street automaton A' with $2|S|$ states and $|F| + 1$ fairness constraints, that has the same language as A. We construct A' as follows:[2]

- the set of states of A' is $S \times \{0, 1\}$,
- the set of initial states of A' is $\{(s, 0) \mid \mathbf{I}(s) = 1\}$,
- for all $s, q \in S$ and all $\sigma \in \Sigma$: $(s, x) \xrightarrow{\sigma} (q, y)$ is in in the transition relation of A' if and only if $\mathbf{T}(s, \sigma, q) = 1$, $x \in \{0, 1\}$ and:

$$y = \begin{cases} 1 & \text{if } (s, q) \in R \ , \\ 0 & \text{if } (s, q) \notin R \ , \end{cases}$$

- the set of fairness constraints of A' contains a constraint $(\mathcal{X}\{(s, 1) | s \in S\}, \mathbf{0})$ and one constraint of the form $(\mathbf{1}, \overline{\mathcal{X}(Z_i)})$ for every cycle set $Z_i \in F$.

[2]This construction is easily derived from the *node-recurring transform* of [Kur94].

The idea is to create a copy of the transition relation on two levels (0 and 1). Every recur edge leads to level 1 and every non-recur edge leads to level 0. Thus, some recur edge is crossed infinitely often if and only if some node on level 1 is visited infinitely often. It is easy to check that the languages of A and A' are the same.

2.3.2 HSIS approach

As we have seen, expressing recur edges as Streett fairness constraints may require doubling the state space. A more efficient alternative is to extended the language emptiness algorithm for Streett automata to handle recur edges directly. Therefore, Hojati and Brayton [HB95] suggested to use *edge-Streett automata* to model systems. An edge-Streett automaton is a 5-tuple (S, $\mathbf{I}$, $\mathbf{T}$, R, F), where R is a set of recur edges, and F is a set of Streett fairness constraints. A run is fair if it does not contain infinitely many recur edges, and it satisfies all the Streett constraints (in the sense of section 2.2.2).

To specify properties, Hojati and Brayton propose deterministic and complete *edge-Rabin automata*, which are dual to *edge-Streett automata* in the same sense as L-automata are dual to L-processes.

Formally, an edge-Rabin automaton is a 5-tuple (S, $\mathbf{I}$, $\mathbf{T}$, R, F), where R is a set of recur edges, and F is some set of *Rabin fairness constraints*, where each fairness constraint $(\mathbf{L}_i, \mathbf{U}_i)$, $i = 1, \ldots, |F|$ is a pair of (characteristic functions of) subsets of states.

We say that a run $s_0, s_1, \ldots$ satisfies a Rabin fairness constraint $(\mathbf{L}, \mathbf{U})$ if:

- $\mathbf{L}(s) = 1$ for some $s \in inf(s_0, s_1, \ldots)$, *and*
- $\mathbf{U}(s) = 0$ for all $s \in inf(s_0, s_1, \ldots)$.

A run is fair if it contains infinitely many recur edges or it satisfies some constraint in F. It is easy to see that a run is fair in an edge-Streett automaton if and only if it is not fair in the syntactically identical edge-Rabin automaton. Thus, complementing a deterministic and complete edge-Rabin automaton into an edge-Streett automaton does not require any computation.

2.3.3 Vardi-Wolper approach

A *Büchi automaton* [Büc60] is a 4-tuple (S, $\mathbf{I}$, $\mathbf{T}$, F), where F is some subset of states. A run $s_0, s_1, \ldots$ is fair if $inf(s_0, s_1, \ldots)$ intersects F. Vardi and Wolper [VW86]

suggested to model a system as a Büchi automaton (say A), and to model a property as another Büchi automaton (say P) which accepts all sequences that are models of some linear temporal logic formula ϕ. Checking whether all computation paths in A are models of ϕ reduces to checking whether $\mathcal{L}(A) \subseteq \mathcal{L}(P)$, which is then reduced to checking whether $\mathcal{L}(A)$ intersects $\mathcal{L}(P')$, where the Büchi automaton P' is constructed such that it accepts all sequences satisfying $\neg\phi$ (rather than complementing P directly). This construction may incur an exponential blow-up, but that is considered acceptable because linear temporal logic is a very succinct language for expressing properties (it can be shown that it is exponentially more succinct than any kind of automata we consider here [SV89]). Any Büchi automaton can be interpreted as a Streett automaton with a single fairness constraint $(\mathbf{1}, \mathcal{Y}(F))$. Thus, the approach by Vardi and Wolper can easily be embedded into our framework. Moreover, verifying the emptiness of the intersection of languages of Büchi automata is essentially of the same complexity as the language emptiness algorithm for Streett automata that we present. Hence, nothing is lost by moving to the more general framework.

Vardi and Wolper were primarily interested in model checking of linear temporal logic, so for their purposes Büchi automata suffice. However, Street automata are a better choice for the specification of systems and properties in the language containment framework because:

1. Deterministic Büchi automata are less expressive then non-deterministic Büchi and deterministic or non-deterministic Street or Rabin automata or L-processes which are all equally expressive and define the class of ω-regular languages.[3]

2. Even though non-deterministic Büchi automata are as expressive as Streett automata, they are exponentially less succinct, i.e. for every $n > 0$ there exists a language $\mathcal{L}_n$ which can be described by a Street automaton with $O(n)$ states, but cannot be described by any Büchi automaton with less than 2^n states [SV89].

In the rest of this work we will restrict our attention to the emptiness problem for the intersection of languages of Streett automata. As we have shown, this problem is general enough to include all language containment based verification approaches that have been considered. It's important to notice that this generality comes at no extra cost: special

[3]It is worth noting that deterministic L-automata are also less expressive than non-deterministic ones which can express any ω-regular language. However, every ω-regular language can be expressed as the intersection of languages of finitely many deterministic L-automata[Kur90]. Thus, it is possible to check any ω-regular property by finitely many containment checks of L-automata languages.

cases that have been proposed are of the same complexity as the general problem. Since no confusion can arise, we will use the term *"automata (on infinite sequences)"* to refer to Streett automata as defined in section 2.2.2. For the survey of language containment algorithms between different kinds of automata we refer the reader to [CDK89].

2.4 Operations on automata

2.4.1 Composition

In the previous section we have showed that the verification problem reduces to checking the emptiness of the intersection of languages of two automata. To reduce it further to the language emptiness check on a single automaton, we define the composition $\otimes$ of automata with the *language intersection property*:

$$\mathcal{L}(A \otimes B) = \mathcal{L}(A) \cap \mathcal{L}(B) \ .$$

The composition operation is also used to represent systems of interacting components. In this case every component is modeled as an automaton, and the whole system is modeled as a composition of component automata.

If $A = (S_A, \mathbf{I}_A, \mathbf{T}_A, \mathbf{F}_A)$ and $B = (S_B, \mathbf{I}_B, \mathbf{T}_B, \mathbf{F}_B)$ are two automata on finite strings, then their composition is defined by:

$$A \otimes B = (S_A \times S_B, \mathbf{I}_A * \mathbf{I}_B, \mathbf{T}_A * \mathbf{T}_B, \mathbf{F}_A * \mathbf{F}_B) \ .$$

Similarly, if $A = (S_A, \mathbf{I}_A, \mathbf{T}_A, F_A)$ and $B = (S_B, \mathbf{I}_B, \mathbf{T}_B, F_B)$ are two automata on infinite sequences, then their composition is defined by:

$$A \otimes B = (S_A \times S_B, \mathbf{I}_A * \mathbf{I}_B, \mathbf{T}_A * \mathbf{T}_B, F_A \cup F_B) \ .$$

In either case the present and next state variables of the composition are vectors $(\mathbf{ps}_A, \mathbf{ps}_B)$ and $(\mathbf{ns}_A, \mathbf{ns}_B)$ and the composition has the same I/O variable as A and B.

In other words:

- states of $A \otimes B$ are pairs, with one component being a state of A, and the other being a state of B,
- a state of $A \otimes B$ is initial if both of its components are,

- an edge $(s, s') \xrightarrow{\sigma} (q, q')$ is in the transition relation of $A \otimes B$ if $s \xrightarrow{\sigma} q$ and $s' \xrightarrow{\sigma} q'$ are in the transition relations of A and B respectively,
- in finite case, a sate of $A \otimes B$ is final if both of its components are,
- in infinite case, a run $(s_0, s'_0), (s_1, s'_1), \ldots$ is fair if $s_0, s_1, \ldots$ is fair in A, and $s'_0, s'_1, \ldots$ is fair in B.

Note, that due to our notation, we can easily interpret fairness constraints of A and B over the state space of $A \otimes B$. Thus, the fairness constraints of $A \otimes B$ is a simple union of constraints of A and B.

2.4.2 Union

It is useful to define a union $\oplus$ of automata that has the language union property:

$$\mathcal{L}(A \oplus B) = \mathcal{L}(A) \cup \mathcal{L}(B) \ .$$

In the definition of $A \oplus B$, we assume both A and B are complete. As we have shown in section 2.2.1 we can make this assumption without loss of generality.

If $A = (S_A, \mathbf{I}_A, \mathbf{T}_A, \mathbf{F}_A)$ and $B = (S_B, \mathbf{I}_B, \mathbf{T}_B, \mathbf{F}_B)$ are two automata on finite strings, then their union is defined by:

$$A \oplus B = (S_A \times S_B, \mathbf{I}_A * \mathbf{I}_B, \mathbf{T}_A * \mathbf{T}_B, \mathbf{F}_A + \mathbf{F}_B) \ .$$

In other words, $A \oplus B$ is the same as $A \otimes B$ except that a state of $A \oplus B$ is final if at least one of its components is.

In this work, we use union only on the automata on finite strings, but for completeness sake we also provide a definition for an infinite case: if $A = (S_A, \mathbf{I}_A, \mathbf{T}_A, F_A)$ and $B = (S_B, \mathbf{I}_B, \mathbf{T}_B, F_B)$ are two automata on infinite sequences, then their union is defined by:

$$A \oplus B = (S_A \times S_B, \mathbf{I}_A * \mathbf{I}_B, \mathbf{T}_A * \mathbf{T}_B, F_{A \oplus B}) \ ,$$

where:

$$F_{A \oplus B} = \{(\mathbf{L}_A * \mathbf{L}_B, \mathbf{U}_A + \mathbf{U}_B) | \, (\mathbf{L}_A, \mathbf{U}_A) \in F_A, (\mathbf{L}_B, \mathbf{U}_B) \in F_B\} \ .$$

One can check that (in both finite and infinite cases) runs in $A \oplus B$ are accepting if one of their components is.

2.4.3 Projection

Often, the I/O variable is a vector, some parts of which may be of interest as an observable behavior, and some parts of which are used only for internal coordination. To hide parts which are not of interest, we define the *projection* operation, both on languages and on automata. Given some language $\mathcal{L} \subseteq (\Sigma_1 \times \Sigma_2)^*$, a *projection* of $\mathcal{L}$ on Σ_1 (denoted by $\mathcal{L}|_{\Sigma_1}$) is defined by:

$$\mathcal{L}|_{\Sigma_1} = \{\sigma_{11}\ldots\sigma_{1n} \in \Sigma_1^* | \text{ there exists } (\sigma_{11},\sigma_{21})\ldots(\sigma_{1n},\sigma_{2n}) \in \mathcal{L} \subseteq (\Sigma_1 \times \Sigma_2)^*\} \ .$$

Similarly, in the infinite case the projection is defined by:

$$\mathcal{L}|_{\Sigma_1} = \{\sigma_{11}\sigma_{12}\ldots \in \Sigma_1^\omega | \text{ there exists } (\sigma_{11},\sigma_{21})(\sigma_{12},\sigma_{22})\ldots \in \mathcal{L} \subseteq (\Sigma_1 \times \Sigma_2)^\omega\} \ .$$

On the other hand, given some automaton $A = (S, \mathbf{I}, \mathbf{T}, \mathbf{F})$, with I/O vector (σ_1, σ_2) ranging over $\Sigma_1 \times \Sigma_2$, the *projection* of A on Σ_1 is an automaton with I/O variable σ_1, defined by:

$$A|_{\Sigma_1} = (S, \mathbf{I}, \exists\sigma_2 \,.\, \mathbf{T}, \mathbf{F}) \ ,$$

and similarly in the infinite case:

$$(S, \mathbf{I}, \mathbf{T}, F)|_{\Sigma_1} = (S, \mathbf{I}, \exists\sigma_2 \,.\, \mathbf{T}, F) \ .$$

It is straightforward to check that (both in finite and infinite cases):

$$\mathcal{L}(A|_{\Sigma_1}) = (\mathcal{L}(A))|_{\Sigma_1} \ .$$

For example, a projection of the automaton in Figure 2.1 to the domain of variable i is shown in Figure 2.4.

2.4.4 Quotient

Assume that an automaton $A = (S, \mathbf{I}, \mathbf{T}, \mathbf{F})$ and a partition[4] $\rho = \{S_1, \ldots, S_{|\rho|}\}$ of S are given. The *quotient* of A with respect to ρ is defined by:

$$A_{/\rho} = (\rho,\ \mathcal{X}\{S_i \,|\, \mathbf{I}(s) = 1 \text{ for some } s \in S_i\},\ \mathbf{T}_{A_{/\rho}},\ \mathcal{X}\{S_i \,|\, \mathbf{F}(s) = 1 \text{ for some } s \in S_i\}) \ ,$$

where:

$$\mathbf{T}_{A_{/\rho}} = \mathcal{X}\{(S_i, \sigma, S_j) \,|\, \mathbf{T}(s,\sigma,q) = 1 \text{ for some } s \in S_i,\ q \in S_j\}, \ .$$

[4]The set $\rho = \{S_1, \ldots, S_{|\rho|}\} \subseteq 2^S$ is a *partition* of S if $S = \bigcup_{i=1}^{|\rho|} S_i$, and $S_i \cap S_j = \emptyset$ for all $i \neq j$.

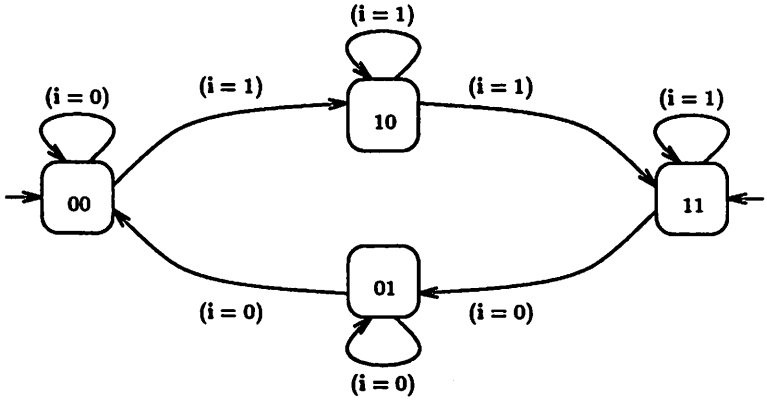


Figure 2.4: An illustration of projection.

We think of states of $A_{/\rho}$ as abstract states obtained by merging concrete states of A. An abstract state is initial (final) if it contains concrete initial (final) states. If σ enables the transition between two concrete states, then it also enables the transitions between abstract states containing them. Therefore, if $s_0, s_1, \ldots, s_n$ is an accepting run of $\sigma_1 \ldots \sigma_n$ in A, we can easily construct a run $S_0, S_1, \ldots, S_n$ of $\sigma_1 \ldots \sigma_n$ in $A_{/\rho}$, by choosing S_i such that $s_i \in S_i$. It follows that:

$$\mathcal{L}(A) \subseteq \mathcal{L}(A_{/\rho}) \ . \tag{2.1}$$

We say that some $S_i \in \rho$ is *reachable* if some state $s \in S_i$ is reachable from some of the initial states in A. We say that $S_i \in \rho$ is *stable* with respect to the transition relation $\mathbf{T}$ if for every I/O value σ and all $S_j \in \rho$:

$$\exists s_i \in S_i \,.\, \exists s_j \in S_j \,.\, \mathbf{T}(s_i, \sigma, s_j) = 1 \implies \forall s_i' \in S_i \,.\, \exists s_j' \in S_j \,.\, \mathbf{T}(s_i', \sigma, s_j') = 1 \ . \tag{2.2}$$

Finally, if every reachable $S_i \in \rho$ is stable with respect to $\mathbf{T}$, then we say so for ρ as well. For example, in Figure 2.5 class S_2 is stable (only states in S_1 are reachable from states in S_2, and all states in S_2 can reach some state in S_1), while class S_1 is not, because s_{11} can reach a state in S_1 (namely s_{22}) while s_{22} cannot.

If $S_0, S_1, \ldots, S_n$ is an accepting run of $\sigma_1 \ldots \sigma_n$ in $A_{/\rho}$, and ρ is stable with respect to $\mathbf{T}$ then we can construct a run $s_0, s_1, \ldots, s_n$ of $\sigma_1 \ldots \sigma_n$ in A:

1. choosing some initial state $s_0 \in S_0$,

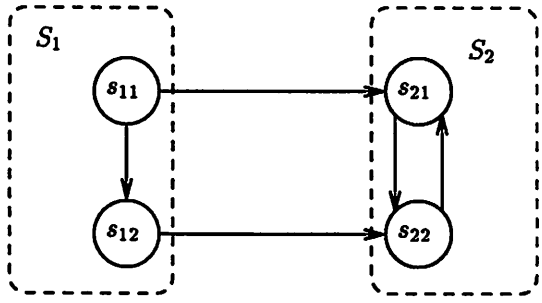


Figure 2.5: An illustration of stability.

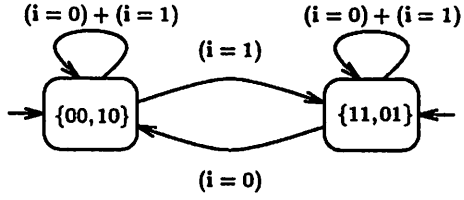


Figure 2.6: An illustration of the quotient construction.

2. choosing some $s_i \in S_i$ such that $\mathbf{T}(s_{i-1}, \sigma, s_i) = 1$, for all $i = 1, \ldots, n$.

The condition (2.2) guarantees that the choice above is always possible. Also, if ρ respects final states (i.e. if for every $Q \in \rho$ either $Q \subseteq \mathcal{S}(\mathbf{F})$ or $Q \cap \mathcal{S}(\mathbf{F}) = \emptyset$), then we know that s_n is a final state, and that $s_0, s_1, \ldots, s_n$ is indeed an accepting run of $\sigma_1 \ldots \sigma_n$ in A. Thus, if ρ is stable with respect to $\mathbf{T}$ and respects final states, then we can strengthen (2.1) to:

$$\mathcal{L}(A) = \mathcal{L}(A_{/\rho}) \ . \tag{2.3}$$

For example, the quotient of the automaton in Figure 2.4 with respect to the partition $\rho = \{\{00, 10\}, \{11, 01\}\}$ is shown in Figure 2.6. The partition ρ is not stable, and indeed (2.3) does not hold, as demonstrated by the string 010 which is in the language of the quotient, but not in the language of the original automaton.

Every equivalence relation induces a partition of its domain. Given an equivalence relation $\sim$ on the state space of some automaton $A = (S, \mathbf{I}, \mathbf{T}, \mathbf{F})$, we use $A_{/\sim}$ to denote the quotient of A with respect to the partition induced by $\sim$.

2.5 Traversing automata

The most basic operation of automata traversal is computing the set of successors of a given set of states. For this purpose we define the *post* operator, which comes in binary and ternary forms.

The ternary *post* operator takes as operands a characteristic function $\mathbf{S}$ (with support $\{\mathbf{ps}\}$) of some set of states, a transition relation $\mathbf{T}$ (with support $\{\mathbf{ps}, \sigma, \mathbf{ns}\}$), and an I/O value σ, and returns a characteristic function of the set $\{q | \mathbf{T}(s, \sigma, q) = 1$ for some $s \in \mathcal{S}(\mathbf{S})\}$ (also with support $\{\mathbf{ps}\}$). Formally, it is defined by:

$$post(\mathbf{S}, \mathbf{T}, \sigma) \equiv [\exists \mathbf{ps} \,.\, \exists \sigma \,.\, (\mathbf{S} * \mathbf{T} * (\sigma = \sigma))]_{\mathbf{ns} \rightarrow \mathbf{ps}} \ .$$

In many cases, we are only interested whether there exists a transition between two states, and do not care which I/O values enable that transition. For that analysis, we first compute:

$$\mathbf{G} \equiv \exists \sigma \,.\, \mathbf{T} \ ,$$

where $\mathbf{T}$ is a transition relation of some automata. We say that $\mathbf{G}$ is the *graph* of the automaton, because it can be interpreted as a graph with nodes being states, and an edge between any two states s and q satisfying $\mathbf{G}(s, q) = 1$. We can explore the graph using the binary post operator defined by:

$$post(\mathbf{S}, \mathbf{G}) \equiv [\exists \mathbf{ps} \,.\, (\mathbf{S} * \mathbf{G})]_{\mathbf{ns} \rightarrow \mathbf{ps}} \ .$$

The operator $post(\mathbf{S}, \mathbf{G})$ computes the characteristic function of immediate successors of states in $\mathbf{S}$. Usually, of more interest is the operator $post^*$ which computes the characteristic function of states that can be reached in any (i.e. zero or more) number of steps. It is defined recursively as follows:

$$post^*(\mathbf{S}, \mathbf{G}) \equiv \begin{cases} \mathbf{S} & \text{if } \mathcal{S}(post(\mathbf{S}, \mathbf{G})) \subseteq \mathcal{S}(\mathbf{S}) \ , \\ post^*(\mathbf{S} + post(\mathbf{S}, \mathbf{G}), \mathbf{G}) & \text{otherwise.} \end{cases}$$

For example, in Figure 2.7 computing $post^*$ with $\{3\}$ as an initial set, would return $\{2, 3, 4, 5\}$, with $\{2, 3, 4\}$ as an intermediate result.

Operators, *post* and $post^*$ are used to traverse the graph forwards, in the direction of the transitions. Their backwards duals are operators *pre* and pre^* defined as follows:

$$pre(\mathbf{S}, \mathbf{G}) \equiv \exists \mathbf{ns} \,.\, ([\mathbf{S}]_{\mathbf{ps} \rightarrow \mathbf{ns}} * \mathbf{G}) \ ,$$

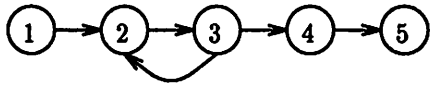


Figure 2.7: A graph used to illustrate traversal operators.

$$pre^*(\mathbf{S},\mathbf{G}) \equiv \begin{cases} \mathbf{S} & \text{if } \mathcal{S}(pre(\mathbf{S},\mathbf{G})) \subseteq \mathcal{S}(\mathbf{S}) \ , \\ pre^*(\mathbf{S} + pre(\mathbf{S},\mathbf{G}),\mathbf{G}) & \text{otherwise.} \end{cases}$$

The operator *pre* computes a characteristic function of states that can reach some state in S in exactly one transition step, and pre^* computes a characteristic function of states that can reach some state in S in zero or more steps. For example, in Figure 2.7 computing pre^* with $\{3\}$ as an initial set, would return $\{1,2,3\}$, with $\{2,3\}$ as an intermediate result.

To eliminate from some set all the states which cannot reach any cycle in that set we use the *cycle* operator defined by:

$$cycle(\mathbf{S},\mathbf{G}) \equiv \begin{cases} \mathbf{S} & \text{if } \mathcal{S}(\mathbf{S}) \subseteq \mathcal{S}(pre(\mathbf{S},\mathbf{G})) \ , \\ cycle(\mathbf{S} * pre(\mathbf{S},\mathbf{G}),\mathbf{G}) & \text{otherwise.} \end{cases}$$

In every recursive call, the first argument characterizes smaller and smaller sets. More precisely $\mathbf{S} * pre(\mathbf{S},\mathbf{G})$ characterizes the set obtained from the set (characterized by) S by removing from it all the state that do not have an immediate successor in S. Repeating this process will eventually remove from S all the states that cannot reach any loop in S. For example, in Figure 2.7 computing *cycle* with $\{1,2,3,4,5\}$ as an initial set, would return $\{1,2,3\}$, with $\{1,2,3,4\}$ as an intermediate result.

2.5.1 Efficiency of traversal

Operators pre^*, $post^*$, and *cycle* are examples of *fixed-point* operators. Their recursive definitions provide a straightforward way of implementing them. In that case, the first arguments in the sequence of recursive calls characterize strictly increasing (in case of pre^* and $post^*$) or strictly decreasing (in case of *cycle*) sequence of subsets of states. Thus, the number of recursive calls is limited by the number of states, and if this number is finite, the recursion will always terminate. This argument is often used to claim that the straightforward implementations of pre^*, $post^*$, and *cycle* are of linear time complexity in the number of state. This would be true if the operators *pre*, *post*, $*$, $+$, and $\subseteq$ were

of constant time complexity. But, in BDD based implementation each of these operation require fixed number of BDD disjunctions, conjunctions and/or existential quantification (set inclusion $\mathcal{S}(\mathbf{F}) \subseteq \mathcal{S}(\mathbf{G})$ can be checked by checking the equivalence of $\mathbf{F} * \mathbf{G}$ and $\mathbf{G}$). The size of a transition relation BDD (which has the largest support of all BDD's involved) is limited by the square of the number of states, so the overall worst case complexity is proportional to the cube of the number of states.

In practice, this bound is much too high for most systems of interest. For BDD based techniques to yield any advantages over standard methods, the following two conditions must be met:

1. The number of nodes in the BDD's must be much smaller than the number of reachable states. This condition is often met for system consisting of components communicating a limited amount of information (for detailed analysis see [McM93]).

2. The number of recursive calls must also be much smaller than the number of reachable states. Fortunately, this condition is often met, because the number of recursive calls is limited by the depth[5] of the graph of the system, which is typically much smaller than the number of reachable states. However, there is one notable exception: counters, where computing the pre^* and $post^*$ operators from any single state requires the number of recursive calls which is equal to the number of reachable states. It is thus not surprising that some researchers have devoted special attention to abstracting the behavior of counters (e.g. [MPS92]).

2.6 Language emptiness algorithm

The basic steps in checking language emptiness are the same both for automata on finite and infinite sequences. They are outlined in Figure 2.8. First, we compute the graph of the system (step 1), and a characteristic function of states reachable from the initial states (step 2). Then, in step 3 we compute a characteristic function $\mathbf{A}$ of some subset of states appearing in accepting runs. In the case of finite strings, the accept($\mathbf{R}$, $\mathbf{G}$, $\mathbf{F}$) simply returns $\mathbf{R} * \mathbf{F}$, the characteristic function of final states reachable from the initial states. If that set is empty, then so is the language, and verify returns $NULL$. Otherwise, it returns

[5]The depth of a graph is the length of maximum shortest path between any two states.

```
        function verify (A)
          /* A = (S, I, T, F) - an automaton on finite strings, or */
          /* A = (S, I, T, F) - an automaton on infinite sequences */
          /* ps, ns, σ - the present, next and I/O variables of A */
step 1:   G := ∃σ . T;
step 2:   R := post*(I, G);
step 3:   A := accept(R, G, F or F);
step 4:   if A ≡ 0 then return NULL;
step 5:   else          return debug(A, I, G);
        end
```

Figure 2.8: LE – language emptiness algorithm

one path from some of the initial state to some of the final states. Such a path is computed by the **debug** function.

For the infinite case, we present the **accept** function similar to the algorithm by Hojati and Brayton [HB95], which in turn can be viewed as an implementation of a fixed point formula by Emerson and Lei [EL86]. But first we need to introduce the notion of a *fair cycle*. A cycle $s_1 \rightarrow s_2 \rightarrow \ldots \rightarrow s_n \rightarrow s_1$ is *fair* if for every fairness constraints $(\mathbf{L}, \mathbf{U}) \in F$ either $\mathbf{L}(s_i) = 0$ for all $i = 1, \ldots, n$, or $\mathbf{U}(s_i) = 1$ for some $i = 1, \ldots, n$. In other words, a cycle is fair if the run generated by traversing the cycle indefinitely is fair.

The existence of a fair cycle that is reachable from some of the initial states is obviously sufficient for the non-emptiness of the language of some automaton. It is perhaps less obvious that for finite-state automata it is also necessary. To see that, consider some accepting run $s_0, s_1, \ldots$, and a finite subsequence $s_n, s_{n+1}, \ldots, s_m$ of that run satisfying the following:

- $s_i \in inf(s_0, s_1, \ldots)$ for all $i \geq n$ (this requirement is always satisfied for sufficiently large n)
- $s_{m+1} = s_n$, and for every $s \in inf(s_0, s_1, \ldots)$ there exists at least one $i \in \{n, \ldots, m\}$ such that $s = s_i$ (if $inf(s_0, s_1, \ldots)$ is finite, then for any n one can choose a sufficiently large m that satisfies this requirement).

```
        function accept(R, G, F)
          /* R - the characteristic function of reachable states */
          /* G - the graph of the automaton */
          /* F = {(L_1, U_1), ..., (L_|F|, U_|F|)} - fairness constraints */
step 1:   Aux := R;
          while TRUE
step 2:     Fair = cycle(Aux, G);
step 3:     for i = 1, ..., |F|
step 4:       Fair := Fair * (¬L_i + pre*(U_i * Fair, G));
            end for
            if Fair ≡ Aux then
step 5:       return Fair;
            else
step 6:       Aux = Fair;
            end if
          end while
        end
```

Figure 2.9: Finding states appearing in fair cycles.

The sequence $s_n, s_{n+1}, \ldots, s_m, s_{m+1}$ forms a loop which is fair and reachable from the initial states (via $s_0, s_1, \ldots, s_n$). Thus, the search for an infinite accepting run is reduced to a search for a finite fair cycle reachable from some of the initial states.

The accept function for automata on infinite sequences is shown is Figure 2.9. At all times it maintains a superset **Fair** of the set of reachable states that appear in some fair cycle. Initially, **Fair** includes all the reachable states. In step 2 we remove from **Fair** all states that cannot reach a cycle in it (and thus cannot participate in any fair cycle). In step 4 we remove from **Fair** all states that are in $\mathcal{S}(\mathbf{L}_i)$, and from which no states in $\mathcal{S}(\mathbf{Fair} * \mathbf{U}_i)$ are reachable. Consider one such a state, say s. Since $\mathbf{L}_i(s) = 1$, the fair cycle that includes s, must visit some state in $\mathcal{S}(\mathbf{U}_i)$. But, all states reachable from s are either not in $\mathcal{S}(\mathbf{U}_i)$, or have been already eliminated as candidates for fair cycles, thus there can

be no fair cycle including s. We iterate this process until we reach the fixed point, i.e. until no more states can be removed from **Fair**. We claim that at this time **Fair** $\equiv$ **0** if and only if the language of the automaton is empty. More precisely, we claim that if **Fair** is a characteristic function returned by **accept**, then the following holds:

1. if some state s appears in some fair cycle reachable from the initial states, then $s \in \mathcal{S}(\mathbf{Fair})$,

2. if none of the cycles reachable from some state s are fair, then $s \notin \mathcal{S}(\mathbf{Fair})$.

Consider some fair reachable cycle. The states in that cycle are obviously reachable (thus in initial **Fair**), and they cannot be deleted either in step 2 or step 4. Thus, the first claim holds.

To see that the second claim holds, assume to the contrary that there exists a state s in the final **Fair**, such that none of the cycles reachable from s are fair. Consider now the graph of the system restricted to the states in final **Fair**, and in particular consider some leaf strongly connected component (say L) of that graph reachable from s (since s is not deleted in step 2, it must be possible to reach some cycle in **Fair** from s, thus also some leaf strongly connected component). States in L can reach only other states in L and none of the states in L are eliminated in step 4, thus for all $i = 1, \ldots, |F|$:

- either none of the states in L are in $\mathcal{S}(\mathbf{L}_i)$, or
- some states in L are in $\mathcal{S}(\mathbf{U}_i)$.

Thus, a cycle that visits all states in L is a fair cycle reachable from s, contradicting our assumption that none of the cycles reachable from s are fair.

If the **accept** function returns a characteristic function of some non-empty set, then the **debug** function is executed, which selects one fair cycle from the set returned by **accept**, and construct one path to that cycle from one of the initial states. The path and the cycle form an accepting run, and thus are the proof that the language is not empty.

The number of passes through the **while** loop in Figure 2.9 is limited by the number of states in $\mathcal{S}(\mathbf{R})$, because the loop is repeated only if at least on state is eliminated from **Fair** in step 2 or 4. Thus, the overall algorithm runs in polynomial time in the number of states.

Chapter 3

Networks of Communicating Automata

In this chapter we specialize the generic iterative algorithm of section 1.3 to networks of communicating automata. Initially, we ignore all the communication between the subsystems, and then iteratively and selectively restore it, if the current abstraction proves to be too simplistic. We show that the procedure will terminate, and we formulate several open problems that could improve efficiency of the procedure. Finally, we present and discuss some initial experimental results.

Although ideas similar to those presented here could easily be applied to any formalism allowing conservative abstractions and any verification algorithm, the effectiveness is dependent on the fact that inside the iteration loop we are using a BDD-based language-containment tool. In particular, heuristic abstractions described in this chapter often result in a system with more reachable states, but smaller BDD representations.

3.1 Abstractions

We address the problem of checking the emptiness of the language of an automaton given as a composition:

$$A = A_0 \otimes A_1 \otimes \ldots \otimes A_n, \tag{3.1}$$

where $A_i = (S_i, \mathbf{I}_i, \mathbf{T}_i, F_i)$ are automata on infinite sequences with the present and next state variables $\mathbf{ps}_i$ and $\mathbf{ns}_i$. Let A_0 be a distinguished *task* automaton, and assume that the

I/O variable σ is actually a vector $\sigma = (\sigma_1, \ldots, \sigma_n)$. We think of σ_i as the output variable of the i-th component.

Our goal is to avoid forming A. The challenge is to find an automaton B which is small, satisfies:

$$\mathcal{L}(A) \subseteq \mathcal{L}(B) \tag{3.2}$$

and yet close enough to A such that its language is empty if $\mathcal{L}(A)$ is.

One way of generating B that is likely to be[1] smaller than A and satisfies (3.2) is to compute the partial product $\bigotimes_{i \in I} A_i$ for some $I \subset \{0, \ldots, n\}$. We say that automata in I are active and others are ignored. An equivalent interpretation of this abstraction is that we have replaced every automaton A_j, $j \notin I$ with an automaton that has the same states but unrestricted transition between any two states (characteristic function of the transition relation equals 1), all states designated as initial (characteristic function of initial states equals 1) and no fairness constraints. The following lemma states that this is indeed the conservative simplification.

Lemma 3.1 *Let $A_0, \ldots, A_n$ be automata and let $I \subset \{0, \ldots, n\}$. Then:*

$$\mathcal{L}(\bigotimes_{i=0}^{n} A_i) \subseteq \mathcal{L}(\bigotimes_{i \in I} A_i).$$

Proof. $\mathcal{L}(\bigotimes_{i=0}^{n} A_i) = \mathcal{L}(\bigotimes_{i \in I} A_i) \cap \mathcal{L}(\bigotimes_{i \notin I} A_i) \subseteq \mathcal{L}(\bigotimes_{i \in I} A_i)$. □

The second kind of abstractions we use is closely related to the language emptiness algorithm. Recall that step 1 of the LE algorithm calls for computing the graph of the system, which in this case requires computing:

$$\mathbf{G} = \exists \sigma \,.\, (\prod_{i=0}^{n} \mathbf{T}_i) \;. \tag{3.3}$$

The following proposition provides for abstraction at the graph level.

Proposition 3.1 *Let* $\mathbf{G}$ *be the graph of a system, and let a graph* $\mathbf{H}$ *be such that:*

$$\mathcal{S}(\mathbf{G}) = \{(s, q) \,|\, \mathbf{G}(s, q) = 1\} \subseteq \mathcal{S}(\mathbf{H}) \;.$$

Then, the absence of a fair run in (the graph characterized by) $\mathbf{H}$ *implies the absence of a fair run in* $\mathbf{G}$.

[1] In all examples where we tried this, it did result in a smaller BDD representation. However, it is not true in the general case. In fact, it is very easy to construct degenerate examples where this is not the case.

Proof. Since every edge in **G** appears also in **H**, so must also every run, and in particular every fair run. □

The following result is a well known fact in Boolean theory, so we state it here without a proof. It offers a convenient way to compute an abstraction satisfying the condition in Proposition 3.1.

Lemma 3.2 *Let* **G** *be as defined by* (3.3) *and let:*

$$\mathbf{H} = \prod_{i=0}^{n} (\exists \sigma \,.\, \mathbf{T}_i) \ . \tag{3.4}$$

Then, $\mathcal{S}(\mathbf{G}) \subseteq \mathcal{S}(\mathbf{H})$.

We can interpret this abstraction as ignoring the communication between subsystems, since we remove enabling conditions from transitions before checking whether they can be simultaneously satisfied. The heuristic argument for computing (3.4) instead of (3.3) is that all the intermediate results in (3.4) have fewer variables in its support than the intermediate result $\prod_{i=0}^{n} \mathbf{T}_i$ in (3.3). This is likely to reduce the size of the intermediate BDD's. In fact the intermediate result $\prod_{i=0}^{n} \mathbf{T}_i$ is often the single largest BDD created throughout the language containment algorithm.

The second heuristic argument follows from the well known fact that the BDD size of the product of two Boolean functions is bounded from above by the product of the sizes of BDD's representing each function, but if they have disjoint supports the bound can be strengthened to the sum of the sizes of BDD's. This suggests that the BDD representing (3.4) is likely to be smaller than the BDD representing (3.3), since intermediate results $(\exists \sigma \,.\, \mathbf{T}_i)$ in (3.4) have disjoint support.

As is the case with other proposed abstractions, these heuristic arguments do not hold in general, and counter-examples are easily constructed. However, our (admittedly limited) experience supports them without exceptions.

It is interesting to note that both of these abstractions result in a graph with the same number of nodes as the original one, but with more edges, therefore possibly more reachable states. So, these abstractions are not at all suitable for a verification tool based on explicit state enumeration, but they should result in a smaller BDD representation in most cases.

3.2 Algorithm

Figure 3.1 shows the iterative algorithm we propose for checking language emptiness of networks of communicating automata (henceforth CLE algorithm, for Compositional Language Emptiness). It consists of the main function verify_comp which in turn calls the function verify of section 2.6. We assume that the system to be verified consists of components $A_i = (S_i, \mathbf{I}_i, \mathbf{T}_i, F_i)$, for $i = 0, \ldots, n$, and that the failure trace returned by verify is a vector $r = (r_0, \ldots, r_n)$, where each $r_i = r_{i,1} r_{i,2} \ldots$ is an infinite sequence of states in S_i. The following definitions are used in the CLE algorithm:

$$\mathbf{L}_{I,k} = \prod_{i \in I} \exists \mathbf{ps}_i . \exists \mathbf{ns}_i . (\mathbf{T}_i * (\mathbf{ps}_i = r_{i,k}) * (\mathbf{ns}_i = r_{i,k+1})) \ , \tag{3.5}$$

$$\mathbf{E}_{I,k} = \prod_{i \in I} (\mathbf{ps}_i = r_{i,k}) * (\mathbf{ns}_i = r_{i,k+1}) \ , \tag{3.6}$$

$$\mathcal{L}_r = \{\sigma \in \Sigma^\omega | r \text{ is a run of } \sigma \text{ in } \bigotimes_{i \in Act} A_i\} \ , \tag{3.7}$$

where I is some subset of $\{0, \ldots, n\}$. Intuitively, $\mathbf{L}_{I,k}$ is the characteristic function of I/O values that enable the k-th edge of the failure report. $\mathbf{E}_{I,k}$ is the characteristic function of the k-th edge of the failure report, restricted to components in I, and $\mathcal{L}_r$ is set of all I/O sequence for which r is a run in all active components.

If $\mathcal{L}(\bigotimes_{i=0}^{n} A_i)$ is empty, then the function verify_comp returns $NULL$, otherwise it returns a run of some sequence in $\mathcal{L}(\bigotimes_{i=0}^{n} A_i)$. In the following paragraphs we identify in verify_comp the four phases of the generic iterative abstraction algorithm of section 1.3, and give the intuition behind the heuristics built in the CLE algorithm.

3.2.1 Initial abstraction

We build an initial abstraction in steps 1 and 2 of the CLE algorithm. Our choice of the initial abstraction is based on the observation that many of the interesting properties are expressed in terms of the output variables of only a few subsystems, and that for some of the properties the behavior of some subsystems is irrelevant. Therefore, as an initial abstraction, in step 2 of the CLE algorithm, we ignore all the subsystems that are not in the Act set, i.e. all except A_0 and those automata that have their output variables in the support of the transition relation of A_0. By choosing this initial abstraction and an appropriate refinement step, we will never consider a subsystem that is irrelevant to the property to be verified.

function verify_comp $(A_0, \ldots, A_n)$

/* $A_i = (S_i, \mathbf{I}_i, \mathbf{T}_i, F_i)$ - components of the system to be verified */

step 1: $Act := \{i | i = 0 \text{ or } \sigma_i \in supp(\mathbf{T}_0)\}$;

step 2: $B := (S_0 \times \ldots \times S_n, \prod_{i \in Act} \mathbf{I}_i, (\exists \sigma . \prod_{i \in Act} \mathbf{T}_i), \bigcup_{i \in Act} F_i)$;

while $TRUE$ **do**

step 3: **if** $(r :=$verify$(B)) = NULL$ **then**

step 4: **return** $NULL$; /* the task is verified */

step 5: **else if** $\mathbf{L}_{I,k} \equiv \mathbf{0}$ for some $I \subseteq Act, k \geq 1$ **then**

step 6: $B := (S_B, \mathbf{I}_B, \mathbf{T}_B * \overline{\mathbf{E}_{I,k}}, F_B)$;

step 7: **else if** $\mathcal{L}_r \cap \mathcal{L}(\bigotimes_{k \in New} A_k) = \emptyset$ for some $New \subseteq \overline{Act}$ **then**

step 8: $C := \bigotimes_{k \in New} A_k$;

step 9: $B := (S_B, \mathbf{I}_B * I_C, \mathbf{T}_B * (\exists \sigma . T_C), F_B \cup F_C)$;

step 10: $Act := Act \cup New$;

else

step 11: **return** r; /* the task is not verified */

end if

end while

end

Figure 3.1: CLE – compositional language emptiness algorithm

3.2.2 Verification

Recall that the function verify of section 2.6 takes an automaton (say A) as input, and it either returns a run of some sequence in $\mathcal{L}(A)$, or it returns $NULL$ if $\mathcal{L}(A)$ is empty.

We stress the importance for our algorithm of the failure report returned by verify. If the task is not satisfied, then a "real" failure is always present, even in the first iteration. However, verify might ignore that failure for many iterations, reporting instead "false" failures. Unfortunately, there are no known heuristics (let alone exact procedures) which would guide the tool in the search of a "real" failure.

3.2.3 Failure analysis

The exact task of the failure analysis step is determined by the failure report and the types of abstraction used. As indicated previously, there are two abstractions used: ignoring communications (step 9 of the CLE algorithm) and ignoring subsystems (step 2 of the CLE algorithm). In our algorithm we first analyze the former in step 5, and then, if no violations are found, we analyze the latter in step 7.

If communication between subsystems is ignored while making the product machine, then the result is a graph with the same nodes and more edges than the exact graph. Since initial states and fairness constraints are exact, one only needs to check that every edge in the report is a "real" edge, i.e. that the enabling conditions of all the active component edges can be simultaneously satisfied.

The function $\mathbf{L}_{Act,k}$ is satisfied by exactly those $\sigma \in \Sigma$ that enable the k-th transition of the failure report in all active automata. Thus, r is an accepting run of some sequence in $\bigotimes_{i \in Act} A_i$, if and only if for all $k \geq 1$: $\mathbf{L}_{Act,k} \not\equiv \mathbf{0}$. Note that only finitely many initial k's need to be checked, because r is ultimately periodic.[2]

Once we find a k such that $\mathbf{L}_{Act,k} \equiv \mathbf{0}$, we could terminate the failure analysis and eliminate the failure by deleting the k-th edge of the failure report from the current graph of the system. However, we prefer to find a small subset I of Act such that $\mathbf{L}_{I,k} \equiv \mathbf{0}$, for at least two reasons:

1. If I is smaller than Act than $\mathbf{E}_{I,k}$ is likely to have a smaller BDD representation than $\mathbf{E}_{Act,k}$, because it takes less BDD operations to construct it.

[2] A sequence $s_0, s_1, \ldots$ is ultimately periodic if there exists integers n and p such that for all $i \geq n$: $s_{i+p} = s_i$. In other words, a run is ultimately periodic if it consists of a cycle and a path to that cycle.

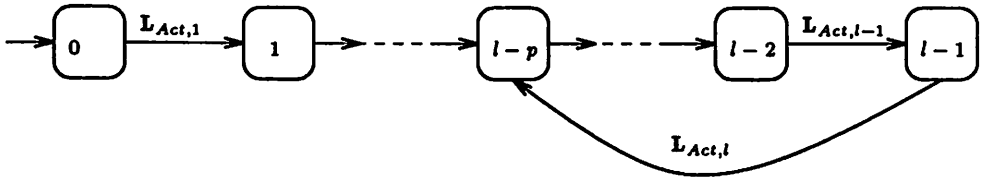


Figure 3.2: Construction of the X automaton.

2. The smaller I is, the larger is the set of edges $\mathbf{E}_{I,k}$ characterizes. If we eliminate all those edges at once, we can speed-up the convergence of the CLE algorithm.

We propose a two-step method of finding a suitable I. First, we evaluate $\mathbf{L}_{I,k}$ iteratively for increasing subsets of Act, starting with a singleton, adding one new element in each iteration, and stopping as soon as $\mathbf{L}_{I,k} \equiv \mathbf{0}$. Say that this happens for some $I' \subseteq Act$. Then, we try as I all two-element subsets of I'. If one of those evaluates $\mathbf{L}_{I,k}$ to $\mathbf{0}$ we use it in computing $\mathbf{E}_{I,k}$, otherwise we use I'. In principle, one could then try all triples, quadruples, etc., but we conjecture that it would actually increase total running time.

If this phase of failure analysis reveals no over-simplifications, we move to the next one in step 7 of the CLE algorithm. To check whether $\mathcal{L}_r \cap \mathcal{L}(\bigotimes_{k \in New} A_k) = \emptyset$ we first construct an automaton X such that $\mathcal{L}(X) = \mathcal{L}_r$, and then execute verify$(X \otimes (\bigotimes_{i \in New} A_i))$ for some candidate set New.

Assuming that the failure report r is of length l and has a period p (i.e. $\forall i \,.\, \forall j > l \,.\, r_{i,j} = r_{i,j-p}$), we define the automaton X by:

$$X = (\{0, \ldots, l-1\},\ \mathcal{X}\{0\},\ \mathbf{T}_X,\ \emptyset)\} \ ,$$

where $\mathbf{T}_X(k, \sigma, m) = 1$ if and only if $\mathbf{L}_{Act,k+1}(\sigma) = 1$, and:

$$m = \begin{cases} l-p & \text{if } k = l-1 \ , \\ k+1 & \text{otherwise.} \end{cases}$$

The construction of X is illustrated if Figure 3.2. Note that while constructing X, we can re-use $\mathbf{L}_{Act,k}$ computed in step 5.

If a candidate New is chosen too big, the check in step 7 may be as complex as the original problem. We propose the following heuristics to avoid this complexity:[3]

[3]The choice of these heuristics was influenced by discussions with Robert Kurshan.

1. first, try to find a single component A_i such that $\mathcal{L}(X) \cap \mathcal{L}(A_i) = \emptyset$,
2. if that fails, order ignored components in a way that every component has at least one variable in common support either with X or with some component preceding it,
3. compute cumulative product $X \otimes A_{i_1} \otimes \ldots$ adding one ignored component at a time in the order chosen in the previous step,
4. repeat the previous step until one of the following conditions is satisfied:
   (a) *the language of the cumulative product becomes empty*: in this case let New contain indexes of all components in the cumulative product,
   (b) *the cumulative product contains all ignored components and its language is not empty*: in this case terminate with failure,
   (c) *the cumulative product is too big (i.e. occupied memory exceeds some given limit)*: in this case let New contain indexes of all components in the cumulative product (we defer eliminating the failure report until subsequent iterations).

3.2.4 Refinement

Depending on the results of failure analysis there are two different refinement problems: deleting certain edges from the current graph (step 6) and including previously ignored subsystems into the current abstraction (step 8–10 of the CLE algorithm).

Deleting edges in step 6 is justified by the following proposition:

Proposition 3.2 *If $I \subseteq Act$ is such that $\mathbf{L}_{I,k} \equiv \mathbf{0}$, for some k, then:*

a) $\mathbf{E}_{I,k}$ *characterizes some edges in the graph of the current abstraction,*

b) *none of the edges characterized by $\mathbf{E}_{I,k}$ appear in the exact graph of the system.*

Proof. The k-th edge of the failure report must be both in the current graph of the system and in $\mathcal{S}(\mathbf{E}_{I,k})$, thus part a) holds. To show part b) assume by contradiction that there exists an edge in the exact graph with components $r_{i,k} \rightarrow r_{i,k+1}$ for all $i \in I$. Then, there must exist at least one $\sigma \in \Sigma$ such that $\mathbf{T}_i(r_{i,k}, \sigma, r_{i,k+1}) = 1$ for all $i \in I$, which contradicts the assumption $\mathbf{L}_{I,k} \equiv \mathbf{0}$. □

The proof of part a) shows not only that $\mathbf{E}_{I,k}$ intersects with the present graph of the system, but also with the failure report. Therefore, the reported failure no longer exists in the current abstraction after it is updated in step 6 of the CLE algorithm.

Proposition 3.2 shows that the criterion $\mathbf{L}_{I,k} \equiv \mathbf{0}$ is correct in the sense that $\mathcal{S}(\mathbf{E}_{I,k})$ never contains edges that are in the exact graph of the system. The following proposition shows that it is also complete, in the sense that if no I and k satisfy $\mathbf{L}_{I,k} \equiv \mathbf{0}$, then the intersection of the languages of active automata is indeed not empty.

Proposition 3.3 *If* $\mathbf{L}_{Act,k} \not\equiv \mathbf{0}$ *for all* $k \geq 0$, *then* $\mathcal{L}(\bigotimes_{i \in Act} A_i) \neq \emptyset$.

Proof. From $\mathbf{L}_{Act,k} \not\equiv \mathbf{0}$ it follows that for every $k \geq 0$ there exists $\sigma_k \in \Sigma$ such that $\mathbf{T}(r_{i,k}, \sigma_k, r_{i,k+1}) = 1$ for every $i \in Act$. Thus, r_i is a run of $\sigma = \sigma_0\sigma_1 \ldots \in \Sigma^\omega$ in A_i, and r is an accepting run of σ in $\bigotimes_{i \in Act} A_i$. □

The second refinement task occurs when we find a subset New of ignored components that eliminates the reported failure. In that case, we include New in the set of active components, and update the current abstraction of the system, as indicated in steps 8–10. A closer look at step 9 reveals that components in New are included in the current abstraction, but *communication is ignored* between them and other active components. This way, we only need to update the graph of the system from the previous iteration. Thus, we never compute the full transition relation of the system. On the other hand, the reported failure will not be eliminated in the updated system. This is important only in terms of efficiency, since the failure will eventually be eliminated in subsequent iterations, but their number may be large. To define a refinement step which retains some of the mentioned advantages, but also eliminates the reported failure is another interesting open problem.

3.2.5 Correctness

In showing the correctness of the CLE algorithm we assume that the **verify** function called in steps 3 and 7 is correct.

Proposition 3.4 *The CLE algorithm is correct.*

Proof. To show that the algorithm can not terminate with a false success we need to show that at every point in the algorithm the description of the system is an abstraction of the exact system. We show this by induction on the number of iteration:

BASE CASE: By Lemma 3.1 initial B computed in step 2 is an abstraction of the exact system.

INDUCTIVE STEP: As an inductive assumption, we postulate that before step 6 is executed, the graph of the current abstraction contains the exact graph of the system, and that before step 9 is executed, B is an abstraction of $\bigotimes_{i \in Act} A_i$.

Now, by Proposition 3.2b and the inductive assumption the updated graph in step 6 contains the exact graph of the system, hence by Proposition 3.1 it is an abstraction. Also, by Lemma 3.2, Proposition 3.1 and the inductive assumption the updated B in step 9 is an abstraction of $\bigotimes_{i \in Act \cup New} A_i$, hence by Lemma 3.1 it is also an an abstraction of the exact system.

To show that the algorithm can not terminate with a false failure we need to show that a failure is reported only if a sequence is found that is in the language of all the components $A_0, \ldots, A_n$. Indeed, a failure is reported only if $\mathcal{L}_r \cap \mathcal{L}(\bigotimes_{i \notin Act} A_i) \neq \emptyset$. Since by definition $\mathcal{L}_r \subseteq \mathcal{L}(\bigotimes_{i \in Act} A_i)$, the claim follows. □

Proposition 3.5 *The CLE algorithm terminates.*

Proof. At each iteration we either execute step 6 or steps 8-10 or terminate. Steps 8–10 can not be executed more than n times, because every time it is executed the set *Act* grows by at least one new element. By Proposition 3.2a, at least one edge is eliminated from the graph of the current abstraction in step 6. Therefore, step 6 can also be executed only finitely many times. □

The proof of the Proposition 3.5 has an unfortunate consequence that the number of steps in the worst case is proportional to the possible number of edges in the exact graph, which is exponential in the number of components.

3.3 Experimental results

We have tested our algorithm on two different properties of the well known Dining Philosopher's problem. We have used the solution with an encyclopedia [KM89] to insure that the system is deadlock and starvation free. All experiments were performed on a 400Mb DEC MIPS 5000 workstation.

Table 3.1: Results for the mutual exclusion property

philosophers	2,000	4,000	6,000	8,000	10,000	12,000	14,000
reachable states	10^{953}	10^{1908}	10^{2862}	10^{3816}	10^{4771}	10^{5618}	10^{6595}
CPU time [sec]	42.8	160.4	359.1	627.8	981.2	1346.6	1854.4

The first property we have verified is that two neighboring philosophers will never eat at the same time (mutual exclusion). More precisely we have verified this property for the first two philosophers. The results are summarized in Table 1. We could not verify any larger examples due to the memory limit. No comparison is given to the direct approach since it can verify systems with at most several hundred philosophers. This property is local, i.e. it depends only on the behavior of the first two philosophers. Thus, for any number of philosophers the algorithm has verified the property in one iteration.

The other property we have verified is that the first philosopher will not be hungry forever (starvation). In this case results were not nearly as good as for the mutual exclusion method. In fact, it performed worse than the direct method with the number of iterations reaching hundreds for less than ten philosophers. Although this property is expressed in terms of outputs of only one philosopher, it is not a local property. In fact, some aspects of the behavior of all philosophers must be included to verify this property. Thus, a complete exact graph was eventually computed and the effort of only the last iteration was the same as that of the direct approach. All the other iterations were basically an (expensive) overhead. This indicates that this approach can be robust only if more sophisticated choices are made on which communication to ignore initially and how to eliminate a failure trace.

3.4 Related work

Several approaches to (more or less) automatic abstraction of communicating finite-state systems have been proposed. Many of these approaches can be seen as a two phase process: first an equivalence relation is defined over states of the system, and then a quotient with respect to that equivalence is computed.

In some cases the equivalence is given by a user [CGL92, GL93], and in others it is computed such that it preserves certain classes of properties (typically fragments of CTL). In particular, approaches are developed for preserving all CTL formulas [CLM89,

BFH90], all ACTL formulas [DGG93], a single CTL formula [SCSVB92, ASSSV94], and a single ACTL formula [DGG93]. Computing the quotient of a given system in the simplest form requires traversal of the state space, and thus is as hard as verifying the system. Therefore, in all of the approaches mentioned above procedures are defined that construct a quotient without ever constructing the full system. In [BFH90] and [DGG93] the problem is addressed by interleaving of the quotient and equivalence computation. Initially, a quotient is constructed with respect to a very coarse equivalence (that does not necessarily have any preservation properties). The equivalence and the quotient are then iteratively refined until the preservation is achieved. In [CLM89, CGL92, SCSVB92, ASSSV94] the quotient is built hierarchically: components are first abstracted, then composed, and then abstracted again to simplify building of even larger blocks. Finally, in [GL93] an approximation of the quotient is computed directly from the program text.

None of the approaches mentioned above can simplify any fairness constraints. Aziz et al. [ASB+94] have defined an equivalence that includes fairness constraints, but they did not provide an efficient algorithm for computing. Another problem is that approaches [CLM89, BFH90, DGG93] compute actually reductions with respect to large sets of properties, so they are not likely to yield significant simplification. For language containment, reduction only needs to preserve the language (which is weaker than preserving even only ACTL formulas), but unfortunately there are no efficient algorithms for language preserving minimization, and in many cases even the optimally reduced system with the same language is still quite large. Better simplifications are to be expected from property-specific approaches like ours and [SCSVB92, ASSSV94, DGG93].

A quite different approach was taken by Halbwachs [Hal93]. Rather than simplifying a system beforehand, the simplifications are made while computing reachable states. Sets of states are computed symbolically, and instead of the exact set of reachable states, a superset with smaller symbolic representation is computed. Since verifying safety properties reduces to deciding reachability of a set of states (see chapter 4), the method is conservative for such properties. In our knowledge, this and ours are the only approaches that demonstrate that state minimization is not the only possible simplification objective.

Chapter 4

Real-Time Systems

Classical formal models abstract quantitative time and retain only the ordering of events (e.g. [CES86, Kur90]). However, for many systems it is essential to verify their real-time behavior. It is not enough that an alarm sent by a smoke detector gets to a fire station, it must get there in time.

Most of the recent developments in formal verification of real-time systems stem from the work of Alur and Dill [Dil89, AD90]. They define *timed automata*, a model where a finite state system is augmented with real-valued time measuring devices called *timers*, and then show that the verification of such systems can be reduced to the verification of "ordinary" (untimed) finite-state systems.

In this chapter we propose an iterative algorithm for verification of timed automata. The algorithm can be used in a fully automatic mode, or in a guided mode where user suggests an initial abstraction. We also introduce *timed automata with decrements*, an extension of timed automata that allows timers to be decremented. This extension has some practical applications as demonstrated by a model of a real-time operating system presented in chapter 5. However, this option must be used cautiously, because it makes the verification problem undecidable in general. To verify systems in the extended model, we propose a semi-decision procedure in chapter 6.

4.1 Timed automata with decrement

Real-time behavior is modeled by adding time-measuring devices called *timers* to the description of a finite-state systems. Timers can then be used to bound the elapsed time

between transitions. Before we formally introduce the model, we introduce some notation that enables us to treat uniformly both strict and non-strict inequalities (e.g.: $\mathbf{x} < 3, \mathbf{y} \leq 5$).

4.1.1 Bounds, timing inequalities and timer valuations

The *domain of bounds* B is the union of the set of *lower bounds* LB and the set of *upper bounds* UB, where:

$$\begin{aligned} UB &= Z \cup \{n^- | n \in Z\}, \\ LB &= Z \cup \{n^+ | n \in Z\}, \end{aligned}$$

and Z is the set of integers. Expressions n^- (respectively n^+) can be thought of as real number infinitesimally smaller (larger) than n. The integer addition is naturally extended to the bounds of the same type, by: $n+m^- = n^-+m = n^-+m^- = (n+m)^-$. Similarly, the integer negation is naturally extended to bounds by: $-(n^-) = (-n)^+$, and $-(n^+) = (-n)^-$. Also, the integer ordering is extended to B by: $n^- < n < n^+ < (n+1)^-$. Finally, we compare real numbers to bounds and say that $x \leq n^-$ if $x < n$ and $x > n^-$ is $x \geq n$.

Let $V = \{\mathbf{x}_1, \ldots, \mathbf{x}_{|V|}\}$ denote some set of *timer variables*. The set of *timing inequalities* Ψ is the set of formulas of the form $\mathbf{x} \leq a$, $\mathbf{x} \geq b$ or $\mathbf{x} - \mathbf{y} \leq c$, where $\mathbf{x}, \mathbf{y} \in V$, $a, c \in UB$, and $b \in LB$. Let 2^{Ψ} denote the set of all *finite* subsets of Ψ.

A *timer valuation* $\tau : V \to R$ assigns a real value to every timer variable. We say that a timer valuation τ *satisfies* a timing inequality $\mathbf{x} \leq a$, if $\tau(\mathbf{x}) \leq a$. Similarly, we say that τ satisfies $\mathbf{x} \geq a$, if $\tau(\mathbf{x}) \geq a$. Finally, τ satisfies $\mathbf{x} - \mathbf{y} \leq a$ if $\tau(\mathbf{x}) - \tau(\mathbf{y}) \leq a$.

In our model, timers can be modified in two ways: they can either be decremented by some positive integer amount, or reset to zero. To represent these modifications, we define the set of *timer decrements* (denoted by $\mathcal{M}$) to be the set of non-negative integers augmented with a special symbol $\perp$ that indicates that a timer is to be reset.

4.1.2 Syntax

A *timed automaton with decrement* (TAD) is a 7-tuple $(S, \mathbf{I}, \mathbf{T}, \mathbf{F}, V, \mathbf{C}, \mathbf{M})$ where $S, \mathbf{I}, \mathbf{T}, \mathbf{F}$ are as in untimed case, and:

- $V = \{x_1, \ldots, x_{|V|}\}$ is a set of timer variables,
- $\mathbf{C} : S \times S \to 2^{\Psi}$ is a *timing constraint* function, and

- $\mathbf{M} : S \times S \times V \to \mathcal{M}$ is a *timer modifier* function.

If $\mathbf{M}(s, q, \mathbf{x}) \in \{\perp, 0\}$ for all $s, q \in S$, then we say that a TAD is a *timed automaton.*

Intuitively, a transition $s \to q$ is enabled only at times where all inequalities in $\mathbf{C}(s, q)$ are satisfied. If $\mathbf{M}(s, q, \mathbf{x}) = \perp$, then the timer $\mathbf{x}$ is reset on $s \to q$. Otherwise, it is decremented by $\mathbf{M}(s, q, x)$. When we draw TAD's we label an edge $s \to q$ with $\mathbf{x} := 0$, if $\mathbf{M}(s, q, \mathbf{x}) = \perp$. If $\mathbf{M}(s, q, \mathbf{x}) = n > 0$, we label $s \to q$ with $\mathbf{x} := \mathbf{x} - n$. If $\mathbf{M}(s, q, \mathbf{x}) = 0$ we omit the label.

4.1.3 Semantics

We provide semantics of timed automata in terms of *timed sequences* of I/O values $(\sigma_1, \delta_1), (\sigma_2, \delta_2), \ldots, (\sigma_n, \delta_n)$ where $\sigma_i \in \Sigma$ is an I/O value, and δ_i is a positive real number. We think of δ_i as an elapsed time between two transitions, which in turn uniquely determine timer valuations at every transition. Formally, given a TAD $(S, \mathbf{I}, \mathbf{T}, \mathbf{F}, V, \mathbf{C}, \mathbf{M})$, with every sequence of states $s_0, s_1, \ldots, s_n$ and every sequence of positive real numbers $\delta_1, \ldots, \delta_n$ we associate a sequence of timer valuations $\tau_1, \tau_2, \ldots, \tau_n$ defined as follows:

$$\begin{aligned} \tau_1(\mathbf{x}) &= \delta_1 \text{ for all } \mathbf{x} \in V \ , \\ \tau_{i+1}(\mathbf{x}) &= \begin{cases} \delta_{i+1} & \text{if } \mathbf{M}(s_{i-1}, s_i, \mathbf{x}) = \perp \ , \\ \tau_i(\mathbf{x}) - \mathbf{M}(s_{i-1}, s_i, \mathbf{x}) + \delta_{i+1} & \text{otherwise,} \end{cases} \end{aligned}$$

for all $i = 1, \ldots, n-1$ and all $\mathbf{x} \in V$.

In case of the timed automata the second condition simplifies to:

$$\tau_{i+1}(\mathbf{x}) = \begin{cases} \delta_{i+1} & \text{if } \mathbf{M}(s_{i-1}, s_i, \mathbf{x}) = \perp \ , \\ \tau_i(\mathbf{x}) + \delta_{i+1} & \text{otherwise.} \end{cases}$$

We say that a sequence of states $s_0, s_1, \ldots, s_n$ is an *accepting run* of a timed sequence $(\sigma_1, \delta_1), (\sigma_2, \delta_2), \ldots, (\sigma_n, \delta_n)$ in a TAD $(S, \mathbf{I}, \mathbf{T}, \mathbf{F}, V, \mathbf{C}, \mathbf{M})$ if:

1. $s_0, s_1, \ldots, s_n$ is an accepting run of $\sigma_1 \sigma_2 \ldots \sigma_n$ in an (untimed) automaton $(S, \mathbf{I}, \mathbf{T}, \mathbf{F})$, and

2. τ_i satisfies *all* timing constraints in $\mathbf{C}(s_{i-1}, s_i)$ for all $i = 1, \ldots, n$, where τ_i is the i-th valuation in the sequence $\tau_1, \tau_2 \ldots \tau_n$ associated with $s_0, s_1, \ldots, s_n$ and $\delta_1, \ldots, \delta_n$.

The *language* of a TAD A (denoted by $\mathcal{L}(A)$) is the set of all timed sequences that have an accepting run in A. The *untimed language* of A is a projection of $\mathcal{L}(A)$ to Σ, i.e.:

$$Untimed(\mathcal{L}(A)) = \{\sigma_1 \ldots \sigma_n | \text{ there exists } (\sigma_1, \delta_1), \ldots, (\sigma_1, \delta_1) \in \mathcal{L}(A)\} \ .$$

4.1.4 Issues in model selection

Notice that in the definition of an accepting run condition 1 does not depend on timing, and condition 2 does not depend on I/O values $\sigma_1\sigma_2 \ldots \sigma_n$, but only on times. If times $\delta_1, \ldots, \delta_n$ satisfy condition 2, we say that they are a consistent timing of $s_0, s_1, \ldots, s_n$. We use this decoupling in our language emptiness algorithm, where we first relax all the timing constraints, and then check whether a run reported by a verification tool admits a consistent timing. To preserve this decoupling, we do not allow timing constraints **C** and modifiers **M** to be functions of the I/O variable as well. This extension would allow for different transitions between two states to be subject to different constraints.

Another extension we have considered (and decided against) is to allow timing constraints to be arbitrary Boolean formulas over timing inequalities. This extension would again violate the property that for every transition there exists a unique set of timing inequalities which must all be satisfied. This property significantly simplifies a check whether a run admits a consistent timing.

Our choices do not sacrifice the expressiveness of the model. It is always possible to transform an automaton in the extended model (i.e. with I/O dependent timing constraints which are arbitrary Boolean formulas), to an equivalent one in our model, at the expense of some states being multiplied.

Another issue is finite versus infinite sequences. We develop our formalism for automata on finite sequences, mostly for simplicity, but also to stress the fact (argued also in [Pne92]) that infinite sequences are more important for untimed than for real-time systems. The argument is that most liveness properties (e.g. "*the gate will not stay down forever*") are really just an abstraction of timed safety properties (e.g. "*the gate will not stay down for more than seven time units*") and thus can be disproved by finite timed sequences. Therefore, in a framework that allows precise timing information safety properties suffices in most cases. Still, we discuss extensions of our algorithms to infinite sequences which are necessary to verify (quite rare) timed liveness properties (e.g. "*the gate will eventually stay down for at least three time units*).

4.1.5 Composition of TAD's

TAD's, like any other model, are not of much use unless a composition is defined over its instances. Similar to the untimed case, the composition is well defined only if two automata have the same I/O variable. Given two TAD's

$$\begin{aligned} A &= (S_A, \mathbf{I}_A, \mathbf{T}_A, \mathbf{F}_A, V_A, \mathbf{C}_A, \mathbf{M}_A) \ , \\ B &= (S_B, \mathbf{I}_B, \mathbf{T}_B, \mathbf{F}_B, V_B, \mathbf{C}_B, \mathbf{M}_B) \ , \end{aligned}$$

satisfying $V_A \cap V_B = \emptyset$, the *composition* of A and B is the TAD:

$$A \otimes B = (S_A \times S_B, \mathbf{I}_A * \mathbf{I}_B, \mathbf{T}_A * \mathbf{T}_B, \mathbf{F}_A * \mathbf{F}_B, V_A \cup V_B, \mathbf{C}_{A\otimes B}, \mathbf{M}_{A\otimes B}) \ ,$$

where for all $q_A, s_A \in S_A$, $q_B, s_B \in S_B$, and all $\mathbf{x} \in V_A \cup V_B$:

$$\begin{aligned} \mathbf{C}_{A\otimes B}((q_A, q_B), (s_A, s_B)) &= \mathbf{C}_A(q_A, s_A) \cup \mathbf{C}_B(q_B, s_B) \ , \\ \mathbf{M}_{A\otimes B}((q_A, q_B), (s_A, s_B), \mathbf{x}) &= \begin{cases} \mathbf{M}_A(q_A, s_A, \mathbf{x}) & \text{if } \mathbf{x} \in V_A \ , \\ \mathbf{M}_B(q_B, s_B, \mathbf{x}) & \text{if } \mathbf{x} \in V_B \ . \end{cases} \end{aligned}$$

It is not hard to show that a run is accepting in $A \otimes B$ if it consists of an accepting run in A and an accepting run in B. Therefore, the composition of TAD's has the language intersection property:

$$\mathcal{L}(A \otimes B) = \mathcal{L}(A) \cap \mathcal{L}(B) \ .$$

The verification paradigm of TAD's is similar to the untimed case. We think of the language of a TAD as the set of all possible behaviors, and to prove (or disprove) that they are all acceptable, we prove (or disprove) the language emptiness of the composition of the system with a TAD whose language represents all unacceptable behaviors.

4.1.6 Example

In the railroad crossing example [ACD+92] shown in Figure 4.1, the system has three components: the train, the gate, and the controller. The train approaches from outside of the crossing. After at least two time units of approaching, the train will enter the crossing, and then exit at most five time units from the beginning of the cycle.

Exactly one time unit after the train approaches the controller commands the gate to lower, and at most one time unit later the gate will close. Similarly, at most one time unit after the train exits the crossing the controller commands the gate to raise, and at

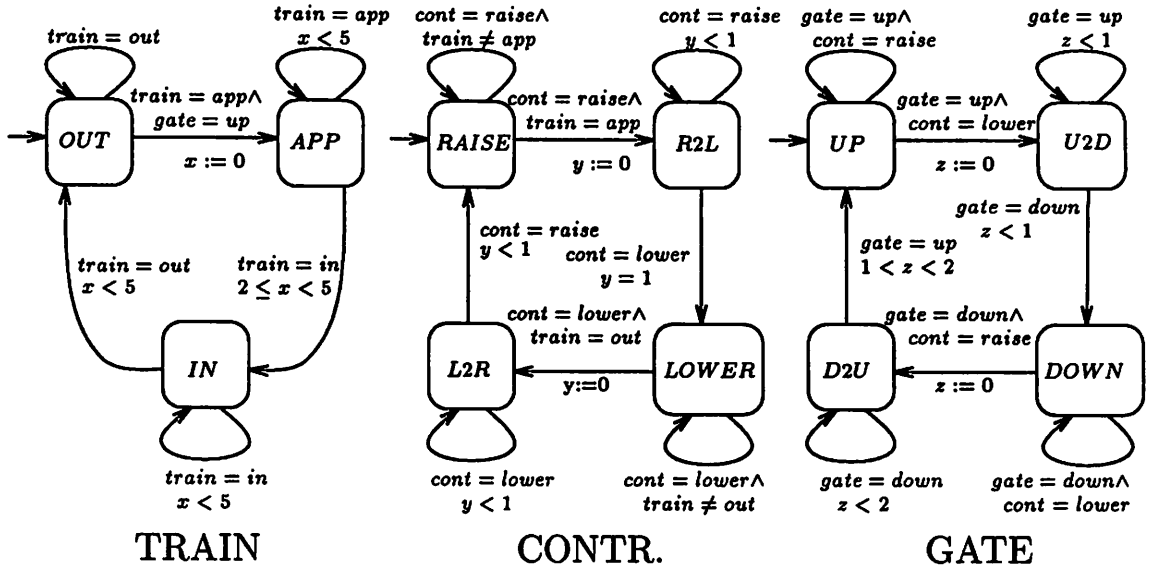


Figure 4.1: Railroad crossing example.

least one and at most two time units after that the gate will open. For simplicity, we only consider the case when there is ample time between trains. Therefore, we require the train to approach only if the gate is *up*.

Formally, common I/O variable of automata in Figure 4.1 is a vector with components *train*, *cont* and *gate* ranging over {*app*, *in*, *out*}, {*lower*, *raise*} and {*up*, *down*} respectively.

Two properties to be verified are: [1]

safety: the gate is *down* whenever the train is *in*, and

liveness: the gate is never down for more than seven time units.

Figure 4.2 shows automata which accept all timed sequences of I/O values that *violate* these properties. A property is satisfied if the composition of the corresponding automaton with the automata in Figure 4.1 has the empty language.

[1]The names of these properties are not to be confused by general classifications of properties to safety and liveness.

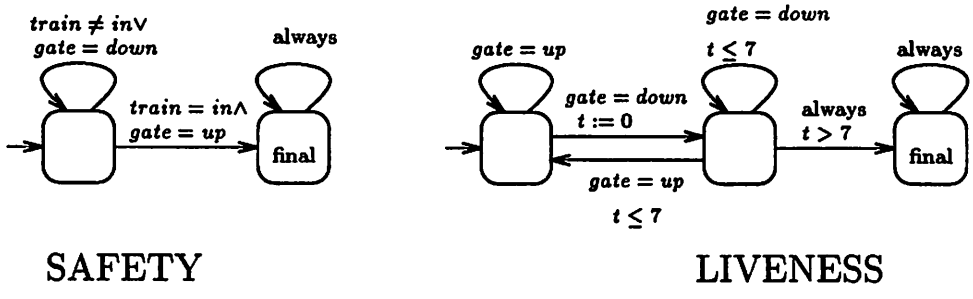


Figure 4.2: Properties for the railroad crossing example.

4.2 Equivalent untimed automaton

In this section we show that the language emptiness problem is decidable for timed automata. This result of Alur and Dill [AD90] enables automatic verification of real-time systems. We present it in two steps. First, we provide an alternative semantics of TAD's, by associating with each TAD A an untimed, infinite-state automaton A^∞, such that $\mathcal{L}(A)$ is equal to $\mathcal{L}(A^\infty)$. Then we show that in the case of timed automata we can define a finite equivalence relation on states of A^∞ such that the quotient of A^∞ with respect to that relation has the same language as A^∞. Since that quotient is finite-state, we can check for language emptiness.

4.2.1 Alternative semantics

The basic idea in constructing A^∞ is to expand explicitly the state of the system with timer values. Also, we expand the set of I/O values, to make time between transitions an explicit part of the language.

Formally, given a TAD $A = (S, \mathbf{I}, \mathbf{T}, \mathbf{F}, V, \mathbf{C}, \mathbf{M})$ over the alphabet Σ, the *companion automaton* A^∞ is defined as follows:

- the state space of A^∞ is $S \times R^{|V|}$, where R is the set of real numbers,
- a state $(s, x_1, \ldots, x_{|V|})$ is initial if $\mathbf{I}(s) = 1$, and $x_1 = \ldots = x_{|V|} = 0$,
- a state $(s, x_1, \ldots, x_{|V|})$ is final if $\mathbf{F}(s) = 1$,
- the alphabet of A^∞ is $\Sigma \times R^+$, where R^+ is the set of positive reals,

- a transition $(s, x_1, \ldots, x_{|V|}) \xrightarrow{\sigma,\delta} (q, y_1, \ldots, y_{|V|})$ is in the transition relation of A^∞ if all of the following holds:

  1. $\mathbf{T}(s, \sigma, q) = 1$,
  2. for all $i = 1, \ldots, |V|$:

$$y_i = \begin{cases} 0 & \text{if } \mathbf{M}(s, q, \mathbf{x}_i) = \perp \ , \\ x_i + \delta - \mathbf{M}(s, q, \mathbf{x}_i) & \text{otherwise,} \end{cases}$$

  3. $x_i + \delta \leq c$ for all i, c such that $(\mathbf{x}_i \leq c) \in \mathbf{C}(s, q)$,
  4. $x_i + \delta \geq c$ for all i, c such that $(\mathbf{x}_i \geq c) \in \mathbf{C}(s, q)$,
  5. $x_i - x_j \leq c$ for all i, j, c such that $(\mathbf{x}_i - x_j \leq c) \in \mathbf{C}(s, q)$.

It is straightforward to show that the language of a TAD as defined in section 4.1.3 is the same as the language of the companion automata. It follows then easily, that the language of the automaton $A^\infty|_\Sigma$ is exactly the untimed language of A.

4.2.2 Equivalence of timer values

The states of the companion automaton A^∞ of some TAD $A = (S, \mathbf{I}, \mathbf{T}, \mathbf{F}, V, \mathbf{C}, \mathbf{M})$ include real values of all the timers in the system. Alur and Dill [AD90] have observed that many of these states are equivalent. We describe this equivalence as the intersection of many coarser equivalence relations. More precisely, for any two states:

$$(s, a_1, \ldots, a_{|V|}), (q, b_1, \ldots, b_{|V|}) \in S \times \mathbf{R}^{|V|}$$

of A^∞ we define:

$(s, a_1, \ldots, a_{|V|}) \sim_S (q, b_1, \ldots, b_{|V|})$ if an only if $s = q$,

$(s, a_1, \ldots, a_{|V|}) \sim_{\mathbf{x}_i \leq c} (q, b_1, \ldots, b_{|V|})$ if an only if either both $a_i \leq c$ and $b_i \leq c$ hold, or neither holds,

$(s, a_1, \ldots, a_{|V|}) \sim_{\mathbf{x}_i - \mathbf{x}_j \leq c} (q, b_1, \ldots, b_{|V|})$ if an only if either both $a_i - a_j \leq c$ and $b_i - b_j \leq c$ hold, or neither holds,

$(s, a_1, \ldots, a_{|V|}) \sim (q, b_1, \ldots, b_{|V|})$ if an only if:

  1. $(s, a_1, \ldots, a_{|V|}) \sim_S (q, b_1, \ldots, b_{|V|})$, and

2. $(s, a_1, \ldots, a_{|V|}) \sim_{x_i \leq c} (q, b_1, \ldots, b_{|V|})$ for every $i = 1, \ldots, |V|$, and every bound $c \in B$ satisfying $-c_{max} \leq c \leq c_{max}$, where c_{max} is the largest (by absolute value) bound appearing in the system, and
3. $(s, a_1, \ldots, a_{|V|}) \sim_{x_i - x_j \leq c} (q, b_1, \ldots, b_{|V|})$ for every $i, j = 1, \ldots, |V|$, $i \neq j$ and every bound $c \in B$ satisfying $-c_{max} \leq c \leq c_{max}$.

Intuitively, equivalence classes of $\sim$ preserve the untimed portion of the state exactly, preserve the integer part of the value of timers (at least up to c_{max}), and also preserve the differences between the timer values. This information allows us to infer the ordering of fractional values of all timer values. This ordering is important because it determines the order in which timer values cross the integer boundaries.

Let us restrict our attention to the case when A is a timed automaton. In this case, values of timers are never negative, so we can restrict the state space of A^∞ accordingly. Therefore, the partition induced by $\sim$ is finite. Also, since $\sim_S$ distinguishes every state in S, the partition induced by $\sim$ respects the final states of A^∞. It is somewhat harder to see that the partition induced by $\sim$ is also stable with respect the transition relation of $A^\infty|_\Sigma$. Recall that the transition relation of A^∞ requires that the present state value of timers increased by δ satisfy all timing constraints and that the next value of a timer is either 0 or the present state value increased by δ, depending on the value of the timer modifier. So, we can break the transition into the following steps:

1. all timer values are increased by the same amount $\delta > 0$,
2. the untimed part makes a transition (say $s \rightarrow q$),
3. new values of timers must satisfy all timing constraint in $\mathbf{C}(s, q)$
4. some timers are reset, as warranted by $\mathbf{M}$.

It is easy to see that if two are states equivalent, steps 2, 3, and 4 will transform them to new states that are also equivalent. But, it is also true that for any pair of equivalent states:

$$(s, a_1, \ldots, a_{|V|}) \sim (s, b_1, \ldots, b_{|V|}) \ ,$$

and any $\delta > 0$, there exists $\epsilon > 0$ such that:

$$(s, a_1 + \delta, \ldots, a_{|V|} + \delta) \sim (s, b_1 + \epsilon, \ldots, b_{|V|} + \epsilon) \ . \tag{4.1}$$

If we start from $(a_1, \ldots, a_{|V|})$ and grow δ from 0 to its final value, we will cross boundaries of equivalence classes of some relation $\sim_{x_i \leq c}$. That growing ϵ from $(b_1, \ldots, b_{|V|})$ will hit the same classes in the same order follows from the fact that the partition induced by $\sim$ refines the partitions induced by $\sim_{x_i - x_j \leq c}$ for all interesting values of c. Therefore to get (4.1) it is enough to choose ϵ that crosses the same number of boundaries as δ.

Let $T^{\infty}|_{\Sigma}$ denote the transition relation of $A^{\infty}|_{\Sigma}$. Using (4.1), for any transition:

$$\Big((s, a_1, \ldots, a_{|V|}), \sigma, (q, c_1, \ldots, c_{|V|})\Big) \in T^{\infty}|_{\Sigma} \ ,$$

and any state:

$$(s, b_1, \ldots, b_{|V|}) \sim (s, a_1, \ldots, a_{|V|}) \ ,$$

we can construct another transition:

$$\Big((s, b_1, \ldots, b_{|V|}), \sigma, (q, d_1, \ldots, d_{|V|})\Big) \in T^{\infty}|_{\Sigma} \ ,$$

such that

$$(q, c_1, \ldots, c_{|V|}) \sim (q, d_1, \ldots, d_{|V|}) \ .$$

In other words, $\sim$ is stable with respect to $T^{\infty}|_{\Sigma}$, and the following holds:

Theorem 4.1 (Alur-Dill) *Let A be a timed automaton and let $A^{\infty}|_{\Sigma}$ and $\sim$ be defined as above. Then:*

$$\mathcal{L}((A^{\infty}|_{\Sigma})_{/\sim}) = Untimed(\mathcal{L}(A)) \ .$$

The states of $(A^{\infty}|_{\Sigma})_{/\sim}$ are often called *regions*, and $(A^{\infty}|_{\Sigma})_{/\sim}$ itself is called a *region automaton*. Theoretically, theorem 4.1 is important because it allows checking language emptiness of timed automata to be done by checking language emptiness of finite-state automata. Unfortunately, the theorem is not directly applicable in practice because the number of regions is exponential not only in the number of timers, but also in size of the constant c_{max} as well.

4.2.3 Extensions to TAD's

For TAD's in general the construction above breaks for two reason:

1. timers can have negative values, so $\sim$ is not finite,

2. $\sim$ is not stable with respect to $\mathbf{T}^{\infty}|_{\Sigma}$, because different values of timers larger than c_{max} can be distinguished after decrementing.

However, we can modify the definition of $\sim$ to be the intersection of $\sim_S$, with equivalence relations $\sim_{\mathbf{x}_i \leq c}$ and $\sim_{\mathbf{x}_i - \mathbf{x}_j \leq c}$ for all $i, j = 1, \ldots, |V|$ and *all* bounds $c \in B$. In this case two states are equivalent if:

1. they agree on untimed state components,

2. they agree on the integer part of the value of every timer, and for every timer the fractional parts in both states is either equal or different from zero,

3. they agree on the integer part of the difference of values of any two timers (and again the fractional parts of the difference in both states is either equal or different from zero).

The decomposition of every transition into four steps is still valid except that in step 4 the timers that are not reset could be decremented by some integer amount (depending on the timer modifier). It can be shown, by the similar reasoning as before, that this modified $\sim$ is stable with respect to $\mathbf{T}^{\infty}|_{\Sigma}$.

Unfortunately, the modified $\sim$ is not finite. For any given problem there might exist some lower and upper bounds, outside of which all timer values are equivalent, but such constants do not always exist, as implied by the following result:

Theorem 4.2 *The language emptiness problem for TAD's is undecidable.*

Proof. By reduction of the halting problem for deterministic two-counter machines.[2] Given a two-counter machine M with counters C_1 and C_2, we construct a TAD A_M with a single I/O value and three timers: x, y and z. The state space and initial states of A_M are those of M, and the unique final state of A_M is the halt state of M. The transition relation of A_M is that of M, except for the following:

[2] A deterministic two-counter machine is basically a finite-state automaton with a unique initial and final state and augmented with two integer-valued counters. The form of the transition relation is restricted such that from every state there are exactly four transitions with mutually disjoint enabling conditions, corresponding to possible combinations of each counter being (or not) equal to zero. Also, every transition specifies for each counter whether it is to be incremented or decremented by 1. Two-counter machines can encode Turing machines and therefore their halting problem is undecidable. For more details, see [HU79, chapters 7, 8].

- timer $\mathbf{x}$ is reset on every transition, and a timing constraint $\mathbf{x} = 1$ is placed on every transition; this ensures that transitions will happen on integer times, starting at time 1,
- on transitions where C_1 is decremented, we decrement $\mathbf{y}$ by 2, and on transitions where C_2 is decremented, we decrement $\mathbf{z}$ by 2,
- on transitions enabled by $C_1 = 0$ ($C_1 \neq 0$), we place a timing constraint $\mathbf{y} = 1$ (respectively $\mathbf{y} \neq 1$) and similarly, on transitions enabled by $C_2 = 0$ ($C_2 \neq 0$), we place a timing constraint $\mathbf{z} = 1$ (respectively $\mathbf{z} \neq 1$).

It is easy to see that on every transitions the values of $\mathbf{y}$ and $\mathbf{z}$ are equal to the values of C_1 and C_2 incremented by 1. Therefore, A_M simulates M and its language is not empty if and only if M halts. □

4.3 Related work

For many systems, the region automaton is far too large to be traversed explicitly. In section 4.4 we will propose an approach that can hopefully avoid building the full region automaton for most of the systems. But first we review some other approaches in this direction.

4.3.1 Successive approximation

Alur et al. [AIKY93] have suggested an approach which can be seen as an application of the generic iterative algorithm of section 1.3 to real-time systems. Initially, all timing constraints are relaxed. Then, the verification is attempted. If successful, then the system is verified. Otherwise, Alur et al. check whether the failure report admits a consistent timing. If it does, then it is a valid proof that the system does not satisfy the property at hand. Otherwise, the following steps are performed to eliminate it:

1. **Minimizing the number of timing constraints:** In this step an exhaustive search is performed to check whether there exists a subset of timing constraints that makes the failure report timing inconsistent. The search is done in increasing order of the size of the subset. The hope is that a small subset that suffices can be found. The argument is made that if a small subset cannot be found, then the region automaton

is far too large to implement, and thus the exhaustive search is not a bottleneck of the algorithm.

2. **Optimizing timing constants:** In this step an attempt is made to relax the timing constraints selected in the previous step (by lowering the lower bounds, and increasing the upper bound) in a way that the failure report is still timing inconsistent, and all the constants have a common factor (as large as possible). Dividing all the constants with their greatest common divider reduces the size of the region automaton. Generally, this is a hard optimization problem, but by similar reasoning as above, it is not a bottleneck of the algorithm.

3. **Building the region automaton:** In this step a full region automaton is built for a subset of timers selected in step 1 with the optimized bounds from step 2.

This approach has been implemented as an extension of the verification tool COSPAN [HK88].

4.3.2 Minimization

We have shown that the semantics of a timed automaton can be defined by an infinite state companion automaton. Alur and Dill construct the region automaton, a finite-state equivalent to the companion automaton, but that is not necessarily the smallest automaton equivalent to the companion automaton. In this section we survey several algorithms that compute an (hopefully small) equivalent to the companion automaton.

Coarsest stable partition

In chapter 2 we showed that if some partition ρ of the state space of an automaton A is stable with respect to the transition relation of A and respects final states of A, then the languages of A and $A_{/\rho}$ are the same.

In general, given an automaton A and some initial partition ρ, there exists a unique coarsest refinement ρ' of ρ which is stable with respect to the transition relation of A. Two algorithms have been proposed by Boujjani, Fernandez and Halbwachs [BFH90], and Lee and Yannakakis [LY92] to compute $A_{\rho'}$ given A and ρ. Both algorithms start with $A_{/\rho}$ and then split some of its reachable, but unstable, states into more stable states. This operation is repeated until all reachable states are stabilized. The algorithms differ in the order of splitting and computing reachability information. The algorithms are given in

terms of abstract operations *post* (for reachability), *pre* (for checking stability) and *split* (for making classes more stable).

Alur et al. [ACD+92, ACH+92] and Yannakakis and Lee [YL93] have specialized these algorithms to timed automata. Given a timed automaton $(S, \mathbf{I}, \mathbf{T}, \mathbf{F}, V, \mathbf{C}, \mathbf{M})$, the initial partition is chosen to be $\rho = \{\{s\} \times \mathbf{R}^{|V|} \mid s \in S\}$, i.e. all untimed components of the states are distinguished and all timer values are grouped in the same equivalence class. As algorithm progresses, ρ is refined, and at all times the classes of the current partition are represented by a single untimed state component and a convex set of timer values bounded by hyperplanes of the form $\mathbf{x} = c$ or $\mathbf{x} - \mathbf{y} = c$ ($\mathbf{x}$ and $\mathbf{y}$ are timers, c is an integer). To represent such convex sets Dill [Dil89] suggested *difference bound matrices*, square matrices with $|V|+1$ rows and columns and entries which are upper bounds. A vector of timer values $(x_1, \ldots, x_{|V|})$ belongs to the set of values represented by a matrix C if:

1. for all $i, j = 1, \ldots |V|$: $x_i - x_j \leq c_{i,j}$, where $c_{i,j}$ is the entry in the i-th row and j-th column of C,
2. for all $i = 1, \ldots |V|$: $x_i \leq c_{i,|V|+1}$,
3. for all $j = 1, \ldots |V|$: $-x_j \leq c_{|V|+1,j}$.

The *pre*, *post*, and *split* operations can be defined on difference bound matrices with $O(|V|^2)$ time complexity. Even though, many matrices represent the same set of timer values, every matrix can be transformed to a canonical form in $O(|V|^3)$ time. Once in the canonical form, the comparison of sets for inclusion and equality is simple ($O(|V|^2)$ time).

Surjective partitions

Another minimization procedure which preserves the language of an automaton $A = (S, \mathbf{I}, \mathbf{T}, \mathbf{F})$ is based on the notion of *surjection* which is dual to stability. Recall that given some partition $\rho = \{S_1, \ldots, S_{|\rho|}\}$ of the state space, we say that some $S_i \in \rho$ is stable with respect to the transition relation $\mathbf{T}$ if for every I/O value σ and all $S_j \in \rho$:

$$\exists s_i \in S_i \,.\, \exists s_j \in S_j \,.\, \mathbf{T}(s_i, \sigma, s_j) = 1 \implies \forall s_i \in S_i \,.\, \exists s_j \in S_j \,.\, \mathbf{T}(s_i, \sigma, s_j) = 1 \ .$$

Similarly, we say that $S_i \in \rho$ is *surjective* with respect to $\mathbf{T}$ if for every I/O value σ and all $S_j \in \rho$:

$$\exists s_i \in S_i \,.\, \exists s_j \in S_j \,.\, \mathbf{T}(s_i, \sigma, s_j) = 1 \implies \forall s_j \in S_j \,.\, \exists s_i \in S_i \,.\, \mathbf{T}(s_i, \sigma, s_j) = 1 \ . \qquad (4.2)$$

Also, we say that ρ is surjective with respect to $\mathbf{T}$ if every of its reachable elements is.

If ρ is surjective with respect $\mathbf{T}$, and it respects initial states (i.e. for each $Q \in \rho$ either $Q \subseteq \mathcal{S}(\mathbf{I})$ or $Q \cap \mathcal{S}(\mathbf{I}) = \emptyset$), then

$$\mathcal{L}(A) = \mathcal{L}(A_{/\rho}) \ .$$

The proof is similar to one for stable partitions. The only difference is that given some accepting run $S_0, S_1, \ldots, S_n$ of abstract states, we construct an accepting run $s_0, \ldots, s_n$ of concrete states backwards, starting from some $s_n \in S_n \cap \mathcal{S}(\mathbf{F})$.

A surjective partition ρ can be constructed on the fly, as shown in Figure 4.3. Initially, ρ contains only classes containing initial states. After we explore all successors from some element of ρ, we mark it to make sure it is explored only once. The function refine can refine ρ in two ways: either some classes of states visited for the first time are added to ρ, or some classes already in ρ are split such that after splitting ρ respects $post(S_i, \mathbf{T}, \sigma)$. In either case, the newly created classes are not marked even if a class being split is marked. The algorithm returns a surjective partition of the reachable subset of the state space S.

```
        function surjective (A, Σ)
          /* A = (S, I, T, F) - an automaton */
          /* Σ - a set of I/O values */
step 1:   ρ := {{(s, 0,...,0)}| I(s) = 1};
                      |V| times
          while there exist unmarked S_i ∈ ρ do
step 2:     mark S_i;
            forall I/O values σ ∈ Σ do
step 3:       ρ :=refine(ρ, post(S_i, T, σ));
            end forall
          end while
step 4:   return ρ;
        end
```

Figure 4.3: Generating a surjective partition.

Alur et al. [ACD+92] have implemented a variant of the algorithm in Figure 4.3

that computes $A_{/\rho}$ for some timed automaton. Again, every class of ρ is represented by a single untimed component and by a difference bound matrix. A similar algorithm is also implemented in another real-time extension of COSPAN by Courcoubetis et al. [CDCT93].

4.3.3 Other formalisms

Several other models have been proposed for real-time systems [dBHdRR91, Vyt91, AH92]. Many of these formalisms have undecidable verification problems, and those that are decidable use in some form the equivalence theorem of Alur and Dill. Thus it is not surprising that even though the syntax of the models are quite different, the implementations of the verification algorithms share the same techniques.

For example, Alur, Courcoubetis and Dill [ACD90] define *timed computation tree logic* (TCTL), a temporal logic for specifying real-time properties. Models of TCTL are *timed graphs*, which are finite-state structures similar to timed automata. They show that one can check whether a given timed graph is a model of a TCTL formula by traversing a finite-state structure similar to a region automaton. Henzinger et al. [HNSY92] show how this structure can be traversed symbolically, using predicates on timers and state variables to represent sets of states. That algorithm is implemented in the verification tool KRONOS [NSY92]. In the implementation, untimed components of the state are explicitly enumerated, and sets of values of timers are again represented by difference bound matrices. It is not clear at this time whether this limitation is intrinsic, i.e. finding an representation that combines an efficient symbolic representation of timer values (such as difference bound matrices) with an efficient symbolic representation of untimed state components (such as BDD's) is still an open problem.

Another approach is based on *timed Petri nets* [YSSC93, RM94]. The straightforward implementation requires a traversal of (equivalents of) region automata, but in a more careful implementation all the untimed states generated by different orderings of independent events are considered equivalent, and only a single representative is explored. Petri nets are well suited for the approach, because they allow easy recognition of independent events, but similar approaches could in principle be applied to other formalisms as well.

4.4 Iterative verification of timed automata

In this section we present an iterative algorithm for language emptiness of timed automata (henceforth TLE, for Timed Language Emptiness). The presented algorithm is a specialization of the generic iterative abstraction algorithm of section 1.3. For simplicity, we first focus on timed automata where all timing constraints are of the form $\mathbf{x} \leq c$ or $\mathbf{x} \geq c$. Later, in section 4.4.6 we extend the algorithm to include timing constraints of the form $\mathbf{x} - \mathbf{y} \leq c$. The algorithm can also be extended into a semi-decision procedure for language emptiness of arbitrary TAD's. That extension is given in chapter 6.

4.4.1 Overview

The TLE algorithm is shown in Figure 4.4. We start the verification process by relaxing all the timing constraints in step 1. If the verification in step 2 succeeds, we have verified the task. If the verification fails, verify returns run r of some sequence in $\mathcal{L}(A)$. If r violates no timing constraints, the language of the original system is not empty and the verification fails. We will show in section 4.4.2 that if the run does violate some timing constraints, they can always be represented in form of a loop (L in Figure 4.4). In that case, we modify A in a way that it still represents an abstraction of the region automaton, but that the behavior causing violations in L is eliminated. We repeat this process until the verification is terminated, either successfully or unsuccessfully.

The function verify_timed returns $NULL$ if the language of timed automaton A is empty, and otherwise it returns an accepting run of some sequence in the language. It calls three auxiliary functions: verify of section 2.6, analyze, and modify. TLE algorithm does not depend on a particular algorithm used in verify, as long as it generates a failure report in the form of a run. Therefore, any advances in the verification of untimed systems can immediately be incorporated into the TLE algorithm.

4.4.2 Failure analysis

Assume that the function verify in Figure 4.4 returns a run $s_0, s_1, \ldots, s_n$. We want to check whether there exists some consistent timing $\delta_1, \ldots, \delta_n$ of that run. With every transition $s_i \rightarrow s_{i+1}$ we can associate some inequalities that a consistent timing must satisfy. Consider for example, the following run of the railroad crossing model from

```
        function verify_timed (A)
          /* A = (S, I, T, F, V, C, M) - a timed automaton */
step 1:   Ã := (S, I, T, F);
          while TRUE do
step 2:     if (r :=verify(Ã)) = NULL then return NULL;
step 3:     if (L :=analyze(A, Ã, r)) = NULL then return r;
step 4:     Ã :=modify(Ã, L);
          end while
        end
```

Figure 4.4: TLE – timed language emptiness algorithm

Figure 4.1:

$$\begin{array}{ll} TRAIN: & \underbrace{\left[\begin{array}{l} OUT \\ RAISE \\ UP \end{array}\right]}_{s_0} \longrightarrow \underbrace{\left[\begin{array}{l} APP \\ R2L \\ UP \end{array}\right]}_{s_1} \longrightarrow \underbrace{\left[\begin{array}{l} APP \\ LOWER \\ U2D \end{array}\right]}_{s_2} \longrightarrow \underbrace{\left[\begin{array}{l} IN \\ LOWER \\ U2D \end{array}\right]}_{s_3}, \\ CONT.: & \\ GATE: & \end{array} \tag{4.3}$$

Among other constraints, a proper timing must satisfy:

$$\delta_2 \leq 1 , \tag{4.4}$$

$$\delta_3 < 1 , \tag{4.5}$$

$$\delta_2 + \delta_3 \geq 2 , \tag{4.6}$$

Inequality (4.4) must hold because $s_1 \rightarrow s_2$ is enabled only if $\mathbf{y} \leq 1$, and $\mathbf{y}$ was reset on $s_0 \rightarrow s_1$. Similarly, (4.6) must hold because $s_2 \rightarrow s_3$ is enabled only if $\mathbf{z} \geq 2$, and $\mathbf{z}$ was last reset on $s_0 \rightarrow s_1$, and so on. Not all inequalities that a consistent timing must satisfy are listed above, but even these three do not have a solution, therefore the sequence (4.3) does not admit a consistent timing.

Besides explicit constraints specified by timers, a consistent timing must also satisfy $\delta_i > 0$ (time is strictly increasing). To be able to treat these constraints as any other, we assume (without loss of generality) that there exists a timer $\mathbf{x} \in V$, such that it is reset on

every transition, and $\mathbf{x} > 0$ is an enabling condition of every transition (i.e. $\mathbf{M}(s,q,\mathbf{x}) = \perp$ and $(\mathbf{x} > 0) \in \mathbf{C}(s,q)$) for all $s,q \in S$).

One way to check whether a failure report violates any timing constraints is to list all inequalities implied by it and check for feasibility. But, closer examination of these inequalities shows that they are of special form, and that more efficient graph algorithms can be applied.

Let $\lambda_i = \sum_{j=1}^{i} \delta_j$ for all $i = 1,\ldots,n$ be the time of the i-th transition given that elapsed times between transitions are: $\delta_1,\ldots,\delta_n$. Also, define $\lambda_0 = 0$. Then, every inequality implied by a failure report can be written in a form:

$$\lambda_i - \lambda_j \leq c \ ,$$

for some $i,j \in \{0,\ldots,n\}$ and some bound c. It is well known (e.g. [Law76]) that such a system of linear inequalities is feasible if and only if there are no negative weighted loops in the graph constructed as follows:

- nodes of the graph are $0,\ldots,n$,
- for every inequality $\lambda_i - \lambda_j \leq c$ add an edge from i to j weighted c.

Finding a negative weighted loop in a graph is a well understood problem for which a polynomial algorithm exists [Tar83].

To analyze a failure report we build a graph similar to the one described above, except that we assign to edges not only their weight, but also some additional information to be used later in the failure elimination phase. The algorithm for building the graph is shown in Figure 4.5. In step 1, we create one node for every transition in the failure report, plus node 0. For every inequality $\mathbf{x} \leq c$ or $\mathbf{x} \geq c$ we first compute k, the index of the transition where $\mathbf{x}$ was last reset (step 2). A constraint of the form $\mathbf{x} \leq c$ induces the inequality $\lambda_i - \lambda_k \leq c$, therefore in step 4 we create an edge from i to k weighted c. A constraint of the form $\mathbf{x} \geq c$ induces the inequality $\lambda_k - \lambda_i \leq -c$, and the corresponding edge is created in step 5. Finally, in step 4 or 5 we label the newly created edge with the timer in ψ, and with a characteristic function of some set of edges constrained by ψ. Note that in Figure 4.5 we do not specify $\mathbf{E}$ exactly, we only require that it contains $s_{i-1} \rightarrow s_i$. Some of the edges characterized by $\mathbf{E}$ will be deleted from the current abstraction of the system, if ψ cannot be satisfied. The larger $\mathcal{S}(\mathbf{E})$ we select, the more behaviors will be

```
        function graph(A, s_0, ..., s_n)
          /* A = (S, I, T, F, V, C, M) - a timed automaton */
          /* s_0, ..., s_n - a failure trace in the abstraction Ã */
step 1:   let G be a graph with nodes 0, ..., n and no edges
          for i = 1, ..., n do
            for each ψ ∈ C(s_{i-1}, s_i) of the form x ≤ c or x ≥ c do
step 2:       let k := max{j < i | M(s_{j-1}, s_j, x) = ⊥ or j = 0};
                /* k is the last place where x was reset */
step 3:       choose E satisfying (s_{i-1}, s_i) ∈ set(E) ⊆ {(s, q) | ψ ∈ C(s, q)};
              if ψ is of the form x ≤ c then
step 4:         add to G an edge i → k weighted c, and labeled x, E;
              else  /* ψ is of the form x ≥ c; */
step 5:         add to G an edge k → i weighted −c, and labeled x, E;
              end if
            end for each
          end for
          return G;
        end
```

Figure 4.5: Forming a graph for failure analysis.

eliminated in every iteration, thus the algorithm will converge faster. On the other hand, to keep the abstraction small, we want a representation of the function **E** to be as small as possible. This freedom can be used to fine tune the algorithm.

For example, for the constraint $\mathbf{x} < 5$ associated with the transition $s_2 \rightarrow s_3$ of the failure trace (4.3) one possible choice of **E** is $\mathbf{ps}_{TRAIN} = APP$. This particular **E** represents all edges in the product machine that have $TRAIN$ component either $APP \rightarrow APP$ or $APP \rightarrow IN$. Another choice could be $\mathbf{ps}_{TRAIN} \neq OUT$ which completely characterizes the set of edges constrained by $\mathbf{x} < 5$.

In what follows, we will use a 5-tuple $(i, j, c, \mathbf{x}, \mathbf{E})$ to denote an edge from i to j, weighted c, and labeled with **x** and **E**.

Once a graph has been formed, we can invoke a standard algorithm for finding a negative weighted loop (also called an *over-constrained loop*). The function analyze returns such a loop, if one exists in the graph, otherwise it returns $NULL$. The over-constrained loop is an input to the modify function that eliminates the failure trace. In doing so , it uses a set of "auxiliary" automata called *guard automata*, which we introduce next.

4.4.3 Guard automata

Consider, for example, the safety property of the railroad crossing. When all the timing constraints are relaxed, the property fails, and a possible failure trace is:

$$\begin{array}{ll} TRAIN: & \underbrace{\left[\begin{array}{l} OUT \\ RAISE \\ UP \end{array}\right]}_{s_0} \longrightarrow \underbrace{\left[\begin{array}{l} APP \\ R2L \\ UP \end{array}\right]}_{s_1} \longrightarrow \underbrace{\left[\begin{array}{l} IN \\ R2L \\ UP \end{array}\right]}_{s_2} . \\ CONT.: & \\ GATE: & \end{array} \tag{4.7}$$

This sequence does not admit a consistent timing, as shown by the following over-constrained loop:

$$\underbrace{(s_0 \rightarrow s_1)}_{\text{node 1}} \xrightarrow{x \geq 2} \underbrace{(s_1 \rightarrow s_2)}_{\text{node 2}} \xrightarrow{y<1} \underbrace{(s_0 \rightarrow s_1)}_{\text{node 1}}, \tag{4.8}$$

We know that $s_1 \rightarrow s_2$ is enabled only if both $\mathbf{x} \geq 2$ and $\mathbf{y} < 1$ are satisfied, which implies that it can be enabled only if $\mathbf{x} - \mathbf{y} > 1$. To eliminate this failure trace, we will keep track of $\mathbf{x} - \mathbf{y}$ by composing the current abstraction of the system with the *guard automaton* $\langle \mathbf{y} - \mathbf{x} < -1 \rangle$ that has two states: "*good*" one corresponding to all valuations of $\mathbf{x}$ and $\mathbf{y}$ satisfying $\mathbf{x} - \mathbf{y} > 1$, and the "*bad*" corresponding to valuations satisfying $\mathbf{x} - \mathbf{y} \leq 1$. In the composition, we can safely eliminate transitions where $s_1 \rightarrow s_2$ occurs while $\langle \mathbf{y} - \mathbf{x} < -1 \rangle$ is in the bad state, because they are subject to conflicting timing constraints.

The automaton $\langle \mathbf{y} - \mathbf{x} < -1 \rangle$ cannot track the value of $\mathbf{x} - \mathbf{y}$ exactly because it can access only incomplete information: it can observe the untimed state components to check whether timers are reset or not, but it cannot know the exact time between transitions. This incompleteness induces non-determinism: if different transition times can lead to different states, a guard automaton can non-deterministically choose between the two. Note that the use of word "choose" is somewhat misleading: the set of accepting runs will include both choices. More precisely, the rules for building the transition relation of an automaton $\langle \mathbf{y} - \mathbf{x} < -1 \rangle$ are as follows:

1. initially it is in the bad state and whenever both $\mathbf{x}$ and $\mathbf{y}$ are reset it must move there (because $\mathbf{x} = \mathbf{y} = 0$ in these cases),
2. as long as neither $\mathbf{x}$ nor $\mathbf{y}$ are reset, it must not change the state (in this case both $\mathbf{x}$ and $\mathbf{y}$ are incremented by the same real number, so $\mathbf{x} - \mathbf{y}$ remains constant),
3. if one of the timers is reset and the other is not, then the automaton $\langle \mathbf{y} - \mathbf{x} < -1 \rangle$ can either change states or remain in the same state (non-determinism).

Composing the initial abstraction with the automaton $\langle \mathbf{y} - \mathbf{x} < -1 \rangle$ and disabling $s_1 \rightarrow s_2$ while it is in the bad state is enough to eliminate the failure trace (4.7), because $\langle \mathbf{y} - \mathbf{x} < -1 \rangle$ must move to the bad state when both $\mathbf{x}$ and $\mathbf{y}$ are reset on $s_0 \rightarrow s_1$. In fact, this action eliminates all the sequences that have a subsequence that satisfies the following:

- both $\mathbf{x}$ and $\mathbf{y}$ are reset on the first transition,
- the last transition is $s_1 \rightarrow s_2$,
- neither $\mathbf{x}$ nor $\mathbf{y}$ are reset in between.

Before we define guard automata formally, we need a bit of notation. Given some timer $\mathbf{x}$, we use $\mathbf{R_x}$ to denote the characteristic function of all edges on which $\mathbf{x}$ is reset, i.e.:

$$\mathbf{R_x} \equiv \mathcal{X}\{(s,q) \,|\, \mathbf{M}(s,q,\mathbf{x}) = \perp\} \ .$$

For example, for timer $\mathbf{x}$ in Figure 4.1:

$$\mathbf{R_x} \equiv (\mathbf{ps}_{TRAIN} = OUT) * (\mathbf{ns}_{TRAIN} = APP) \ .$$

Formally, for any two timers $\mathbf{x}$ and $\mathbf{y}$ and any bound $c \geq 0$ let the *guard automaton* $\langle \mathbf{x} - \mathbf{y} \leq c \rangle$ be defined by:

$$\langle \mathbf{x} - \mathbf{y} \leq c \rangle = (\{good, bad\}, \mathcal{X}\{good\}, \mathbf{T}_{\langle \mathbf{x}-\mathbf{y} \leq c \rangle}, \mathbf{1}) \ ,$$

where:[3]

$$\begin{aligned} \mathbf{T}_{\langle \mathbf{x}-\mathbf{y} \leq c \rangle} \equiv \ & \overline{\mathbf{R_x}} * \overline{\mathbf{R_y}} * (\mathbf{ps}_{\langle \mathbf{x}-\mathbf{y} \leq c \rangle} = \mathbf{ns}_{\langle \mathbf{x}-\mathbf{y} \leq c \rangle}) + \\ & \mathbf{R_x} * \mathbf{R_y} * (\mathbf{ns}_{\langle \mathbf{x}-\mathbf{y} \leq c \rangle} = good) + \\ & \overline{\mathbf{R_x}} * \mathbf{R_y} + \mathbf{R_x} * \overline{\mathbf{R_y}} \ . \end{aligned}$$

[3] A formula $\mathbf{F} = \mathbf{G}$ is an abbreviation for $\mathbf{F} * \mathbf{G} + \overline{\mathbf{F}} * \overline{\mathbf{G}}$.

The first line corresponds to the requirement that $x - y$ cannot change unless either x or y are reset. The second line indicates that when both x and y are reset, the guard automaton must move to the state that contains the valuation $(0,0)$. Finally, the third line indicate that the next state can be arbitrary if one of the timers is reset and the other one is not.[4]

When c is a negative upper bound, the definition changes slightly, as shown in Figure 4.6. It is also useful to extend the definition for the case $x = y$ and $c < 0$ as follows:

$$A_{(x-x\leq c)} = (\{bad\}, 1, 1, 1) .$$

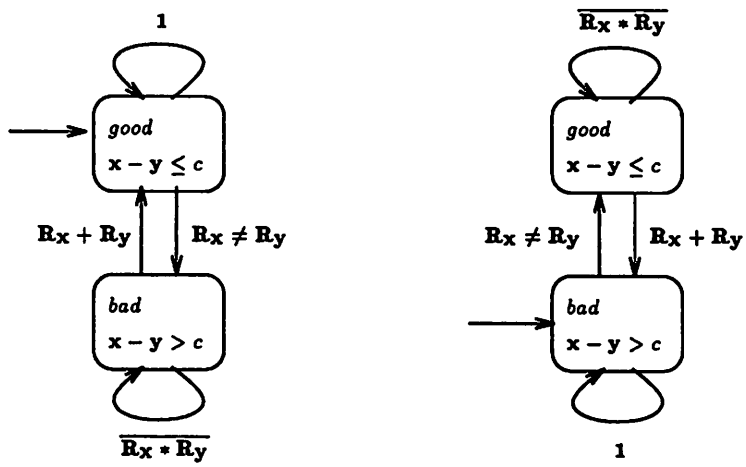


Figure 4.6: Guard automata.

Note that guard automata are the same for all positive bounds. They will be distinguished only later, in the failure elimination phase, when we compose the current abstraction of the system with some guard automata, and then eliminate some transitions from the composition. Which transitions can or cannot be safely eliminated depends on the constant c. It might seem redundant to have in the abstraction of the system several components with identical transition relations. But, since the transition relation is non-deterministic and the automata are independent, not all of the components are always in the same state in every run. In fact, after some transitions are eliminated from the

[4]Based on the reseting information, it is possible to a define less non-deterministic transition relation. For example, if $c \geq 0$, and x is reset and y is not, then the next state must be *good*. However, the conservative assumption that the next state could also be *bad* does not affect our algorithm and provides for a simpler presentation.

composition, it might be the case that two guard automata have to be in different states in all accepting runs.

We think of states of guard automata as abstractions of states of the companion automaton A^∞ of some timed automaton $A = (S, \mathbf{I}, \mathbf{T}, \mathbf{F}, V, \mathbf{C}, \mathbf{M})$. To make this relation precise, let $\mathcal{V}$ be the set that contains variables **ps**, **ns**, σ (the state and I/O variables of A), and the state variables $\mathbf{ps}_{\langle \mathbf{x}_i - \mathbf{x}_j \leq c \rangle}$ and $\mathbf{ns}_{\langle \mathbf{x}_i - \mathbf{x}_j \leq c \rangle}$ of all guard automata, and let the *abstraction function* ϕ that maps every every transition $t = (s, x_1, \ldots, x_{|V|}) \xrightarrow{\sigma,\delta} (q, y_1, \ldots, y_{|V|})$ of A^∞ to an assignment of variables in $\mathcal{V}$ be defined as follows:

$$
\begin{aligned}
\phi(t)(\sigma) &= \sigma \ , \\
\phi(t)(\mathbf{ps}) &= s \ , \\
\phi(t)(\mathbf{ns}) &= q \ , \\
\phi(t)(\mathbf{ps}_{\langle \mathbf{x}_i - \mathbf{x}_j \leq c \rangle}) &= \begin{cases} \mathit{good} & \text{if } x_i - x_j \leq c \ , \\ \mathit{bad} & \text{if } x_i - x_j > c \ , \end{cases} \\
\phi(t)(\mathbf{ns}_{\langle \mathbf{x}_i - \mathbf{x}_j \leq c \rangle}) &= \begin{cases} \mathit{good} & \text{if } y_i - y_j \leq c \ , \\ \mathit{bad} & \text{if } y_i - y_j > c \ . \end{cases}
\end{aligned}
$$

If $\mathbf{E}$ is some formula over variables in $\mathcal{V}$, then we use $\mathbf{E}(\phi(t))$ to denote the value obtained by:

1. applying ϕ to t to obtain the assignment for variables in $\mathit{supp}(\mathbf{E})$, and then
2. evaluating $\mathbf{E}$ with that assignment.

The transition relation of guard automata contains abstractions of all the transitions in A^∞, as claimed by the following proposition:

Proposition 4.1 *Let* $t = (s, x_1, \ldots, x_{|V|}) \xrightarrow{\sigma,\delta} (q, y_1, \ldots, y_{|V|})$ *be some transition of the companion automaton* A^∞ *and let* $(\{\mathit{bad}, \mathit{good}\}, \mathbf{I}_{\langle \mathbf{x}_i - \mathbf{x}_j \leq c \rangle}, \mathbf{T}_{\langle \mathbf{x}_i - \mathbf{x}_j \leq c \rangle}, \mathbf{1})$ *be some guard automaton. Then:*

1. $\mathbf{T}_{\langle \mathbf{x}_i - \mathbf{x}_j \leq c \rangle}(\phi(t)) = 1$,
2. *if* $x_i = x_j = 0$, *then* $\mathbf{I}_{\langle \mathbf{x}_i - \mathbf{x}_j \leq c \rangle}(\phi(t)) = 1$.

Proof. If $i = j$ then the proposition follows trivially because initial states and transitions relations are **1** in that case.

Recall that if $i \neq j$, then $\mathbf{I}_{\langle \mathbf{x}_i - \mathbf{x}_j \leq c \rangle} \equiv (\mathbf{ps}_{\langle \mathbf{x}_i - \mathbf{x}_j \leq c \rangle} = init)$, where *init* is *good* if $c \geq 0$ and *bad* if $c < 0$. In this case, part 2 follows because $x_i = x_j = 0$ implies that:

$$\phi(t)(\mathbf{ps}_{\langle \mathbf{x}_i - \mathbf{x}_j \leq c \rangle}) = init = \begin{cases} good & \text{if } c \geq 0 = x_i - x_j \ , \\ bad & \text{if } c < 0 = x_i - x_j \ . \end{cases}$$

Also recall that if $i \neq j$:

$$\begin{aligned} \mathbf{T}_{\langle \mathbf{x}_i - \mathbf{x}_j \leq c \rangle} \equiv \ & \overline{\mathbf{R}_{\mathbf{x}_i}} * \overline{\mathbf{R}_{\mathbf{x}_j}} * (\mathbf{ps}_{\langle \mathbf{x}_i - \mathbf{x}_j \leq c \rangle} = \mathbf{ns}_{\langle \mathbf{x}_i - \mathbf{x}_j \leq c \rangle}) + \\ & \mathbf{R}_{\mathbf{x}_i} * \mathbf{R}_{\mathbf{x}_j} * (\mathbf{ns}_{\langle \mathbf{x}_i - \mathbf{x}_j \leq c \rangle} = init) + \\ & \overline{\mathbf{R}_{\mathbf{x}_i}} * \mathbf{R}_{\mathbf{x}_j} + \mathbf{R}_{\mathbf{x}_i} * \overline{\mathbf{R}_{\mathbf{x}_j}} \ . \end{aligned}$$

According to the definition of the companion automaton, to prove part 1, we need to consider the following cases:

Case 1: $\mathbf{M}(s, q, \mathbf{x}_i) = \mathbf{M}(s, q, \mathbf{x}_j) = 0$ and $x_i - x_j = y_i - y_j$. The claim follows because in this case:

$$\mathbf{R}_{\mathbf{x}_i}(\phi(t)) = \mathbf{R}_{\mathbf{x}_i}(s, q) = \mathbf{R}_{\mathbf{x}_j}(\phi(t)) = \mathbf{R}_{\mathbf{x}_j}(s, q) = 0 \ ,$$

and:

$$\phi(t)(\mathbf{ps}_{\langle \mathbf{x}_i - \mathbf{x}_j \leq c \rangle}) = \phi(t)(\mathbf{ns}_{\langle \mathbf{x}_i - \mathbf{x}_j \leq c \rangle}) = \begin{cases} good & \text{if } x_i - x_j = y_i - y_j \leq c \ , \\ bad & \text{if } x_i - x_j = y_i - y_j > c \ , \end{cases}$$

implying that:

$$\left(\overline{\mathbf{R}_{\mathbf{x}_i}} * \overline{\mathbf{R}_{\mathbf{x}_j}} * (\mathbf{ps}_{\langle \mathbf{x}_i - \mathbf{x}_j \leq c \rangle} = \mathbf{ns}_{\langle \mathbf{x}_i - \mathbf{x}_j \leq c \rangle})\right)(\phi(t)) = 1 \ .$$

Case 2: $\mathbf{M}(s, q, \mathbf{x}_i) = \mathbf{M}(s, q, \mathbf{x}_j) = \perp$ and $y_i = y_j = 0$. The claim follows because in this case:

$$\mathbf{R}_{\mathbf{x}_i}(\phi(t)) = \mathbf{R}_{\mathbf{x}_i}(s, q) = \mathbf{R}_{\mathbf{x}_j}(\phi(t)) = \mathbf{R}_{\mathbf{x}_j}(s, q) = 1 \ ,$$

and:

$$\phi(t)(\mathbf{ns}_{\langle \mathbf{x}_i - \mathbf{x}_j \leq c \rangle}) = init = \begin{cases} good & \text{if } c \geq 0 = y_i - y_j \ , \\ bad & \text{if } c < 0 = y_i - y_j \ . \end{cases}$$

implying that:

$$\left(\mathbf{R}_{\mathbf{x}_i} * \mathbf{R}_{\mathbf{x}_j} * (\mathbf{ns}_{\langle \mathbf{x}_i - \mathbf{x}_j \leq c \rangle} = init)\right)(\phi(t)) = 1 \ .$$

Case 3: $\mathbf{M}(s,q,\mathbf{x}_i) = 0$ and $\mathbf{M}(s,q,\mathbf{x}_j) = \perp$. The claim follows because:

$$(\overline{\mathbf{R}_{\mathbf{x}_i}} * \mathbf{R}_{\mathbf{x}_j})(\phi(t)) = \overline{\mathbf{R}_{\mathbf{x}_i}(s,q)} * \mathbf{R}_{\mathbf{x}_j}(s,q) = 1$$

in this case.

Case 4: $\mathbf{M}(s,q,\mathbf{x}_i) = \perp$ and $\mathbf{M}(s,q,\mathbf{x}_j) = 0$. The claim follows because:

$$(\mathbf{R}_{\mathbf{x}_i} * \overline{\mathbf{R}_{\mathbf{x}_j}})(\phi(t)) = \mathbf{R}_{\mathbf{x}_i}(s,q) * \overline{\mathbf{R}_{\mathbf{x}_j}(s,q)} = 1$$

in this case.

□

Note that the converse of Proposition 4.1 does not hold: some transitions of guard automata are not abstractions of any transitions in A^∞. We have chosen this particular definition of guard automata because it is the simplest one that satisfies our purposes, but our approach would not change if the definition is replaced with one with less transitions which still satisfies Proposition 4.1. Actually, an interesting trade-off could be explored here. Composing a system with a more restrictive guard automata eliminates more behavior, and thus can lead to fewer iterations. On the other hand, more restrictive automata would have to be more complex (in terms of BDD size), thus the complexity of every iterations could be increased.

4.4.4 Failure elimination

In this section we describe the modify function used in the verification procedure in Figure 4.4. The inputs to modify are the current abstraction of the system $\tilde{A}$ and an over-constrained loop L, and it returns a modified abstraction of the system, in which the failure report that has induced L is no longer a run.

The modify function is shown in Figure 4.7. We start by selecting any two adjacent edges $(i,k,w,\mathbf{x},\mathbf{E})$ and $(k,j,v,\mathbf{y},\mathbf{H})$ of the over-constrained loop induced by constraints $\mathbf{x} \geq -w$, and $\mathbf{y} \leq v$ (i.e. one backward and one forward pointing edge). Then, we construct a guard automaton $\langle \mathbf{y} - \mathbf{x} \leq w + v \rangle$ (step 1), and compose it with the current abstraction of the system (step 2). Recall that transitions characterized by $\mathbf{E}$ are constrained by $\mathbf{x} \geq -w$, and that transitions characterized by $\mathbf{H}$ are constrained by $\mathbf{y} \leq v$. Thus, transitions in their intersection cannot occur if $\mathbf{y} - \mathbf{x} > w + v$, i.e. if $\langle \mathbf{y} - \mathbf{x} \leq w + v \rangle$ is in the *bad* state.

function modify($L, \tilde{A}$)
  /* L - an over-constrained loop */
  /* $\tilde{A}$ - an abstraction of the system to be verified */
  while there are edges $(i,k,w,\mathbf{x},\mathbf{E})$, $(k,j,v,\mathbf{y},\mathbf{H})$ in L s.t. $k > i,j$ **do**
step 1: $A_k := \langle \mathbf{y} - \mathbf{x} \leq w + v \rangle$;
step 2: $\tilde{A} := \tilde{A} \otimes A_k$;
step 3: $\tilde{A} := (S_{\tilde{A}}, \mathbf{I}_{\tilde{A}}, \mathbf{T}_{\tilde{A}} * \overline{(\mathbf{ps}_{A_k} = bad) * \mathbf{E} * \mathbf{H}}, \mathbf{F}_{\tilde{A}})$;
step 4: remove from L edges $(i,k,w,\mathbf{x},\mathbf{E})$ and $(k,j,v,\mathbf{y},\mathbf{H})$ and node k;
step 5: **if** $i < j$ **then** add to L an edge $(i,\ j,\ w+v,\ \mathbf{x},\ (\mathbf{ns}_{A_k} = good) * \mathbf{R}_y * \overline{\mathbf{R}_x})$;
step 6: **else if** $j < i$ **then** add to L an edge $(i,\ j,\ w+v,\ \mathbf{y},\ (\mathbf{ns}_{A_k} = good) * \mathbf{R}_x * \overline{\mathbf{R}_y})$;
step 7: **else** /* $i = j$ */ **return** $\tilde{A}$;
  end while
end function

Figure 4.7: Failure elimination procedure for timed automata.

Therefore, in step 3 we delete corresponding transitions from the current abstraction. Recall that by step 3 in Figure 4.5, both $\mathbf{E}$ and $\mathbf{H}$ must contain the k-the transition of the failure report, therefore the failure trace cannot be completed if $\langle \mathbf{y} - \mathbf{x} \leq w + v \rangle$ is in the bad state after $k-1$ transitions.

At this point we do not have as yet precise information whether $\langle \mathbf{y} - \mathbf{x} \leq w + v \rangle$ is in the *good* or *bad* after $k-1$ transitions of the failure report. However, we do know that that it cannot change states unless either $\mathbf{x}$ or $\mathbf{y}$ are reset. Therefore, we focus to the nearest previous transition in the failure trace where either $\mathbf{x}$ or $\mathbf{y}$ were reset, i.e. to transitions i or j, whichever is larger.

If $j > i$ (i.e. $\mathbf{y}$ was most recently reset), we use the fact the $\mathbf{y} - \mathbf{x} \leq w + v$ implies that $-\mathbf{x} \leq w+v$ when $\mathbf{y}$ is reset. We want $\langle \mathbf{y} - \mathbf{x} \leq w+v \rangle$ to track the truth of $\mathbf{y} - \mathbf{x} \leq w+v$ as close as possible, therefore in step 5 we assert that transitions characterized by $(\mathbf{ns}_{A_k} = good)$ can occur concurrently with the j-th transition of the failure trace (actually with any transition characterized by $\mathbf{R}_y$ and $\overline{\mathbf{R}_x}$) only if $-\mathbf{x} \leq w+v$. In other words, the action in step 5 follows from the observation that the failure trace can be completed only if $-\mathbf{x} \leq w + v$

at the j-th transition. We enforce this observation in step 3 of some later iteration by eliminating from the abstraction (under suitable conditions given by $\mathbf{H}$ in that iteration) the transitions characterized by $(\mathbf{ps}_{A_j} = \mathit{bad}) * (\mathbf{ns}_{A_k} = \mathit{good}) * \mathbf{R}_y * \overline{\mathbf{R}_x}$. This effectively "connects" A_j and A_k, and ensures that if A_j is in the *bad* state after $j-1$ transitions of the failure trace (and conditions specified by $\mathbf{H}$ are met), then A_k must be in the *bad* state after $k-1$ transitions. Informally, in step 5 we propagate the constraints backwards and in step 3 we connect guard automata so that *bad* states are propagated forwards. Step 6 is based on similar analysis in case $j < i$. Note that even though for some $\mathbf{x}$ and $\mathbf{y}$ and all $c \geq 0$ (or $c < 0$) all guard automata have initially the same transition relation, some of their transitions are (conditionally) disabled in step 3. Which transitions can be safely disabled under which conditions (without disabling abstract images of accepting runs in the companion automaton), depends on the actual value of c.

We iterate this process until there are no more edges in the loop (step 7). Since L is over-constrained not all of the generated constraints can be satisfied, and at least one A_k will be in the *bad* state, disabling the k-th transition and eliminating the failure trace.

For example, consider the failure trace (4.3), and the following over-constrained loop induced by it:

$$\underbrace{(s_0, s_1)}_{\text{node 1}} \xrightarrow{\mathbf{x}\geq 2} \underbrace{(s_2 \rightarrow s_3)}_{\text{node 3}} \xrightarrow{\mathbf{z}<1} \underbrace{(s_1 \rightarrow s_2)}_{\text{node 2}} \xrightarrow{\mathbf{y}\leq 1} \underbrace{(s_0 \rightarrow s_1)}_{\text{node 1}}. \tag{4.9}$$

The edges in the loop are: $(1, 3, -2, \mathbf{x}, \mathbf{E})$, $(3, 2, 1^-, \mathbf{z}, \mathbf{H})$, and $(2, 1, 1, \mathbf{y}, \mathbf{D})$, where $\mathcal{S}(\mathbf{E}*\mathbf{H})$ must contain $s_2 \rightarrow s_3$ and $\mathcal{S}(\mathbf{D})$ must contain $s_1 \rightarrow s_2$.

In the first pass through the **while** loop, $\tilde{A}$ is composed with $A_3 = \langle \mathbf{z} - \mathbf{x} < -1 \rangle$ (steps 1 and 2), the transitions characterized by:

$$(\mathbf{ps}_{A_3} = \mathit{bad}) * \mathbf{E} * \mathbf{H} \tag{4.10}$$

are disabled (step 3), and a constraint $\mathbf{x} > 1$ is placed on $s_1 \rightarrow s_2$ (step 5). More precisely, an edge:

$$(1,\ 2,\ -1^-,\ \mathbf{x},\ (\mathbf{ns}_{A_3} = \mathit{good}) * \mathbf{R}_\mathbf{z} * \overline{\mathbf{R}_\mathbf{x}})$$

is added to L. In the second and final pass through the **while** loop, the current abstraction is composed with $A_2 = \langle \mathbf{y} - \mathbf{x} < 0 \rangle$, and transitions characterized by:

$$(\mathbf{ps}_{A_2} = \mathit{bad}) * \mathbf{D} * (\mathbf{ns}_{A_3} = \mathit{good}) * \mathbf{R}_z * \overline{\mathbf{R}_x} \tag{4.11}$$

are disabled.

Now, $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow s_3$ is eliminated because:

1. A_2 must move to the *bad* state when both **y** and **x** are reset on $s_0 \rightarrow s_1$, thus
2. $(\mathbf{ps}_{A_2} = bad)$, **D**, $\mathbf{R}_z$, and $\overline{\mathbf{R}_x}$ are all satisfied at node 2 ($s_1 \rightarrow s_2$), so by (4.11) $s_1 \rightarrow s_2$ can occur only if $\mathbf{ns}_{A_3} = bad$, but then
3. $(\mathbf{ps}_{A_3} = bad)$, **E** and **H** would all be satisfied at $s_2 \rightarrow s_3$, therefore by (4.10) $s_2 \rightarrow s_3$ cannot occur, since it is disabled under these conditions.

Incidentally, the safety property of the railroad crossing can be verified in three iterations of the verify_timed algorithm. Two iterations include calls to the modify function with the over-constrained loops (4.8) and (4.9) and the third includes a call to modify with the over-constrained loop induced by the failure trace:

$$\begin{array}{ll} TRAIN: & \underbrace{\left[\begin{array}{l} OUT \\ RAISE \\ UP \end{array}\right]}_{s_0} \longrightarrow \underbrace{\left[\begin{array}{l} APP \\ R2L \\ UP \end{array}\right]}_{s_1} \longrightarrow \underbrace{\left[\begin{array}{l} IN \\ LOWER \\ U2D \end{array}\right]}_{s_2} . \\ CONT.: & \\ GATE: & \end{array}$$

This trace is eliminated by composing $\tilde{A}$ with $A_2 = \langle \mathbf{y} - \mathbf{x} \leq -1 \rangle$, and disabling the transition $s_1 \rightarrow s_2$ while A_2 is in the *bad* state. Note that this is the same guard automaton as one used in eliminating the over-constrained loop (4.8). Hence, it can be reused. We use a hash table to detect if a guard automaton already exists.

It might happen that in step 1 **x** is the same timer as **y**. This can happen only in the last iteration (because $\mathbf{x} = \mathbf{y}$ implies $i = j$), thus $w + v < 0$ must hold (because $w + v$ in the last iteration is the total weight of the loop). Therefore, $(\mathbf{ps}_{A_k} = bad)$ is a tautology (by definition in section 4.4.3), and all transitions characterized by $\mathbf{E} * \mathbf{H}$ are disabled in step 3. This is indeed warranted by timing constraints, because **E** is constrained by $\mathbf{x} \geq -w$, and **H** is constrained by $\mathbf{x} \leq v$ which obviously is not consistent with $w + v < 0$.

4.4.5 Correctness

To show the correctness of the TLE algorithm we need to show that the function verify_timed:

1. returns $NULL$ only if the language of the timed automaton A is empty (no false positives),

2. returns a run r only if it is truly a run of some sequence in $\mathcal{L}(A)$ (no false negatives),

3. terminates in a finite number of steps.

We develop this result through a series of lemmas. In developing the proof we assume that the verify and analyze functions are correct, because for these two functions we just apply algorithms developed elsewhere.

Assumption 4.1 *The* verify *function returns $NULL$ if $\mathcal{L}(A)$ is empty, otherwise it returns a run of some string in $\mathcal{L}(A)$.*

Assumption 4.2 *The* analyze *function returns $NULL$ if the failure trace r violates no timing constraints, otherwise it returns a negative weighted cycle in a graph induced by r.*

To establish that TLE cannot produce a false positive result, we need the following lemma:

Lemma 4.1 *Let $(a, b, w, \mathbf{x}_i, \mathbf{E})$ be some edge in the over-constrained loop at some point of the TLE algorithm, let $t = (s, x_1, \ldots, x_{|V|}) \xrightarrow{\sigma,\delta} (q, y_1, \ldots, y_{|V|})$ be some transition of the companion automaton A^{∞}, and let ϕ be the abstraction function as defined in section 4.4.3. The following holds:*

1. *if $a > b$, then $\mathbf{E}(\phi(t)) = 1$ only if $x_i + \delta \leq w$,*

2. *if $a < b$, then $\mathbf{E}(\phi(t)) = 1$ only if $x_i + \delta \geq -w$,*

Proof.

Case $a > b$. If this case, the edge could have been created either in step 4 of the algorithm in Figure 4.5 or in step 6 of the algorithm in Figure 4.7. From step 3 in Figure 4.5 it follows that if $\mathbf{E}$ is created in step 4, then $\mathbf{E}(\phi(t)) = \mathbf{E}(s, q)$ is 1 only if $\mathbf{x}_i \leq w$ is in $\mathbf{C}(s, q)$, and therefore $x_i + \delta \leq w$ must hold, by condition 3 for the transition relation in the definition of the companion automaton.

If $(a, b, w, \mathbf{x}_i, \mathbf{E})$ is created in step 6 in Figure 4.7, then $\mathbf{E}$ is of the form:

$$\overline{\mathbf{R}_{\mathbf{x}_i}} * \mathbf{R}_{\mathbf{x}_j} * (\mathbf{ns}_{\langle \mathbf{x}_i - \mathbf{x}_j \leq w \rangle} = good) \ .$$

By the definition, $(\overline{\mathbf{R}_{\mathbf{x}_i}})(\phi(t)) = \overline{\mathbf{R}_{\mathbf{x}_i}(s,q)}$ is 1 if and only if $\mathbf{M}(s,q,\mathbf{x}_i) = 0$, and similarly $(\mathbf{R}_{\mathbf{x}_j})(\phi(t)) = \mathbf{R}_{\mathbf{x}_j}(s,q)$ is 1 if and only if $\mathbf{M}(s,q,\mathbf{x}_i) = \perp$. Therefore, $y_j = 0$ and $y_i = x_i + \delta$ must hold, by condition 2 for the transition relation in the definition of the companion automaton. On the other hand, $(\mathbf{ns}_{\langle \mathbf{x}_i - \mathbf{x}_j \leq w \rangle} = good)(\phi(t))$ is 1 if and only if $y_i - y_j \leq w$. Substituting $y_j = 0$ and $y_i = x_i + \delta$ completes the proof in this case.

Case $a > b$. If this case, the edge could have been created either in step 5 of the algorithm in Figure 4.5 or in step 5 of the algorithm in Figure 4.7. From step 3 of the algorithm in Figure 4.5 it follows that if $\mathbf{E}$ is created in step 5 of that algorithm, then $\mathbf{E}(\phi(t)) = \mathbf{E}(s,q)$ is 1 only if $\mathbf{x}_i \geq -w$ is in $\mathbf{C}(s,q)$, and therefore $x_i + \delta \geq -w$ must hold, by condition 4 for the transition relation in the definition of the companion automaton.

If $(a, b, w, \mathbf{x}_i, \mathbf{E})$ is created in step 5 of the algorithm in Figure 4.7, then $\mathbf{E}$ is of the form:

$$\overline{\mathbf{R}_{\mathbf{x}_i}} * \mathbf{R}_{\mathbf{x}_j} * (\mathbf{ns}_{\langle \mathbf{x}_j - \mathbf{x}_i \leq w \rangle} = good) \ .$$

By the same argument as in case $a > b$, $y_j = 0$, $y_i = x_i + \delta$, and $y_j - y_i \leq w$ must hold, thus a simple substitution completes the proof.

□

Theorem 4.3 *If* verified_timed *returns NULL, then the language of the timed automaton A is empty.*

Proof. Given the Assumption 4.1, it suffices to show that if:

$$\begin{pmatrix} s_0 \\ x_{1,0} \\ \vdots \\ x_{|V|,0} \end{pmatrix} \longrightarrow \begin{pmatrix} s_1 \\ x_{1,1} \\ \vdots \\ x_{|V|,1} \end{pmatrix} \longrightarrow \cdots \longrightarrow \begin{pmatrix} s_k \\ x_{1,k} \\ \vdots \\ x_{|V|,k} \end{pmatrix}$$

is an accepting run in A^∞ of some timed sequence $(\sigma_1, \delta_1), \ldots, (\sigma_k, \delta_k)$, then the assignments $\phi(t_1), \ldots, \phi(t_k)$ induce an accepting run in the abstraction $\tilde{A}$, where t_m is the m-th transition in the failure trace, i.e. for all $m = 1, \ldots, k$:

$$t_m \ = \ (s_{m-1}, x_{1,m-1}, \ldots, x_{|V|,m-1}) \stackrel{\sigma_m, \delta_m}{\longrightarrow} (s_m, x_{1,m}, \ldots, x_{|V|,m}) \ .$$

More precisely, we need to show:

Claim 1: $\phi(t_1)$ satisfies the characteristic function of the initial states of $\tilde{A}$.

Claim 2: the assignments to next state variables in $\phi(t_k)$ satisfies the characteristic function of the final states of $\tilde{A}$.

Claim 3: $\phi(t_m)$ satisfies the transition relation of $\tilde{A}$ for all $m = 1, \ldots, k$.

To show Claim 1, observe that at any point of the TLE algorithm, the characteristic function of initial states of $\tilde{A}$ is a conjunction of:

- the characteristic function $\mathbf{I}$ of initial states of A (by step 1 in Figure 4.4), and
- the characteristic functions of initial states of some guard automata (by steps 1 and 2 in Figure 4.7).

Now, $\mathbf{I}(\phi(t_1)) = \mathbf{I}(s_0)$ must be 1 by the definition of initial states of the companion automaton, and by the same definition $x_{1,0} = \ldots = x_{|V|,0} = 0$. Therefore by Proposition 4.1, $\mathbf{I}_G(\phi(t_1))$ is 1 for any guard automaton G (QED Claim 1).

To show Claim 2, observe that at any point of the TLE algorithm, the characteristic function of final states of $\tilde{A}$ is equal to the characteristic function $\mathbf{F}$ of final states of A. Therefore, $([\mathbf{F}]_{\mathbf{ps}\mapsto\mathbf{ns}})(\phi(t_k)) = \mathbf{F}(s_k)$ must be 1 by the definition of final states of the companion automaton (QED Claim 2).

To show Claim 3 observe that at any point of the TLE algorithm, the transition relation of $\tilde{A}$ is a conjunction of:

- the transition relation $\mathbf{T}$ of $\mathbf{A}$ (by step 1 in Figure 4.4), and
- the transition relations of some guard automata (by steps 1 and 2 in Figure 4.7), and
- expressions of the form: $\overline{(\mathbf{ps}_{\langle \mathbf{x}_i - \mathbf{x}_j \leq w+v \rangle} = bad) * \mathbf{E} * \mathbf{H}}$ where $\mathbf{E}$ and $\mathbf{H}$ are labels of some edges $(a, b, w, \mathbf{x}_j, \mathbf{E})$ and $(b, c, v, \mathbf{x}_i, \mathbf{H})$ satisfying $a < b$ and $b > c$ (by step 3 in Figure 4.7).

Therefore, we need to show that for all $m = 1, \ldots, k$:

Claim 3.1: $\mathbf{T}(\phi(t_m)) = 1$,

Claim 3.2: $\mathbf{T}_G(\phi(t_m)) = 1$ for any guard automaton G,

Claim 3.3: if $(a, b, w, x_j, \mathbf{E})$ and $(b, c, v, \mathbf{x}_i, \mathbf{H})$ are two edges in the over-constrained loop satisfying $a < b$ and $b > c$, then:

$$\left((\mathbf{ps}_{\langle \mathbf{x}_i - \mathbf{x}_j \leq w+v \rangle} = bad) * \mathbf{E} * \mathbf{H}\right)(\phi(t_m)) = 0 \ .$$

Claim 3.1 holds because $\mathbf{T}(\phi(t_m)) = \mathbf{T}(s_{m-1}, \sigma_m, s_m)$ must be 1 by condition 1 for the transition relation in the definition of the companion automaton.

Claim 3.2 holds by Proposition 4.1.

To show Claim 3.3, we apply Lemma 4.1 obtain:

$$\mathbf{E}(\phi(t_m)) = 1 \quad \text{only if} \quad x_{j,m-1} + \delta \geq -w \ , \tag{4.12}$$

$$\mathbf{H}(\phi(t_m)) = 1 \quad \text{only if} \quad x_{i,m-1} + \delta \leq v \ , \tag{4.13}$$

If $i = j$, then it must be that $a = c$, thus $w + v$ is the total weight of the loop and must be negative. It follows that $v < -w$, and therefore (by (4.12) and (4.13)) either $\mathbf{E}(\phi(t_m))$ or $\mathbf{H}(\phi(t_m))$ must be 0.

If $i \neq j$, then (by definition of *phi*):

$$(\mathbf{ps}_{\langle \mathbf{x}_i - \mathbf{x}_j \leq w+v \rangle} = bad)(\phi(t_m)) = 1 \quad \text{only if} \quad x_{i,m-1} - x_{j,m-1} > w + v \ . \tag{4.14}$$

Since enabling conditions in (4.12)–(4.14) cannot all be satisfied, it follows that at least one of the three conjuncts must evaluate to 0 (QED Claim 3.3). □

By Assumptions 4.1 and 4.2, the TLE algorithm can never return a false negative results. To show that it always terminates we will:

- show that the **modify** function indeed eliminates a run that has induced an over-constrained loop, implying that the abstraction of the system changes in every iteration,
- show that all the changes of the abstraction of the system comes from a finite set, implying that there can be only finitely many iterations.

First, we introduce the notion of a *peak node*. A node k is said to be *peak* if its immediate neighbors in the over-constrained loop that is input to **modify** (say i and j) satisfy $i < k$ and $j < k$. Note that we refer to as peak nodes only only those nodes that satisfy $i < k$ and $j < k$ at the beginning of **modify**, even though through the process of node elimination every node eventually satisfies $i < k$ and $j < k$. In other words, peak nodes are

those that *could* be selected in the *first* pass through the **while** loop of modify. Also, in the rest of the section we will assume that $s_0, s_1, \ldots, s_n$ is a failure report that has induced the over-constrained loop which is the input to modify. The basic mechanism for elimination is provided by the following lemma.

Lemma 4.2 *If k is a peak node and A_k (generated in step 1 of* modify*) is in the bad state, then the transition $s_{k-1} \rightarrow s_k$ is disabled.*

Proof. From step 3 of modify it follows that A_k being in the *bad* state implies that all transitions in $\mathcal{S}(\mathbf{E} * \mathbf{H})$ are disabled, and $(s_k, s_{k-1}) \in \mathcal{S}(\mathbf{E} * \mathbf{H})$ by step 3 in Figure 4.5. □

Therefore, to eliminate the failure trace it is enough that A_k for some peak node k is in the *bad* state when the rest of the system has gone through the sequence $s_0, s_1, \ldots, s_{k-1}$. The following two lemmas show that this is always the case. Loosely speaking, Lemma 4.3 asserts that there is always a *bad* state at the beginning of the loop, and Lemma 4.4 asserts that *bad* states are propagated forwards, and therefore must reach a peak node.

Lemma 4.3 *If A_k is the automaton generated in the last pass through the* **while** *loop of* modify*, then A_k must be in the bad state after the initial $\tilde{A}$ has gone through the sequence of states $s_0, s_1, \ldots, s_{k-1}$.*

Proof. The total weight of the loop remains constant (i.e. negative) throughout the modify function, because if two edges weighted w and v are removed, a new edge weighted $w + v$ is always added. Therefore, in the last pass through the **while** loop $w + v < 0$ must hold and $A_k = \langle \mathbf{y} - \mathbf{x} \leq w + v \rangle$ must move to the *bad* state when both $\mathbf{x}$ and $\mathbf{y}$ are reset at node $i = j$ (Figure 4.6b). Furthermore, since neither $\mathbf{x}$ nor $\mathbf{y}$ are reset on any transition between $s_{i-1} \rightarrow s_i$ and $s_{k-1} \rightarrow s_k$, A_k must remain in the *bad* state during that time. □

Lemma 4.4 *Let A_k be an automaton generated in step 1 of* modify *for some non-peak node k. If A_k is in the bad state after the initial $\tilde{A}$ has gone through the sequence of states $s_0, s_1, \ldots, s_{k-1}$, then there exists $m > k$ such that A_m is in the bad state after the initial $\tilde{A}$ has gone through the sequence of states $s_0 \ldots s_{k-1} \ldots s_{m-1}$.*

Proof. Since k is not a peak node, it can become a candidate for elimination only after a new edge has been created between k and some lesser-numbered node in steps 5 or 6 of modify. Such an edge must be labeled with some timer (say $\mathbf{x}$), and an expression

of the form $(\mathbf{ns}_{A_m} = \mathit{good}) * \mathbf{R_y} * \overline{\mathbf{R_x}}$ where $m > k$ is some node eliminated before k. The other edge incident with k may also be generated by elimination of some other node $m' > k$ (in which case it must labeled with some timer $\mathbf{x}'$ and an expression of the form $(\mathbf{ns}_{A_{m'}} = \mathit{good}) * \mathbf{R_{y'}} * \overline{\mathbf{R_{x'}}}$, or it might be an original edge of the loop in which case it must labeled with $\mathbf{x}'$ and some $\mathbf{E}$ which contains at least $s_{k-1} \rightarrow s_k$. Thus, when we eliminate node k, step 3 will require that none of the transitions in the modified $\tilde{A}$ satisfy an expression which is either of the form:

$$(\mathbf{ps}_{A_k} = \mathit{bad}) * (\mathbf{ns}_{A_m} = \mathit{good}) * \mathbf{R_y} * \overline{\mathbf{R_x}} * \mathbf{E} \ , \tag{4.15}$$

or of the form:

$$(\mathbf{ps}_{A_k} = \mathit{bad}) * (\mathbf{ns}_{A_m} = \mathit{good}) * \mathbf{R_y} * \overline{\mathbf{R_x}} * (\mathbf{ns}_{A_{m'}} = \mathit{good}) * \mathbf{R_{y'}} * \overline{\mathbf{R_{x'}}} \ . \tag{4.16}$$

Assume that the expression is of the form (4.15). Then, at $s_{k-1} \rightarrow s_k$, $(\mathbf{ps}_{A_k} = \mathit{bad})$ is satisfied by the assumption of the theorem, $\mathbf{E}$ is satisfied by the definition, and $\mathbf{R_y} * \overline{\mathbf{R_x}}$ is satisfied because the over-constrained loop indicates that $\mathbf{y}$ is reset on $s_{k-1} \rightarrow s_k$ and $\mathbf{x}$ is not. Thus, $\mathbf{ns}_{A_m} = \mathit{good}$ cannot be satisfied, and hence A_m must move to the bad state on $s_{k-1} \rightarrow s_k$.

Similarly, if the expression is of the form (4.16), at $s_{k-1} \rightarrow s_k$ $(\mathbf{ns}_{A_m} = \mathit{good}) * (\mathbf{ns}_{A_{m'}} = \mathit{good})$ cannot be satisfied, so either A_m or $A_{m'}$ must move to the bad state. Without loss of generality, assume it is A_m.

So far, we have shown that A_m must move to the bad state on $s_{k-1} \rightarrow s_k$. Since neither $\mathbf{x}$ nor $\mathbf{y}$ are reset between $s_{k-1} \rightarrow s_k$ and $s_{m-1} \rightarrow s_m$, A_m must remain in the *bad* state during that time, and the proof is complete. □

Theorem 4.4 *In the modified $\tilde{A}$ returned by the* modify *function there can be no run whose projection on S is $s_0, s_1, \ldots, s_n$.*

Proof. It follows by induction (using Lemma 4.3 as a base case, and Lemma 4.4 as an inductive step) that for every k in the over-constrained loop either:

1. there exist some peak node $i < k$ such that A_i is in the *bad* state after the initial $\tilde{A}$ has gone through $s_0, \ldots, s_{i-1}$, or
2. there exist some node $m \geq k$ such that A_m is in the *bad* state after the initial $\tilde{A}$ has gone through $s_0, \ldots, s_{m-1}$.

Applying the above to the last node in the loop, we conclude that there always exist some peak node k such that A_k is in the *bad* state after the initial $\tilde{A}$ has gone through $s_0, \ldots, s_{k-1}$, therefore (by Lemma 4.2) the transition $s_{k-1} \rightarrow s_k$ is disabled, and the run cannot be completed. □

To complete the proof of correctness, we show that only finitely many guard automata can be generated throughout the TLE algorithm. To simplify the proof, we make an additional requirement on analyze. We say that a portion of the loop from some node j to some node $i > j$ (denoted by $j \rightsquigarrow i$) is a *shortcut* if it consists of edges:

$$(j) \rightarrow (j+1) \rightarrow (j+2) \rightarrow \ldots \rightarrow (i-1) \rightarrow (i) \ ,$$

and every edge in that portion is weighted 0^-. We say that an over-constrained loop is *minimal* if for any two nodes $i > j$ in the loop, either the total weight of $i \rightsquigarrow j$ is positive, or $j \rightsquigarrow i$ is a shortcut.

For example, the loop $1 \rightarrow 2 \rightarrow 4 \rightarrow 3 \rightarrow 1$ in Figure 4.8 is over-constrained but not minimal, because of the loop $1 \rightarrow 2 \rightarrow 3 \rightarrow 1$ which is both over-constrained and minimal.

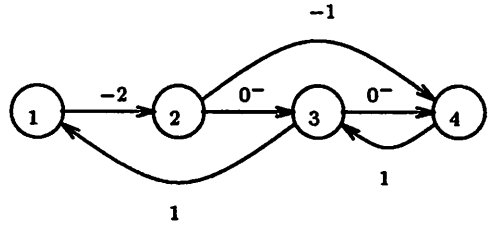


Figure 4.8: Minimality of a loop.

Assumption 4.3 *If the failure trace violates some timing constraints, then the* analyze *function returns a minimal over-constrained loop induced by it.*

Recall that we assume that there exists a timer x such that $x > 0$ is an enabling condition of every transition, and that it is reset on every transition (to acknowledge that time is strictly increasing). Such a timer induces an edge $(i) \rightarrow (i+1)$ weighted 0^- for every i. Thus, we can always make any over-constrained loop minimal, by repeating the following for all $i = 2, \ldots, n$ and all $j = 1, \ldots, i-1$:

- if the total weight of $i \rightsquigarrow j$ is not positive, then replace $j \rightsquigarrow i$ with a shortcut.

This assumption simplifies, but is not crucial to our algorithm. Without it, it would be possible that a failure report is eliminated even before all nodes are processed in function modify. To avoid extra checks, we shift a burden of finding a minimal loop to the analyze function.

Lemma 4.5 *Let c_{max} be some bound satisfying:*

1. *c_{max} is larger or equal than any other bound appearing in any timing inequality in the system,*
2. *$c_{max} = c_{max} + 0^+$ (i.e. c_{max} is of the form n^+).*

Then, the following is true for every edge $(i, j, w, \mathbf{x}, \mathbf{E})$ of an over-constrained loop throughout the modify *function:*

$$-c_{max} \leq w < 0 \qquad \text{if } i < j \ , \tag{4.17}$$

$$0 < w \leq c_{max} \qquad \text{if } i > j \text{ and } j \rightsquigarrow i \text{ is not a shortcut.} \tag{4.18}$$

$$-c_{max} \leq w \leq c_{max} \qquad \text{if } i > j \text{ and } j \rightsquigarrow i \text{ is a shortcut.} \tag{4.19}$$

Proof. By induction on the number of the passes through the **while** loop modify. Initially, (4.17)–(4.19) are satisfied by steps 4 and 5 of graph function in Figure 4.5. Now, assume that (4.17)–(4.19) hold before a new edge is generated in steps 5 or 6 of modify, i.e. assume that w and v satisfy either:

$$-c_{max} \leq w < 0 \quad \text{and} \quad 0 < v \leq c_{max} \ ,$$

or

$$w = 0^- \quad \text{and} \quad -c_{max} \leq v \leq c_{max} \text{ and } j \rightsquigarrow i \text{ is a shortcut.}$$

In either case, the weight of the new edge satisfies:

$$-c_{max} \leq w + v \leq c_{max} \ . \tag{4.20}$$

Furthermore, by the minimality Assumption 4.3, if $i > j$, then either $w + v > 0$, or $j \rightsquigarrow i$ is a shortcut, because $w + v$ is a total weight of $i \rightsquigarrow j$. Similarly, if $i < j$, then $w + v < 0$. To see this assume towards contradiction that $w + v \geq 0$. In that case $i \rightsquigarrow j$ is not a shortcut, because all shortcuts have total weight 0^-. Furthermore, the total weight of $j \rightsquigarrow i$ has to be negative (to make the total weight of the loop negative) contradicting the minimality Assumption 4.3. □

Corollary 4.1 *In step 1 of the* modify *function, the bound $w + v$ always satisfies (4.20).*

Theorem 4.5 *The TLE algorithm terminates with the correct answer in a finite number of steps.*

Proof. Theorem 4.3 states the the TLE algorithm cannot terminate with the false positive result, and by Assumptions 4.1 and 4.2 it cannot terminate with false negative results. By Theorem 4.4 each timing-violating failure report will be eliminated by modify. To eliminate a failure trace either at least one *new* guard automaton has to be generated in step 1, or some edges have to be eliminated from the current abstraction in step 3. But, by Corollary 4.1 only finitely many different guard automata are generated, so after the last one has been generated the number of edges in $\tilde{A}$ must be reduced in every iteration, therefore there can only be finitely many iterations. □

The proof of Theorem 4.5 suggests that in the worst case there might be as many iterations as edges in the composition of the initial abstraction with all the guard automata. In that case, the final abstraction of the system is equivalent to the region automaton. It follows that the iterative approach has no advantages in the worst case. We argue that determining the worst case number of iterations precisely has little practical value. With existing computing resources and in a reasonable time one can expect to perform at most several hundreds of iterations. That is definitely much less than the worst case even for very simple systems. Thus, a successful completion can almost never be guaranteed. Nevertheless, in many cases fewer than a hundred iterations is sufficient to verify the system. We believe that the TLE algorithm has two important properties that makes it suitable for verification in practice:

1. It is independent of the underlying untimed language containment algorithm. Any advances in the verification of untimed systems can immediately be incorporated in it. In particular, the TLE algorithm can be implemented on top of BDD-based techniques, which are receiving significant attention as a tool to manage complexity of finite-state systems.

2. The complexity of every iteration does not depend on the sizes of time constants. Thus, any change in time constants that does not change the untimed language of the system (including, but not limited to multiplying all time constants in the system by the same number) will not change the execution time of the algorithm. Of course,

if a change in time constants induces a change in the behavior, then the number of iterations can change.

4.4.6 Extension to $\mathbf{x} - \mathbf{y} \leq c$ constraints

To extend our approach to constraints of the form $\mathbf{x} - \mathbf{y} < c$, only the analyze function in the verify_timed algorithm needs to be changed. The rules for building a graph in the failure analysis phase are augmented such that for every $i = 1, \ldots, n$ and for every timing constraint $\mathbf{x} - \mathbf{y} \leq c$ in $\mathbf{C}(s_{i-1}, s_i)$:

1. Add an edge $(m, k, c, \mathbf{x}, \mathbf{0})$ to the graph, where k (respectively m) is the last node before i on which the timer $\mathbf{x}$ ($\mathbf{y}$) was reset, i.e.:

$$\begin{aligned} k &= \max\{j < i | j = 0 \text{ or } \mathbf{M}(s_{j-1}, s_j, \mathbf{x}) = \perp\} \ , \\ m &= \max\{j < i | j = 0 \text{ or } \mathbf{M}(s_{j-1}, s_j, \mathbf{y}) = \perp\} \ . \end{aligned}$$

2. If the edge $(m, k, c, \mathbf{x}, \mathbf{0})$ appears in the over-constrained loop, then before calling the modify function it must be replaced with edges $(i, k, c, \mathbf{x}, \mathbf{E})$ and $(m, i, 0, \mathbf{y}, \mathbf{E})$, where $\mathbf{E}$ is a characteristic function of some set of transitions enabled by $\mathbf{x} - \mathbf{y} \leq c$ that includes at least the transition (s_{i-1}, s_i), i.e.:

$$(s_{i-1}, s_i) \in \mathcal{S}(\mathbf{E}) \subseteq \{(s, q) | (\mathbf{x} - \mathbf{y} \leq c) \in \mathbf{C}(s, q)\} \ .$$

Using $(m, k, c, \mathbf{x}, \mathbf{0})$ in the failure analysis phase is justified because the requirement $\mathbf{x} - \mathbf{y} \leq c$ induces the inequality:

$$\lambda_m - \lambda_k \leq c \ .$$

If $k < m$, we must ensure in the failure elimination phase that transitions satisfying $\mathbf{E}$ are enabled only if $\mathbf{x} \leq c$ when $\mathbf{y}$ is reset at node m. Similarly, if $m < k$ we must ensure that transitions satisfying $\mathbf{E}$ are enabled only if $\mathbf{y} \geq -c$ when $\mathbf{x}$ is reset at node k. We achieve this be replacing $(m, k, c, \mathbf{x}, \mathbf{0})$ with $(i, k, c, \mathbf{x}, \mathbf{E})$ and $(m, i, 0, \mathbf{y}, \mathbf{E})$. We do this replacement in a manner that preserves the simplicity of the loop: if node i already appears in the over-constrained loop, then these two edges are connected to a distinct copy of node i. Thus, we always create a new peak node in the loop. Eliminating this node in modify indeed has the desired effect:

1. $\tilde{A}$ is composed with $A_i = \langle x - y \leq c \rangle$, and transitions satisfying $(\mathbf{ps}_{A_i} = bad) * \mathbf{E}$ are disabled,

2. edges of the loop incident to i are replaced with either

$$(m, k, c, \mathbf{x}, (\mathbf{ns}_{A_i} = good) * \overline{\mathbf{R_x}} * \mathbf{R}_y)$$

in case $k < m$, or in case $m < k$ with:

$$(m, k, c, \mathbf{y}, (\mathbf{ns}_{A_i} = good) * \mathbf{R_x} * \overline{\mathbf{R}_y}) \ .$$

One can check that proofs of all theorems in section 4.4.5 are valid even with this extension. Thus even with constraints of the form $\mathbf{x} - \mathbf{y} < c$, the verified_timed function will terminate with the correct result in finitely many steps.

4.4.7 Extension to infinite sequences

When infinite sequences are considered, the failure report and the corresponding graph are infinite. The key in overcoming this obstacle is the fact that if every prefix of some (infinite) run of a timed automaton can be consistently timed, then so can the whole run. In other words, if a run cannot be timed, then neither can some of its finite prefixes, and thus there exists a finite over-constrained loop that indicates the inconsistency. Thus, we can apply a standard algorithm to an ever increasing (finite) graph, adding one node at a time, in natural order. If an over-constrained loop exists, it will be found in a finite number of steps. Alur et al. [AIKY93] have showed that if a prefix of some bounded length (where that bound depends on the number of timing constraints) can be consistently timed, then so can the infinite run obtained by repeating a portion of the prefix infinitely often. Therefore, if an over-constrained loop does not exist, we can terminate the search after a bounded number of steps, and conclude that the language of a timed automaton is not empty.

When considering infinite sequences, the attention is usually restricted to those on which time diverges, i.e. only finitely many transitions happen in any finite interval of time. To eliminate sequences with convergent time, a fairness constraint must be added for every timer $\mathbf{x}$ requiring that any transitions with an enabling condition of the form $\mathbf{x} \leq c$ can be traversed infinitely often, only if some transition on which $\mathbf{x}$ is reset is traversed infinitely often.

4.5 Guided verification

In this section we describe how user guidance can increase the efficiency of the TLE algorithm. Usually, the designer knows which timing constraints are critical for a particular property. Not relaxing these constraints initially can dramatically reduce the number of iterations. Therefore, in the verification system HSIS [ABB+94, BBC+] which includes the implementation of the TLE algorithm we provide an option to the user of *"hinting"* which timing constraints *not* to ignore initially. This may increase the size of the initial abstraction, so one must be careful not to list too many constraints. Ideally, the user would list exactly those constraints that are necessary to prove the property at hand. In that case, the algorithm would terminate in a single iteration.

To specify a hint one needs to specify:

1. a timing constraint of the form $\mathbf{x} \geq -w$,
2. a characteristic function $\mathbf{E}$ of some set of transitions constrained by $\mathbf{x} \geq -w$,
3. a timing constraint of the form $\mathbf{y} \leq v$, and
4. a characteristic function $\mathbf{H}$ of some set of transitions constrained by $\mathbf{y} \leq v$.

Given such a hint, HSIS first checks its validity: it checks whether in the original description of the system all transitions characterized by $\mathbf{E}$ were indeed enabled only if $\mathbf{x} \geq -w$, and all transitions characterized by $\mathbf{H}$ were indeed enabled only if $\mathbf{y} \leq v$. If the hint is valid, HSIS composes the current abstraction with the guard automaton $A = \langle \mathbf{y} - \mathbf{x} \leq w + v \rangle$, and then eliminates all the transitions characterized by $(\mathbf{ps}_A = \mathit{bad}) * \mathbf{E} * \mathbf{H}$.

For example, consider the safety property of the railroad crossing example in Figure 4.1. It is satisfied because:

- the train will enter the crossing at least two time units after it approaches ($\mathbf{x} \geq 2$), and
- the gate will be *down* in less than two time units after the train approaches: one time unit for the controller to issue the command ($\mathbf{y} = 1$), and less than one unit for the gate to close after that ($\mathbf{z} < 1$),

This reasoning can be converted into three hints. The first hint can be denoted by:

$$(\mathbf{ps}_{TRAIN} = APP) * (\mathbf{ns}_{TRAIN} = IN) \implies \mathbf{x} \geq 2 \ ,$$
$$(\mathbf{ps}_{GATE} = U2D) \implies \mathbf{z} < 1 \ ,$$

indicating that $TRAIN$ can move from APP to IN only if $\mathbf{x} \geq 2$, and that $\mathbf{z} < 1$ must hold as long as $GATE$ is in the state $U2D$. The constraints $\mathbf{x} \geq 2$ and $\mathbf{z} < 1$ cannot be satisfied simultaneously if $\mathbf{x} - \mathbf{z} \leq 1$. Therefore, HSIS composes the initial abstraction of the system with $\langle \mathbf{z} - \mathbf{x} < -1 \rangle$, and then eliminates all transitions characterized by:

$$(\mathbf{ps}_{\langle \mathbf{z}-\mathbf{x}<-1 \rangle} = bad) * (\mathbf{ps}_{TRAIN} = APP) * (\mathbf{ns}_{TRAIN} = IN) * (\mathbf{ps}_{GATE} = U2D) \ .$$

Similarly, given a hint:

$$(\mathbf{ps}_{\langle \mathbf{z}-\mathbf{x}<-1 \rangle} = good) \implies \mathbf{x} > 1 \ ,$$
$$(\mathbf{ps}_{CONT} = R2L) \implies \mathbf{y} < 1 \ ,$$

HSIS composes the current abstraction with $\langle \mathbf{y} - \mathbf{x} < 0 \rangle$, and then eliminates all transitions characterized by:

$$(\mathbf{ps}_{\langle \mathbf{y}-\mathbf{x}<0 \rangle} = bad) * (\mathbf{ps}_{\langle \mathbf{z}-\mathbf{x}<-1 \rangle} = good) * (\mathbf{ps}_{CONT} = R2L) \ .$$

Finally, given a hint:

$$(\mathbf{ps}_{TRAIN} = APP) * (\mathbf{ns}_{TRAIN} = IN) \implies \mathbf{x} \geq 1 \ ,$$
$$(\mathbf{ps}_{CONT} = R2L) \implies \mathbf{y} < 1 \ ,$$

HSIS does not generate any new automata ($\langle \mathbf{y} - \mathbf{x} < 0 \rangle$ is already a component of the current abstraction), it only eliminates transitions characterized by:

$$(\mathbf{ps}_{\langle \mathbf{y}-\mathbf{x}<0} = bad) * (\mathbf{ps}_{TRAIN} = APP) * (\mathbf{ns}_{TRAIN} = IN) * (\mathbf{ps}_{CONT} = R2L) \ .$$

The result is an abstraction of the system in which none of the reachable states satisfy:

$$(\mathbf{ps}_{TRAIN} = IN) * ((\mathbf{ps}_{GATE} = UP) + (\mathbf{ps}_{GATE} = U2D)) \ .$$

It is easy to check that this implies that the safety property is satisfied.

Note that instead of the first implication in the final hint, we could have used a stronger implication:

$$(\mathbf{ps}_{TRAIN} = APP) * (\mathbf{ns}_{TRAIN} = IN) \implies \mathbf{x} \geq 2 \ ,$$

but this would require HSIS to generate a new guard automaton $\langle \mathbf{y} - \mathbf{x} < -1 \rangle$. Thus, by using the weaker implication we were able to guide HSIS to build an abstraction with fewer states.

A careful reader will notice that the modifications of the initial abstraction induced by hints are identical to the modifications done in the failure elimination phase of the TLE algorithm. Thus, a designer can use hints to eliminate all potential timing violations, and ensure that the verification terminates in one iteration. However, usually this is too big a burden. In our experience, it is most productive to generate hints interactively, i.e. a designer takes over the failure analysis phase of the TLE algorithm. Often, only a few iterations are needed in the interactive mode to generate a complete set of hints, while in the fully automatic mode even tens of iterations may not be enough to verify the system. Consequently, the experimental results in section 4.7 show that using hints enables verification of much larger systems. This suggests, that of potentially many timing violations in a failure trace, a user with an understanding of the intended behavior of the system can usually recognize violations essential to that behavior and give appropriate hints. On the other hand HSIS will report the first violation it finds, which is apparently not always the best choice.

4.6 Comparison of approaches

There are two distinct sources of complexity in verification of timed automata:

1. Systems usually consist of several interacting components, and the number of reachable states of the whole system is typically exponential in the number of components. Of course, this state explosion problems is not restricted to real-time systems, and has been a subject of a wide range of research. The state-of-the-art approach to this problem is to use BDD's to traverse the state space implicitly.

2. As shown in chapter 4, the number of states of the region automaton is proportional to the absolute value of every time constant. This is particularly problematic for *stiff*

real-time systems, i.e. systems consisting of components operating at widely different speeds. Most embedded systems can be classified as stiff real-time systems. For example, in an automotive system the engine revolution data may be collected several thousands times per second, while the cabin temperature may be checked only once every ten seconds. Even when time constants in the environment do not vary so widely, most embedded systems contain both (typically slow) software and (typically fast) hardware components.

Implementations of both minimization and TCTL model checking algorithms described in the previous sections have a common state space representation: all reachable untimed state components are enumerated explicitly, and difference bound matrices are used to represent timer values. Thus, they are expected to be fairly insensitive to stiffness[5], but quite limited in the number of untimed states. The approaches based on timed Petri nets hold some promise of alleviating this problem, but at the moment too few experimental results are available to make any judgements.

The successive approximation approach of section 4.3.1 is compatible with both explicit and implicit state enumeration techniques, thus it is possible to incorporate the best available techniques to deal with the untimed state explosion problem. However, since it requires building a full region automaton (for a subset of timers), it is very sensitive to stiffness. Even the optimization of time constants does not address the stiffness problem: if the ratios of time constants are large, dividing them with the greatest common denominator will leave at least some of them large.

Compared to the TLE algorithm, successive approximation (SA for short) represents a different trade-off between the number of iterations and the complexity of modification. Generating a full region automaton (even for a subset of constraints) as done in SA is much more complex and adds much more information about timing than the failure elimination procedure in Figure 4.7. Thus the TLE algorithm is expected to prove (or disprove) the property to be verified on a much smaller abstraction. On the other hand, since more information is used in every iteration, the SA algorithm could require fewer iterations. Unfortunately, no experimental results are available for SA to confirm these expectations.

An approach somewhat between the successive approximation and the TLE algorithm was suggested by Balarin and Sangiovanni-Vincentelli [BSV93]. This approach

[5]This expectation is supported by some experimental results [NSY92]

Table 4.1: Results for the timed automata algorithm

	no hints			with hints			
example	iter.	reach. st.	time	hints	iter.	reach. st.	time
csma	5	120	3.2s	1	1	48	1.2s
fddi-l	2	15	0.8s	1	1	15	0.6s
fddi-s	22	95	21.8s	6	1	29	0.8s
fis3	14	826	26.4s	6	1	830	0.8s
fis4	29	42,998	1,141.8s	12	1	44,545	6.7s
fis5	space out			20	1	$6 * 10^6$	168.4s
belt	space out			5	1	299	0.8s
fact	space out			58	3	$8 * 10^7$	84.7s
cross-s	4	16	0.6s	3	1	11	0.3s
cross-l	13	78	3.6s	11	1	17	0.4s

follows basically the failure elimination algorithm in Figure 4.7, but instead of explicitly eliminating inconsistent behavior in step 3, guard automata are coordinated with the rest of the system through auxiliary I/O variables, which indicate timer values approximately, up to the closest integer. Since the number of values of these variables obviously depends on the sizes of the constants in the system, this approach still suffers from the stiffness problem, as shown experimentally in next section. We will refer to this approach as preTLE, because the TLE algorithm was developed as a successor to this approach.

Finally, the TLE approach can be combined with implicit state enumeration techniques, and the complexity of every iteration does not depend on time constants. Thus, it addresses both untimed state explosion and stiffness problems.

4.7 Experimental results

We have applied TLE algorithm (with and without hints) to the following examples of timed automata:

- a model of the CSMA/CD protocol [NSY92] consisting of two stations and a channel; we have verified that a collision on the channel is always detected within a given time frame,

- a model of the FDDI token ring protocol [CDCT93] consisting of three stations; we have verified that every station transmits infinitely often (labeled by fddi-l in Table 4.1), and that there is an upper bound on time between two consecutive receptions

of a token by a station (labeled fddi-s),

- a model of the Fischer's mutual exclusion protocol consisting of three, four or five processes (labeled fis3, fis4, and fis5 in Table 4.1),
- a model of a seat-belt alarm controller [CGH+93] (labeled belt); we have verified that the alarm is never on for more than twenty time units,
- a model of an automated factory [PV94] consisting of a production line, a service station, two boxes, and two robots that move boxes between the service station and the line; we have verified that robots will always pick up a box before it reaches the end of the line,
- both safety (labeled cross-s) and liveness (labeled cross-l) properties of the railroad crossing example from Figure 4.1.

Experimental results are summarized in Table 4.1. All experiments were performed on a DEC MIPS 5000 workstation with 440Mb of physical memory.

The value of hints is obvious from Table 4.1. Without them, only smaller examples can be verified. In all but one example (fact), we have developed a complete set of hints, and with hints only one iteration was necessary. In these cases, the hints constitute the proof of correctness, and we are using our tool to check, rather than construct a proof. The automated factory examples illustrates how automatic verification can complement hints. After several tries, we were able to develop a set of hints that enforces most of the timing constraints necessary to verify the property. Automatic verification then filled-in the remaining gaps.

The number of reachable states reported in Table 4.1 refers to the abstraction of the system in the last iteration. One way of evaluating the effectiveness of our algorithm is to compare the sizes of automatically created abstraction (in no-hints column) with the sizes of hand generated ones (hints column) which we believe are close to optimal. In most of the cases the TLE algorithm was quite successful in abstracting the behavior of the system, except for the seat belt example, where a small abstraction could be generated by hand, but the TLE algorithm could not find an abstraction that can fit in 440Mb of memory. We traced this problem to to the component of that example that represents a very fast counter. The key to successful hand abstraction was the observation that the counter interacts with

Table 4.2: Sensitivity to stiffness

cycle	5	100	200	500	1,000	2,000
preTLE	1.8s	2.9s	4.5s	14s	45s	162s
TLE	0.6s	0.6s	0.6s	0.6s	0.6s	0.6s

Table 4.3: Comparison of results

example	TLE (no hints)	TLE (hints)	KRONOS	min. sur.	min. stable
csma	3.2s	1.2s	45s	NA	NA
fis3	26.4s	0.8s	NA	2s	8s
fis4	1,141.8s	6.7s	NA	45s	192s
cross-s	0.6s	0.3s	0.6s	1s	6s
cross-l	3.6s	0.4s	1.4s	NA	NA

the rest of the system only occasionally, and for short period of times. The TLE algorithm, however, was often finding the counter involved in timing violations, even though these violations were not significant, because at these points there was no interaction with the rest of the system. This suggests that more sophisticated failure analysis schemes need to combine timing constraints and communication patterns between system components.

To test sensitivity to stiffness we have compared the preTLE and TLE approaches on the safety property of the railroad crossing (cross-s) example in Figure 4.1 with different values of the train cycle time (instead of the original value 5). We have chosen this example because the untimed behavior of the system is independent of this time constant. The results are shown in Table 4.2. The time in the preTLE approach grows more than linearly with the cycle time, while with the TLE approach the time stays constant. These experiments were performed without hints.

In Table 4.3 we compare the results for TLE algorithm with the results from the tool KRONOS [Yov92], and the results obtained by minimization approaches (using both surjective and stable partitions) presented by Alur et al. [ACD+92].

Compared to other available results TLE algorithm is comparable without hints and always better with hints. That is not surprising because the hints rely on the knowledge (and the effort) of the user, while other approaches are completely automatic. The user's effort is hard to quantify, but in our experience *"three hints per hour"* seems to be a good rule of thumb. It seems that the TLE algorithm (being fully implicit) has a capacity to deal with larger untimed state spaces (e.g. fis5 and fact), but to use that capacity efficiently, user intervention is often crucial. Without it, the extra capacity is often lost on enforcing

timing constraints that are irrelevant to the property to be verified. Hints enforce the timing constraints using the same mechanism as the TLE algorithm, but they are driven by a user instead of an automatic failure analysis. Therefore, developing more sophisticated failure reporting and analysis schemes should narrow the performance gap between automatic and guided version of the TLE algorithm. Nevertheless, we believe that the TLE algorithm is uniquely suited for verification of embedded real-time systems because it addresses both sources of complexity of such systems:

- because it is fully implicit it can deal with large untimed sate spaces,
- because the complexity of every iteration is independent of time constants, it can be used effectively for stiff systems.

In practice, designs are often annotated with some additional information explaining the intended behavior. In principle, this information could be used to generate hints. The problem is that this information is typically in the form of informal notes or timing diagrams. Developing formalisms for this information and using it for the automatic generation of hints represents an interesting open problem.

Chapter 5

Real-Time Operating System

To compare and test formal verification tools there is a need for a well-documented set of benchmarks. The lack of such examples is particularly evident for real-time systems where only a handful of formal models have been published and verified. In the following two chapters, we present several models of a real-time operating system for an automatically controlled vehicle. We believe that these models are good candidates to be included in such a set, because:

- it comes from a real-life system,
- it is easily scalable in the size of the state space, in the level of details that the model provides, and in the complexity of the properties to be verified,
- time constants vary by more than an order of magnitude, thus it is a stiff systems,
- it is representative of many verification problems for embedded systems: typically embedded systems have to react in time to external stimuli and that is captured by formal models of PATHO.

In this chapter we use timed automata as a model, and describe how to verify that the operating system using the TLE algorithm in automatic and interactive modes. In the next chapter, we present a model that uses timer decrements. An extension of the TLE algorithm that could be used to verify such systems is also presented.

5.1 PATHO operating system

PATHO real-time operating system [Pet93] is an example of an embedded system. It is intended for the automatic control of a vehicle [Var93]. To preserve vehicle safety, PATHO must manage a collection of subroutines, called tasks, such that they meet strict real-time requirements imposed by the environment. There are three different types of tasks: background tasks, control tasks, and event tasks. Background tasks have low priority, and they need to run only when the CPU is free. Control tasks need to run at a fixed period, e.g. a clock or a control calculation. These tasks have very high priorities and hard time limits. The third type are event tasks. These tasks respond to hardware interrupts generated by the environment. These interrupts can come from a radio communications channel, saying that it has a packet of information, from a sensor on the engine of the car, saying that it has some measurement information, or other devices. Whatever the source, the interrupt needs to be serviced in a timely manner. Interrupts arrive randomly, but for every task there is a lower bound on elapsed time between two interrupts.

When the PATHO system is initialized, all tasks are created from subroutines written by the user. These tasks, depending on their type, are then placed in various queues. The control tasks are placed in a waiting queue and their individual clocks are set to their time-out values. While running, PATHO decrements these clocks continuously until they expire. This indicates that those tasks are ready to run. At that time the ready tasks are placed on the ready-to-run queue and their clocks are reset to their time-out values. Once they have finished they are placed back on the waiting queue. Event tasks also start off on a waiting queue. Each event task is associated with a particular interrupt line. When that interrupt occurs, the event task is taken off of the waiting queue and put on the ready-to-run queue by a special routine called an Interrupt Service Routine (ISR). The tasks on the ready-to-run queue are serviced according to pre-defined priorities, with no preemption. After the task finishes, it is put back on the waiting queue. A simple way to view the process of running an event task is that the ISR simply tells the PATHO scheduler that the task needs to run and the scheduler takes care of running it. Finally, the background tasks are placed in the background queue. PATHO will try to run one of these tasks whenever no other tasks are ready to run.

Task Name	Function	Priority	Period	Run Time
Long. Cntrl	Control longitudinal position	0	$50ms$	$200\mu s$
Lat. Cntrl	Control lateral position	1	$50ms$	$10\mu s$
Opt. Pos.	Optical triangulation of position	4	$50ms$	$1\mu s$
Safety	Periodic safety check of car	7	$1s$	$200\mu s$
Calibration	Periodic calibration of car	9	$60s$	$200\mu s$

Table 5.1: Control tasks.

Task Name	Function	Priority	Lower Bound	Run Time
Radio	Platoon communications	2	$20ms$	$10\mu s$
InfraRed	Car to car communications	3	$20ms$	$10\mu s$
Radar	Radar position measurement	5	$20ms$	$200\mu s$
Sonar	Sonar position measurement	6	$6ms$	$15\mu s$
User	User interface	8	$100ms$	$20\mu s$

Table 5.2: Hardware interrupt tasks.

5.1.1 A typical PATHO system

Consider an example where PATHO controls every aspect of the vehicle: the steering, the speed, the spacing between adjacent cars, etc. As a result there are many pieces of hardware that are needed and many different control tasks. Control tasks are listed in Table 5.1, while event tasks are listed in Table 5.2. In both cases functions that they perform, their relative priority (with 0 being the highest priority), and the maximum time they can take are all shown.

Reason to wait	Wait time
Jump to an ISR and back	$2\mu s$
Switch from one task to another	$100\mu s$

Table 5.3: Wait times in PATHO.

There are two different delay times associated with the overhead of PATHO. One is the task switching time, that takes about $100\mu s$. Another delay that should be modeled is the time that it takes to jump to an ISR and back. As shown in Table 5.3, this delay is typically much smaller than either task switching time or tasks running times.

5.2 Modeling PATHO with timed automata

As usual, the models of PATHO capture only some of its features. Other features were left out either because they are not expressible in the selected formalism or because they require too complex a model. In all models presented in this chapter, the following features of PATHO were abstracted:

Message passing PATHO allows the various tasks to pass messages to each other using system resources called pipes. Each time there is a system call to send or receive a message there is a possibility that a task will get blocked until the transaction is complete. We model this by adding some estimated overhead to tasks running times.

Mutual exclusion of resources It is possible for a few different tasks to share the same resource (e.g. a printer, memory, etc.). Thus a task might be blocked for a while if another task uses a resource. Similarly to message passing, we model mutual exclusion by increasing running times.

Extensive user interface code The form of the user interface for PATHO has not been decided upon. It could be running on the same machine as the other control algorithms or it could be running on a separate machine over a network. The real-time characteristics of these two different interfaces are unknown. Therefore a generic interrupt driven task is substituted in its place.

Data logging Data logging is simply the process of storing the state variables either in memory, to a disk drive, or to another machine across the network. This aspect of the PATH project has not been implemented yet and therefore, like the user interface, the real-time characteristics are unknown.

Background tasks Background tasks were left out because they have lower priority and do not influence the behavior of control and event task.

Message passing and mutual exclusion can be modeled precisely within a framework of *integrator systems* [ACHH93], where it is possible to stop, for a while, a timer that measures the run time of a task. Unfortunately, verification of integrator systems is undecidable in general.

In this section we present two models of PATHO based on timed automata. The first is the most natural formalization of the informal description in section 5.1, but not very efficient for verification. The second one is an equivalent model optimized for verification.

The main features of PATHO represented in our models are:

- external interrupts and internal time-outs,
- control and event tasks,
- queue of tasks waiting to run and priority scheduling,
- PATHO overhead time.

In all models control and event tasks are treated in the same way, except that event tasks are driven by external hardware interrupts while the control tasks are scheduled by the internal control task timers. Also, the task switching time is modeled by increasing the task running times.

In the rest of this section we present models of a sample PATHO system with n tasks labeled: $0, 1, \ldots, n-1$. We assume that labels correspond to the priorities of the tasks, with 0 being the highest priority. We use r_i to denote the running time of task i. If i is a control task, it has to become ready to run periodically, with period p_i. If i is an event task, we denote by p_i a lower bound on time between the occurrence of two successive interrupts of task i.

The property we want to prove is that all tasks are executed in a timely manner. More precisely, we want to show that:

- if an interrupt occurs, then the corresponding event tasks will be executed before another interrupt of the same kind occurs,
- when control task i becomes ready to run it will be executed before p_i time units expire.

This property is naturally expressed as a conjunction of n sub-properties, one for each task. We use $Prop_i$ to denote these sub-properties. It should be intuitively clear that although their specification is the same, $Prop_i$'s are not all equally hard to prove (at least for human reasoning). The easiest one is $Prop_0$ corresponding to the task with highest priority. When interrupt 0 occurs the corresponding task will execute as soon as the currently running task

(if there is any) completes. So, $Prop_0$ is satisfied if $r_0 + r_i < p_0$ for all $i > 0$. Lower priority tasks can be delayed longer, and more complicated reasoning is necessary to prove that the corresponding properties hold.

5.2.1 A simple model

In our first attempt to model PATHO interrupts, time-outs, tasks and scheduler are all modeled separately. In this case, the I/O variable is a vector :

$$(\mathbf{s}, \mathbf{e}_0, \mathbf{t}_0, \mathbf{e}_1, \mathbf{t}_1, \ldots, \mathbf{e}_{n-1}, \mathbf{t}_{n-1}) \ ,$$

where:

- $\mathbf{e}_i$ taking values in $\{0, 1\}$ is associated with an interrupt (if i is an event task) or a time-out (if i is a control task). If i is an event task, then $\mathbf{e}_i = 1$ indicates the occurrence of the interrupt. Similarly, if i is a control task, $\mathbf{e}_i = 1$ indicates that p_i time units have elapsed.

- $\mathbf{t}_i$ taking values in $\{idle, pend, run, done\}$ is associated with the task i. If the task is in the waiting queue, then $\mathbf{t}_i = idle$; if it is in the ready-to-run queue, then $\mathbf{t}_i = pend$; if it is running, then $\mathbf{t}_i = run$; finally, when it is transferred from the ready-to-run to the waiting queue, then $\mathbf{t}_i = done$.

- $\mathbf{s}$ taking value in $\{0, 1, \ldots, n-1, n\}$ is associated with the scheduler. If $\mathbf{s} = i$, then task i can start running; if $\mathbf{s} = n$ either some task is running, or there are no tasks ready to run.

The set of timers is $V = \{\mathbf{x}_0, \mathbf{y}_0, \ldots, \mathbf{x}_{n-1}, \mathbf{y}_{n-1}\}$, where the timer $\mathbf{y}_i$ measures the execution time of task i, and the timer $\mathbf{x}_i$ measures the elapsed time between two interrupts or time-outs corresponding to task i.

A simple model of a periodic time-out is shown in Figure 5.1. It has two states, any one of which can be initial. Exactly every p_i time units there will be a change of states, and only at that time, $\mathbf{e}_i = 1$. At all other times (i.e. in self-loops) $\mathbf{e}_i = 0$. The model of an interrupt is similar; one only needs to remove timing conditions from self-loops and to replace timing conditions $\mathbf{x}_i = p_i$ with $\mathbf{x}_i \geq p_i$.

A simple model of a task is also shown in Figure 5.1. The task starts in the *idle* state and remain there until $\mathbf{e}_i = 1$. If $\mathbf{s} = i$ when that happens, it moves to the *run* state.

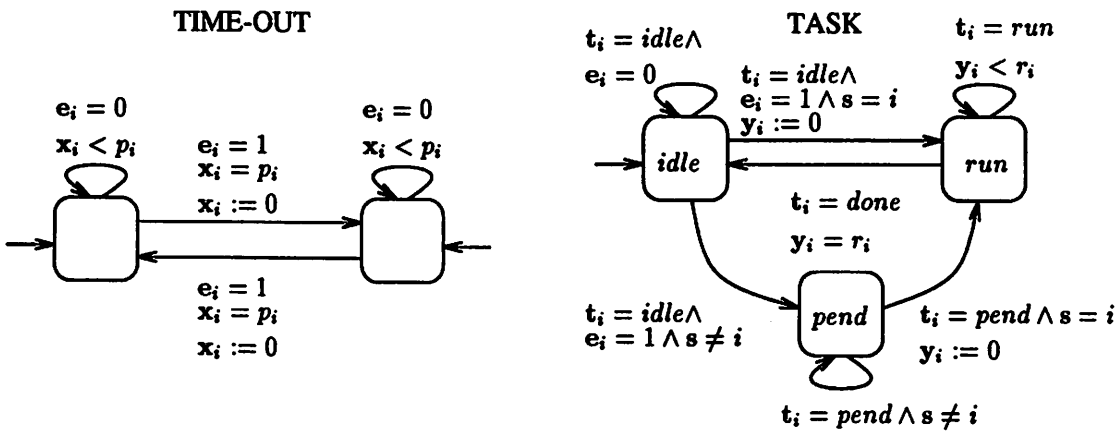


Figure 5.1: Simple models of a time-out and a task.

Otherwise, it moves to the *pend* state, and remains there until $s = i$. Once in the *run* state, the task will remain there for exactly r_i time units.

The scheduler is modeled as a one-state machine that computes the value of variable s as follows:

$$s = \begin{cases} n & \text{if } t_i = run \text{ for some } i \\ \min(\{n\} \cup \{i | t_i = pend \vee e_i = 1\}) & \text{otherwise} \end{cases}.$$

In this case the property $Prop_i$ can be described by:

It is always the case that $(e_i = 1) \Longrightarrow (t_i = idle)$.

It is straightforward to define a two-state automaton which is initially in the "*good*" state and remains there as long as either $e_i = 0$ or $t_i = idle$. If that is not the case, it moves to the "*bad*" state and remains there forever. The verification problem is then to show that the *bad* state is not reachable.

5.2.2 An optimized model

The model describe in the previous section can be optimized to reduce the number of states and I/O values. The optimized model is shown[1] in Figure 5.2. In this case, the

[1]Again, Figure 5.2 is valid only for control tasks. To model event task, remove the timing constraints $x_i < p_i$, and replace constraints $x_i = p_i$ with $x_i \geq p_i$.

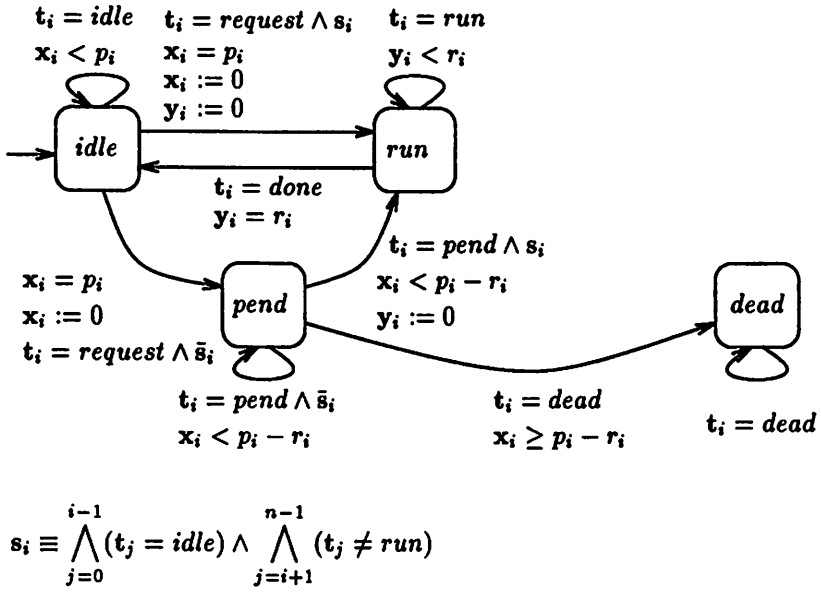


Figure 5.2: An optimized model of a task.

I/O variable is a vector $(t_0, \ldots, t_{n-1})$, where each t_i takes values in:

$$\{idle, request, pend, run, done, dead\} \ .$$

Values *idle*, *pend*, *run* and *done* have the same intuitive meaning as before. The value *request* corresponds to the occurrence of a time-out (or an interrupt, in case of event tasks). The value *dead* corresponds to the case when a task is not finished on time.

The main differences compared to the simple model are:

1. Models of a task and the corresponding time-out are merged. To model a time-out the process in Figure 5.2 issues a *request* if it is in state *idle* and the value of the timer x_i is p_i. If the system behaves correctly this will be repeated periodically, every p_i time units. However, if the process cannot make it on time back to the *idle* state, it moves to the *dead* state and remains there forever. The automaton moves from the *pend* state to the *dead* state whenever there is not enough time for the task to finish before x_i reaches p_i.

   Another approach is to monitor $x_i = p_i$ in the *pend* and the *run* state, and move to the *dead* state if it becomes true before the automaton moves to the *idle* state. However, the approach we have taken proved to be more efficient for verification.

2. There is no separate module for the scheduler. Instead, its functionality is distributed to models of tasks. The task will move to the run state only if expression $\mathbf{s}_i$, defined in Figure 5.2, is satisfied. It is easy to check that that condition exactly expresses the PATHO scheduling policy.

To verify property $Prop_i$ one only needs to check whether the *dead* state is reachable in the i-th process. Although the model in Figures 5.1 and 5.2 do not have the same languages, they are equivalent in a sense that properties $Prop_i$ are all satisfied in 5.1 if and only if they are satisfied in 5.2.

5.3 Experimental results and discussion

In this section we present experimental results obtained using HSIS. All results are obtained for the model consisting of n event tasks like those described in section 5.2.2, with $p_i = 20s$ and $r_i = 1s$ for all $i = 1, \ldots, n-1$. These numbers were obtained from Tables 5.1 and 5.2 by taking the smallest period or lower bound ($6ms$) and the largest run time ($200\mu s + 100\mu s$ for task switching) and scaling. We have experimented with two values of n: 4 and 10.

Under these assumptions, $Prop_i$ (stating that the *dead* state is not reachable in process i) is satisfied, because the execution of the i-th task can be delayed at most $i+1$ seconds (up to one second if some task is running when i makes a request, and up to i seconds for higher priority tasks that could be pending). That is less than $p_i - r_i = 19s$ even for $i = n - 1 = 9$ (the task with the lowest priority).

This reasoning can be converted into hints. One group of hints ensures that higher priority tasks do not occur too often. More precisely, the hint:

$$(\mathbf{ps}_j = idle) * (\mathbf{ns}_j \neq idle) \implies \mathbf{x}_j \geq 20$$
$$(\mathbf{ps}_i = pend) * (\mathbf{ns}_i = pend) \implies \mathbf{x}_i \leq 19$$

disallows two consecutive service requests for task j to occur while the task i is continuously pending. If we generate one such hint for every task j with a higher priority than i, we will ensure that task i can be delayed by task j at most twice: once if j is already pending when i requests a service, and j can require service at most one more time while i is pending.

The second group of hints ensures that task i does not move to the *dead* state too soon. Even though this condition can be expressed using standard hints introduced in

section 4.5, for reasons of efficiency we have developed a new type of hint for this purpose. An example of this type of hint is:

$$\begin{aligned}(\mathbf{ps}_3 = \mathit{pend}) * (\mathbf{ns}_3 = \mathit{dead}) &\implies \mathbf{x}_3 > 19 \\ (\mathbf{ps}_0 = \mathit{run}) &\implies \mathbf{y}_0 \leq 1 \\ (\mathbf{ps}_1 = \mathit{run}) &\implies \mathbf{y}_1 \leq 1 \\ (\mathbf{ps}_2 = \mathit{run}) &\implies \mathbf{y}_2 \leq 1 \ .\end{aligned}$$

The first line must satisfy the same requirements as the lower bound implication of a standard hint. There can be more than one upper bound implication, which must satisfy an additional constraint: the left-hand side must be a predicate on present state variables only. Again, an implication is valid if all the transitions satisfying the predicate on the left-hand side are conditioned with the lower bound inequality on the right-hand side.

Given the hint above, HSIS will compose the initial abstraction of the system with the automaton H shown in Figure 5.3. Basically, H measures time by counting the occurrences of:

$$C_3 \equiv (\mathbf{R}_{\mathbf{y}_0} + \mathbf{R}_{\mathbf{y}_1} + \mathbf{R}_{\mathbf{y}_2}) \ .$$

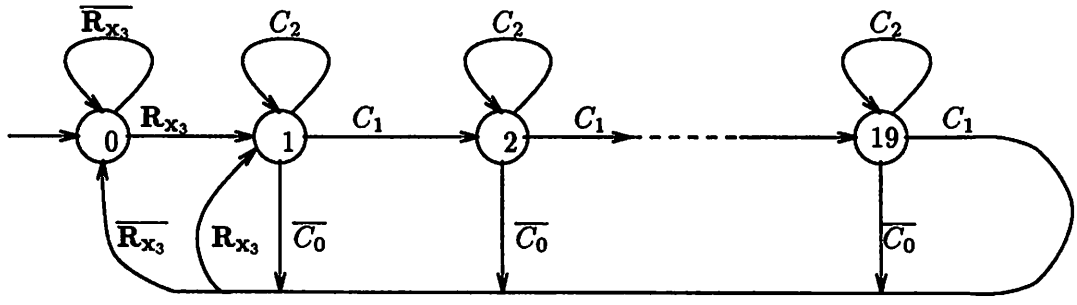
Since $\mathbf{R}_{\mathbf{y}_i}$ is satisfied only when the i-th task starts running, and since no task can run for more than one time unit, it follows that as long as:

$$C_0 \equiv (\mathbf{ps}_0 = \mathit{run}) + (\mathbf{ps}_1 = \mathit{run}) + (\mathbf{ps}_2 = \mathit{run})$$

holds, no more than one time unit can elapse between two occurrences of C_3. Therefore, H will not allow the transition $(\mathbf{ps}_3 = \mathit{pend}) * (\mathbf{ns}_3 = \mathit{dead})$ to occur unless C_3 occurs at least nineteen times since the reset of $\mathbf{x}_3$. Of course, if at any time C_0 stops holding, we loose track of time, and must allow $(\mathbf{ps}_3 = \mathit{pend}) * (\mathbf{ns}_3 = \mathit{dead})$ to occur.

In general, this type of hint is represented by:

$$\begin{aligned}\mathbf{E} &\implies \mathbf{x}_0 \geq c_0 \\ (\mathbf{ps}_1 = \mathit{val}_1) &\implies \mathbf{x}_1 \leq c_1 \\ (\mathbf{ps}_2 = \mathit{val}_2) &\implies \mathbf{x}_2 \leq c_2 \\ &\vdots \\ (\mathbf{ps}_k = \mathit{val}_k) &\implies \mathbf{x}_k \leq c_k \ ,\end{aligned}$$



$$C_0 \equiv (\mathbf{ps}_0 = run) + (\mathbf{ps}_1 = run) + (\mathbf{ps}_2 = run)$$

$$C_1 \equiv C_0 * (\mathbf{R}_{\mathbf{y}_0} + \mathbf{R}_{\mathbf{y}_1} + \mathbf{R}_{\mathbf{y}_2}) * \overline{(\mathbf{ps}_3 = pend) * (\mathbf{ns}_3 = dead)}$$

$$C_2 \equiv C_0 * \overline{(\mathbf{R}_{\mathbf{y}_0} + \mathbf{R}_{\mathbf{y}_1} + \mathbf{R}_{\mathbf{y}_2})} * \overline{(\mathbf{ps}_3 = pend) * (\mathbf{ns}_3 = dead)}$$

Figure 5.3: An automaton implied by a hint.

where $\mathbf{E}$ is a characteristic function of some set of transitions, $\mathbf{ps}_1, \ldots, \mathbf{ps}_k$ are (not necessarily distinct) present state variables, $\mathbf{x}_0, \ldots, \mathbf{x}_k$ are (not necessarily distinct) timers, and $c_0, \ldots, c_k$ are bounds. Given such a hint, HSIS composes the current abstraction of the system with an automaton which we call H. The state space of H is $\{0, 1, \ldots, n\}$ where n is the largest integer satisfying $n * c_i < c_0$ for all $i = 1, \ldots, k$. All states of H are final, and 0 is a unique initial state. Let the *if-then-else* operator $ite(a, b, c)$ be an abbreviation for $a * b + \overline{a} * c$. Then, the transition relation of H is given by:

$$(\mathbf{ps}_H = 0) * ite(\ \mathbf{R}_{\mathbf{x}_0},\ \mathbf{ns}_H = 1,\ \mathbf{ns}_H = 0) \ + \ \sum_{i=0}^{n} (\mathbf{ps}_H = i) * \mathbf{N}_i \ ,$$

where:

$$\begin{aligned} \mathbf{N}_i \ \equiv \ & ite(\ \textstyle\sum_{j=1}^{k} (\mathbf{ps}_j = val_j), \\ & \quad \overline{\mathbf{E}} * ite(\ \textstyle\sum_{j=1}^{k} \mathbf{R}_{\mathbf{x}_j},\ \mathbf{ns}_H = (i+1) \bmod n,\ \mathbf{ns}_H = i), \\ & \quad ite(\mathbf{R}_{\mathbf{x}_0},\ \mathbf{ns}_H = 1,\ \mathbf{ns}_H = 0) \\ &) \ . \end{aligned}$$

Experimental results are summarized in Table 5.4. All times are in CPU seconds on DEC MIPS 5000 workstation with 440Mb of memory. The number of states reported in the table is the size of the abstraction in the last iteration. Where indicated, to verify $Prop_i$ we have used one hint of the form:

$$(\mathbf{ps}_i = pend) * (\mathbf{ns}_i = dead) \implies \mathbf{x}_i > 19$$

property	$n = 4$ (with hints)		$n = 4$ (no hints)		$n = 10$ (with hints)	
	time	states	time	states	time	states
$Prop_0$	$0.4s$	216	0.9	654	$1.1s$	$3 * 10^5$
$Prop_1$	$0.4s$	242	2.7	$2 * 10^3$	$1.3s$	$3 * 10^5$
$Prop_2$	$0.5s$	564	1379	$8 * 10^6$	$2.4s$	$6 * 10^5$
$Prop_3$	$1.1s$	1292	space out		$5.3s$	10^6
$Prop_4$	not defined				$16.5s$	$3 * 10^6$
$Prop_5$					$65.3s$	$6 * 10^6$
$Prop_6$					$281s$	10^7
$Prop_7$					$2,413s$	$6 * 10^7$
$Prop_8$					$15,252s$	10^8
$Prop_9$					space out	

Table 5.4: Experimental results.

$$
\begin{aligned}
(\mathbf{ps}_0 = run) &\implies \mathbf{y}_0 \leq 1 \\
&\vdots \\
(\mathbf{ps}_{i-1} = run) &\implies \mathbf{y}_{i-1} \leq 1 \\
(\mathbf{ps}_{i+1} = run) &\implies \mathbf{y}_{i+1} \leq 1 \\
&\vdots \\
(\mathbf{ps}_n = run) &\implies \mathbf{y}_n \leq 1
\end{aligned}
$$

(where n is 4 or 10), and for every $j < i$ we have added a hint:

$$
\begin{aligned}
(\mathbf{ps}_j = idle) * (\mathbf{ns}_j \neq idle) &\implies \mathbf{x}_j \geq 20 \\
(\mathbf{ps}_i = pend) * (\mathbf{ns}_i = pend) &\implies \mathbf{x}_i \leq 19 \ .
\end{aligned}
$$

Given these $i+1$ hints, HSIS verifies $Prop_i$ in one iteration. Without hints, tens of iterations were necessary. For $n = 10$ we report only results with hints, because without hints only $Prop_0$ can be verified. Other properties violate the memory limit.

In our experience, finding right hints has been an iterative process. Initial hints were not sufficient, and better hints were discovered after analyzing the failure report. Thus, we were following the same basic steps as the automatic procedure, but apparently our manual failure analysis was much more effective than the automatic one.

The second observation is that even with hints, we have to deal with the state explosion problem. Increasing the size of the system from four to ten tasks, increases the

number of reachable states by three orders of magnitude. Fortunately, the run times do not increase proportionally. But, verifying property $Prop_i$ requires approximately twice as many states and four times as much time as verifying $Prop_{i-1}$. So, there still exists a fundamental problem of an exponential increase. There are many theoretical results (e.g. [Alu91]) suggesting that this complexity is inherent. Nevertheless recent advances have made possible the verification of examples that were previously intractable. We can only hope that this trend will continue.

Chapter 6

Verification of TAD's

In this chapter, we first present of model of PATHO that uses timer decrements, and then use this example to introduce an extension of the TLE algorithm to arbitrary TAD's. The basic structure of the TLE algorithm shown in Figure 4.4 remains the same even in this case. Also, since at all times the abstraction of the system $\tilde{A}$ is just an ordinary (not timed) automaton, the verify function remains unchanged. Thus, we focus on extended versions of analyze and modify functions which are capable of dealing with arbitrary TAD's, and discuss the effect of these extensions to the proof of correctness. It is important to notice that these are indeed only generalizations of the TLE algorithm, i.e. for the special case of timed automata the algorithm presented in this chapter reduces exactly to TLE.

6.1 A model of PATHO with decrements

Both models PATHO presented in chapter 5 ignored the time taken by the ISR. In this section we present a model of PATHO that takes this effect into account. Assume that ISR requires Δ time units. There are two ways to model the effect of ISR to the execution of a task [MV94]:

1. whenever ISR is executed, the value of a timer measuring running time of task (i.e. $\mathbf{y}_i$ in Figure 5.2) is "frozen" for Δ time units,

2. whenever ISR is executed, the value of $\mathbf{y}_i$ is decremented by Δ, but $\mathbf{y}_i$ continues to increase.

In both approaches it takes the same time for $\mathbf{y}_i$ to reach r_i.

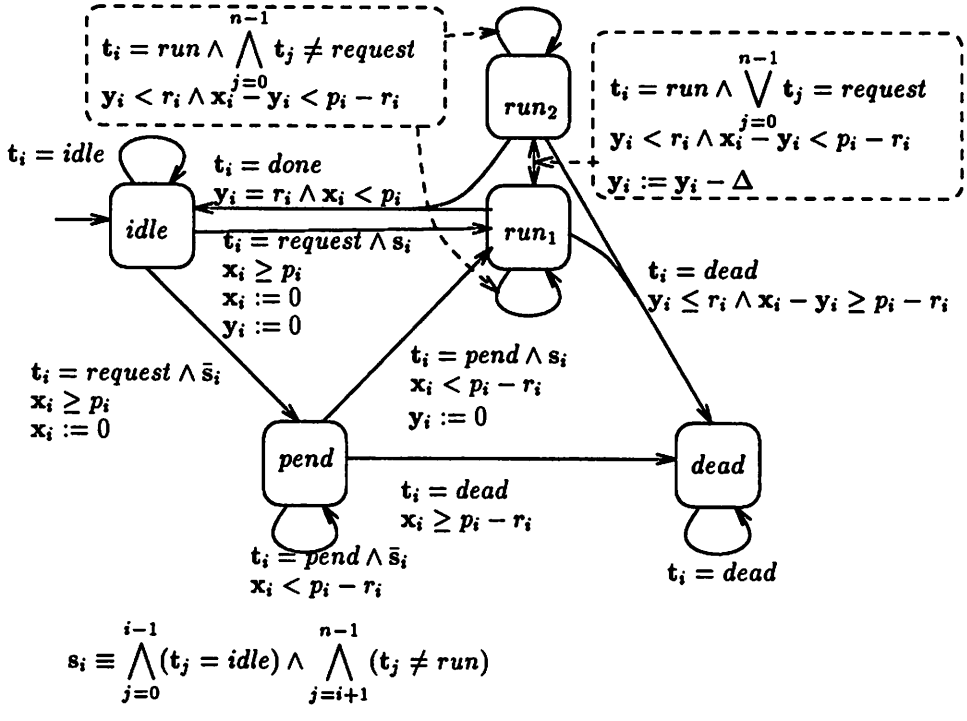


Figure 6.1: A model of PATHO that includes ISR.

The model of PATHO that includes the effect of ISR is shown in Figure 6.1. It shares the I/O variables and the basic structure of the optimized model in Figure 5.2. The difference is that the *run* state has been split into run_1 and run_2 states. Whenever ISR is executed (i.e. whenever $\mathbf{t}_i = request$ for some i) the process switches run states and decrements the timer $\mathbf{y}_i$ by Δ. As long as the process is in one of the run states, $r_i - \mathbf{y}_i$ represents the minimum time before the task completes, and $p_i - \mathbf{x}_i$ represent the time after which the next interrupt can occur. Whenever the former becomes larger than the latter, $Prop_i$ can be violated and the process moves to the *dead* state. The verification problem is again to check whether some process can reach the *dead* state.

6.2 Failure analysis

Consider the failure report:

$$\underbrace{idle}_{s_0} \longrightarrow \underbrace{run_1}_{s_1} \longrightarrow \underbrace{run_2}_{s_2} \longrightarrow \underbrace{dead}_{s_3} , \tag{6.1}$$

of the automaton in Figure 6.1. Since the timer $\mathbf{y}_i$ is reset on the transition $s_0 \rightarrow s_1$ and decremented on $s_1 \rightarrow s_2$, the value of $\mathbf{y}_i$ on $s_2 \rightarrow s_3$ is $\lambda_3 - \lambda_1 - \Delta$. Therefore, the constraint $(\mathbf{y}_i \leq r_i) \in \mathbf{C}(s_2, s_3)$ induces the inequality:

$$\lambda_3 - \lambda_1 \leq r_i + \Delta \ .$$

In the constraint graph this inequality needs to be represented by an edge from 3 to 1 weighted $r_i + \Delta$. The **graph** function in Figure 4.5 would generate an edge from 3 to 1 weighted r_i instead, which is wrong. The correct constraint graph for a failure report $s_0, \ldots, s_n$ of an arbitrary TAD can be built by applying one of the following rules for every $i = 1, \ldots, n$ and every $\psi \in \mathbf{C}(s_{i-1}, s_i)$:

Rule 1: This rule apply if ψ is of the form $\mathbf{x} \leq c$. Let k be the index of the last transition before the i-th one on which $\mathbf{x}$ was reset, i.e.:

$$k = \max\{j < i | j = 0 \text{ or } \mathbf{M}(s_{j-1}, s_j, \mathbf{x}) = \perp\} \ . \tag{6.2}$$

In this case, we add to the graph an edge from i to k weighted:

$$c + \sum_{p=k+1}^{i-1} \mathbf{M}(s_{p-1}, s_p, \mathbf{x}) \ , \tag{6.3}$$

and label it with $\mathbf{x}$ and $\mathbf{E}$, where $\mathbf{E}$ is some function satisfying:

$$(s_{i-1}, s_i) \in \mathcal{S}(\mathbf{E}) \subseteq \{(s,q) | \psi \in \mathbf{C}(s,q)\} \ . \tag{6.4}$$

Rule 2: This rule apply if ψ is of the form $\mathbf{x} \geq c$. Let k and $\mathbf{E}$ be as defined by (6.2) and (6.4). In this case we add to the graph an edge from k to i weighted:

$$-c - \sum_{p=k+1}^{i-1} \mathbf{M}(s_{p-1}, s_p, \mathbf{x}) \ , \tag{6.5}$$

and label it with $\mathbf{x}$ and $\mathbf{E}$.

Rule 3: This rule apply if ψ is of the form $\mathbf{x} - \mathbf{y} \leq c$. Let k and $\mathbf{E}$ be as defined by (6.2) and (6.4), and let m be the index of the last transition before the i-th one on which $\mathbf{y}$ was reset, i.e.:

$$m = \max\{j < i | j = 0 \text{ or } \mathbf{M}(s_{j-1}, s_j, \mathbf{y}) = \perp\} \ . \tag{6.6}$$

In this case, we add an edge from m to k weighted:

$$c + \sum_{p=k+1}^{i-1} \mathbf{M}(s_{p-1}, s_p, \mathbf{x}) - \sum_{p=m+1}^{i-1} \mathbf{M}(s_{p-1}, s_p, \mathbf{y}) \ , \tag{6.7}$$

and label it with $\mathbf{x}$ and $\mathbf{E}$.

In the failure analysis phase, an edge $(i, j, c, \mathbf{x}, \mathbf{E})$ indicates that a consistent timing must satisfy:

$$\lambda_i - \lambda_j \leq c \ .$$

However, in the failure elimination phase the same edge indicates that transitions in $\mathcal{S}(\mathbf{E})$ cannot occur unless either $i \geq j$ and $\mathbf{x} \leq c$ is satisfied, or $i < j$ and $\mathbf{x} \geq -c$ is satisfied. To account for this difference in interpretation, we will change the weight of all edges in the over-constrained loop before the failure elimination. If an edge is generated by Rule 1 above we change its weight from (6.3) to c. For example, the edge $1 \rightarrow 3$ induced by the inequality $(\mathbf{y}_i \leq r_i) \in \mathbf{C}(s_2, s_3)$ would have the weight $r_i + \Delta$ during the search for an over-constrained loop, and the weight r_i in the failure elimination phase. Similarly, we will change the the weight of edges generated by Rule 2 from (6.5) to $-c$.

Finally, we will replace any edge of the over-constrained loop generated by rule 3 with two edges: one from m to i weighted 0 and labeled $\mathbf{y}$, $\mathbf{E}$, and one from i to k weighted

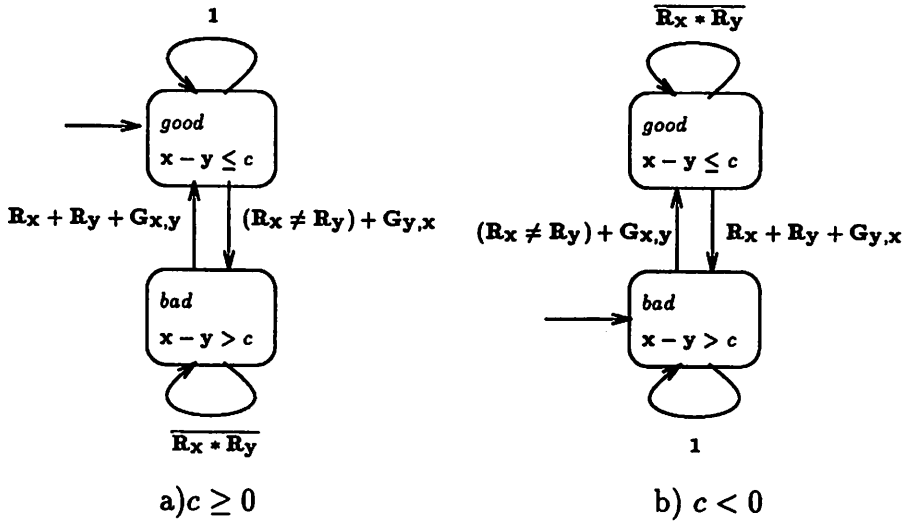


Figure 6.2: Guard automata for arbitrary TAD's.

c and labeled $\mathbf{x}$, $\mathbf{E}$. Again, we do this in a manner that preserves the simplicity of the loop: if node i already appears in the over-constrained loop, then these two edges are connected to a distinct copy of node i. Thus, this modification creates a new peak node. When that node is eliminated in the modify function, the automaton $\langle \mathbf{x} - \mathbf{y} \leq c \rangle$ is generated, and transitions in $\mathcal{S}(\mathbf{E})$ (including $s_{i-1} \rightarrow s_i$) are disabled in its *bad* state.

6.3 Guard automata

The definition of guard automata in section 4.4.3 does not represent an abstraction of an arbitrary TAD. The problem is that a guard automaton $\langle \mathbf{x} - \mathbf{y} \leq c \rangle$ can change states only if one of the timers is reset, while in a TAD the truth of $\mathbf{x} - \mathbf{y} \leq c$ can also be changed by decrementing $\mathbf{x}$ and $\mathbf{y}$ by a different amount. The generalized definition of guard automata is shown in Figure 6.2, where expression $\mathbf{G_{x,y}}$ denotes the characteristic function of transition on which $\mathbf{x}$ is decremented more than $\mathbf{y}$ is, i.e.:

$$\mathbf{G_{x,y}} \equiv \mathcal{X}\{(s,q) \mid \mathbf{M}(s,q,\mathbf{x}) \neq \perp,\ \mathbf{M}(s,q,\mathbf{y}) \neq \perp,\ \mathbf{M}(s,q,\mathbf{x}) > \mathbf{M}(s,q,\mathbf{y})\}\ .$$

It is easy to check that for the special case of timed automata (i.e. $\mathbf{G_{x,y}} \equiv \mathbf{0}$), the automata in Figure 6.2 are equivalent to those in Figure 4.6.

6.4 Failure elimination

The modify function for arbitrary TAD's is shown shown in Figure 6.3. First four steps are identical to the first four steps of the timed automata version of modify shown in Figure 4.7. In step 5 we compute the index of the last transition where A_k could have moved to the *good* state, that is the larger of i and j, or the index of the last transition where $\mathbf{x}$ has been decremented more than $\mathbf{y}$. In case of timed automata, m is always equal to the larger of i and j, thus steps 7 and 8 are never executed. Also, steps 9–11 are the same as steps 5–7 of the timed automata algorithm in Figure 4.7.

Let us now consider the effect of steps 7 and 8 on the example of the failure report (6.1) with $p_i = 4$, $r_i = 2$ and $\Delta = 1$. In this case, we can find a single-edge over-constrained loop (1) $\rightarrow$ (1) weighted -1, induced by inequality $\mathbf{y}_i - \mathbf{x}_i \leq -2$ in $\mathbf{C}(run_2, dead)$. Therefore, before the failure elimination, we replace that edge with edges $(0) \xrightarrow{\mathbf{x}_i \geq 0} (3)$ and $(3) \xrightarrow{\mathbf{y}_i \leq -2} (0)$.

In the first pass through the **while** loop, $\tilde{A}$ is composed with $A_3 = \langle \mathbf{y}_i - \mathbf{x}_i \leq -2 \rangle$ (step 2) and the transition $run_2 \rightarrow dead$ is disabled when A_3 is in the *bad* state (step 3). In step 5, m is chosen to be 2, and then the loop modified in steps 6–8. The modification reflects the fact that $\mathbf{y}_i - \mathbf{x}_i \leq -2$ can hold at $run_2 \rightarrow dead$, only if $\mathbf{y}_i - \mathbf{x}_i \leq -1$ holds before $\mathbf{y}_i$ is decremented by 1 on $run_1 \rightarrow run_2$. Therefore, we create a new peak node 2, elimination of which will generate $A_2 = \langle \mathbf{y}_i - \mathbf{x}_i \leq -1 \rangle$. This will happen in the second (and final) pass through the **while** loop. Also, in the second pass the transitions characterized by:

$$(\mathbf{ps}_{A_2} = bad) * (\mathbf{ns}_{A_3} = good) * \mathcal{X}\{(s,q) \,|\, \mathbf{M}(s,q,\mathbf{x}_i) = 0,\ \mathbf{M}(s,q,\mathbf{y}_i) = 1\} \qquad (6.8)$$

will be eliminated from the current $\tilde{A}$ (step 3). Now, A_2 must move to the *bad* state when both $\mathbf{x}_i$ and $\mathbf{y}_i$ are reset on $idle \rightarrow run_1$, thus by (6.8) A_3 must move to the *bad* state on $run_1 \rightarrow run_2$, so $run_2 \rightarrow dead$ is disabled, and the whole failure trace is eliminated.

6.5 Correctness

The proofs of Theorems 4.3 and 4.4 carry over directly to the extended TLE algorithm in Figure 6.3. Therefore, we conclude that the extended TLE algorithm can never terminate with the wrong answer. However, Lemma 4.5, and consequently Theorem 4.5 do

function modify($G, \tilde{A}, s_0, \ldots, s_n$)
/* G - an over-constrained loop */
/* $\tilde{A}$ - an abstraction of the system to be verified */
/* $s_0, \ldots, s_n$ - a failure report */
while there are edges $(i, k, w, \mathbf{x}, \mathbf{E})$, $(k, j, v, \mathbf{y}, \mathbf{H})$ in G s.t. $k > i, j$ **do**
step 1: $A_k := \langle \mathbf{y} - \mathbf{x} \leq w + v \rangle$;
step 2: $\tilde{A} := \tilde{A} \otimes A_k$;
step 3: $\tilde{A} := (S_{\tilde{A}}, \mathbf{I}_{\tilde{A}}, \mathbf{T}_{\tilde{A}} * \overline{(p_{A_k} = bad) * \mathbf{E} * \mathbf{H}}, \mathbf{F}_{\tilde{A}})$;
step 4: remove from G edges $(i, k, w, \mathbf{x}, \mathbf{E})$ and $(k, j, v, \mathbf{y}, \mathbf{H})$ and node k;
step 5: $m := \max\{p < k \mid p = i$ or $p = j$ or $\mathbf{G}_{\mathbf{x},\mathbf{y}}(s_{p-1}, s_p) = 1\}$;
step 6: $\mathbf{E} := \mathcal{X}\{(s, q) \mid \mathbf{M}(s, q, \mathbf{x}) = \mathbf{M}(s_{m-1}, s_m, \mathbf{x}),$
$\mathbf{M}(s, q, \mathbf{y}) = \mathbf{M}(s_{m-1}, s_m, \mathbf{y})\}$;
if $m > i$ and $m > j$ **then**
step 7: add to G an edge $(i,\ m,\ w + \mathbf{M}(s_{m-1}, s_m, \mathbf{x}),\ \mathbf{x},\ (\mathbf{ns}_{A_k} = good) * \mathbf{E})$;
step 8: add to G an edge $(m,\ j\ ,v - \mathbf{M}(s_{m-1}, s_m, \mathbf{y}),\ \mathbf{y},\ (\mathbf{ns}_{A_k} = good) * \mathbf{E})$;
else if $i < j = m$ **then**
step 9: add to G an edge $(i,\ m,\ w + v + \mathbf{M}(s_{m-1}, s_m, \mathbf{x}),\ \mathbf{x},\ (\mathbf{ns}_{A_K} = good) * \mathbf{E})$;
else if $j < i = m$ **then**
step 10: add to G an edge $(m,\ j,\ w + v - \mathbf{M}(s_{m-1}, s_m, \mathbf{y}),\ \mathbf{y},\ (\mathbf{ns}_{A_k} = good) * \mathbf{E})$;
else /* $i = j = m$ */
step 11: **return** $\tilde{A}$;
end if
end while
end function

Figure 6.3: Failure elimination for timed automata with decrement.

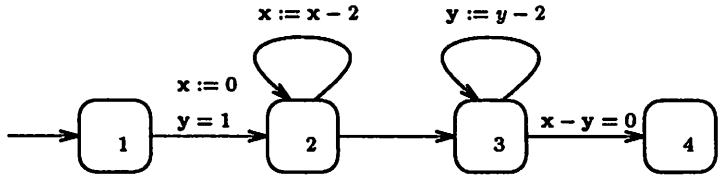


Figure 6.4: An automaton for which the extended TLE algorithm does not terminate.

not hold, thus the extended TLE algorithm may not terminate. This is not surprising because the termination would contradict the undecidability result of Theorem 4.2.

Figure 6.4 shows an example of a TAD for which the extended TLE algorithm does not terminate. If we let state 4 be the unique final state, then the language is empty, because every run from 1 to 4 has the form:

$$(1) \to (2) \underbrace{\to (2) \cdots \to (2) \to (2)}_{i \text{ times}} \to (3) \underbrace{\to (3) \cdots \to (3) \to (3)}_{j \text{ times}} \to (4) \ ,$$

so the difference between $\mathbf{x}$ and $\mathbf{y}$ on $(3) \to (4)$ is $2 * (j - i) - 1$ and can never be zero. However, there is no finite constant beyond which all the values of $\mathbf{x} - \mathbf{y}$ are equivalent, and thus not all the runs can be eliminated with any finite set of guard automata.

The example in Figure 6.4 shows that extended TLE algorithm may not terminate when the language of a TAD is empty. Under some mild assumptions on the failure reports returned by the verify function, we can show that the extended TLE algorithm will always terminate if the language of a TAD is not empty. For example, it is not too hard to guarantee that the verify function always returns the shortest failure report [HBK93]. If that is the case, we can claim that the true accepting run will eventually be returned by verify, because there are only finitely many shorter runs, so they can all be eliminated in finitely many iterations of the TLE algorithm.

For the model of PATHO in Figure 6.1 we can argue about bounds on values of $\mathbf{y}_i$ as follows:

- $\mathbf{y}_i$ can be decremented only while $\mathbf{y} < 1$, and it is never compared to any larger constant, thus all values of $\mathbf{y}$ beyond 1 are equivalent,
- if the number of tasks is n and $n < 20$, then the value of $\mathbf{y}_i$ can never be smaller than

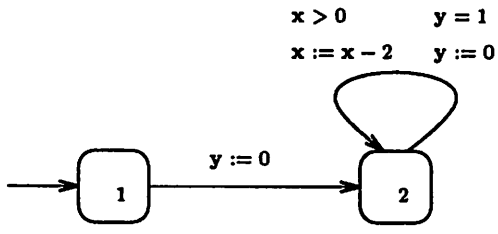


Figure 6.5: An automaton with no infinite sequences admitting consistent timing.

$-(n-1)$, because in time interval of 20 time units $\mathbf{y}_i$ can be decremented at most $(n-1)$ times.

Since we can establish a lower and an upper bound on the values of $\mathbf{y}_i$ of interest, we can generate a finite stable partition.

6.6 Extensions to infinite sequences

In the case of timed automata, extension to infinite sequences was relatively straightforward, because an infinite run of a timed automaton can be properly timed if and only if all of its finite prefixes can. Using this fact, we could analyze infinite failure traces by analyzing their finite prefixes, and we could represent every timing violation by a finite over-constrained loop.

This property does not hold for arbitrary TAD's, as demonstrated by the example in Figure 6.5 (assume there are no fairness constraints). The first transition $1 \rightarrow 2$ can occur at any time, but thereafter the transitions $2 \rightarrow 2$ occur regularly, one time unit apart (ensured by timer $\mathbf{y}$). In other words, the value of $\mathbf{x}$ can initially be arbitrarily large, but after that it will be decreased by one on every transition. For every finite prefix, it is possible to select large enough initial value of $\mathbf{x}$ such that $\mathbf{x} > 0$ is satisfied throughout the sequence. However, $\mathbf{x} > 0$ must eventually be violated, thus an infinite run does not admit a consistent timing. This is an example of a timing violation that cannot be represented by an over-constrained loop, and thus cannot be eliminated with guard automata. In general, the behavior which is not acceptable, but whose finite prefixes are all acceptable has to be eliminated by fairness constraints. In our example, the constraint that state 2 cannot occur infinitely often would suffice. Formulating general rules for elimination of this type

of timing violations is still an open problem.

6.7 Experimental results

We have tested an implementation of the extended TLE algorithm on the example consisting of three TAD's like those in Figure 6.1, with a choice of parameters $p_i = 20s$, $r_i = \Delta = 1s$. Experiments were performed on the same machine as the rest of results in this work (DEC MIPS 5000 workstation, 440Mb of memory).

It takes $16s$ of CPU time to verify that the interrupt of the task with the highest priority will never be missed. For the task with the middle priority it takes $54s$, and for the lowest priority task the program did not terminate after more than twelve hours of CPU time. This indicates that even for simple systems, the verification problem is hard and further advancements are necessary to make the verification practical. One approach could be extending guided verification to arbitrary TAD's.

Chapter 7

Arrays of Identical Components

In this chapter we study *network invariants*, abstractions of systems consisting of arbitrary many identical components. In particular, we study a case when an instance of some fixed size serves as an invariant. We show that the existence of such an invariant is undecidable, present a sem-decision procedure that will find it, if one exists, and finally give some conditions under which such an invariant does not exist. These conditions can be checked in finite time, and if satisfied, they can be used in further searches for an invariant.

7.1 Introduction

It is often the case that parts of large systems are generated by replicating the same basic cell. Ideally, an abstraction of such a system should not depend on the actual number of components. Such an abstraction is called a *network invariant.* Once an invariant of manageable size is found it allows:

- a verification of a large system with a fixed number of components; and at the same time also
- a verification of the entire class of systems with the same structure but with different number of components.

This is of particular interest for distributed systems where algorithms (e.g. mutual exclusion) are usually designed to be correct for systems of any number of concurrent processes, therefore the algorithms are not verified until instances of *all* sizes are.

We address a problem of finding an invariant automatically. Intuitively, an invariant of a class of systems is a finite-state system that can exhibit any behavior that some system in the class can, and possibly some additional behaviors. Thus, in formal verification by language containment, an invariant is a conservative simplification: if an abstraction is verified, so is every system in the class, but not vice versa. A *tight* invariant is an exact abstraction in a sense that again if it is verified, so is every system in the class, but if a tight invariant is not verified, than there must exist at least one system in the class which exhibits undesirable behavior. Thus, a tight invariant must exhibit *exactly* those behaviors that are exhibited by systems in the class.

Finding a tight invariant is easy if a *finite invariant* exists, i.e. if there exists a *finite* subclass such that any behavior exhibited by any system in the class is also exhibited by some system in the subclass. The main result of this chapter is the test that can show that a finite invariant does not exist. The test is constructive in a sense that if successful, it identifies a set of behaviors that cannot be "covered" by any finite subclass, but must be exhibited by a tight invariant. Once identified, such a behavior can be added to the behavior of some finite subclass to possibly generate a tight (but not finite) invariant. Unfortunately, we can not hope for a general algorithm that identifies all such sets of behaviors, because the existence of a finite invariant is undecidable (see Theorem 7.1).

7.2 Iterative systems

To develop our results we have to restrict somewhat the class of systems we consider. In particular, we consider only networks of chain structure, i.e. every component can communicate only with its left and right neighbors. We also assume that a state of the basic cell is fully observable (see Fig. 7.1). Also, we consider only automata on finite strings. Thus, using our approach only safety properties can be verified. These restrictions still enable us to model many regular hardware arrays, such as stacks, FIFO buffers, counters and multi-processor busses. Other examples that fit into our framework are a token passing mutual exclusion protocol [WL90] and the ever-so-popular Dining Philosophers Problem (e.g. [KM89]).

To define an array of identical components, we first define a generic *cell* as a quadruple (S, I, T, F), where S is some non-empty set of states. I (initial states) and F (final states) are subsets of S, and a transition relation is given by $T \subseteq S^2 \times S^2 \times S^2$.

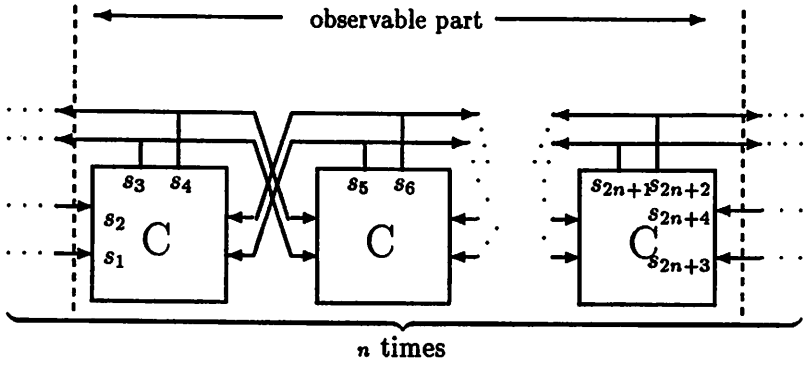


Figure 7.1: An open network N_n.

Instances of a generic cell differ by the choice of variables used to represent characteristic functions of I, F, and T. Instances that are connected together share some components of the I/O vectors. More precisely, the i-the *instance* of a cell (S, I, T, F) is an automaton:

$$C_i = (S, \mathcal{X}(I), \mathcal{X}(T), \mathcal{X}(F)) \ , \tag{7.1}$$

with the present and next state variables $\mathbf{ps}_i$ and $\mathbf{ns}_i$ and a vector I/O variable:

$$(\mathbf{ps}_{i-1}, \mathbf{ns}_{i-1}, \mathbf{ps}_i, \mathbf{ns}_i, \mathbf{ps}_{i+1}, \mathbf{ns}_{i+1}) \ ,$$

all components of which range over S. Note that $\mathbf{ps}_i$ and $\mathbf{ns}_i$ are not only the state variables, but also I/O variables because they can be used by the other automata for coordination.

To eliminate possible ambiguities induced by variable renaming, we require that in (7.1):

$$supp(\mathcal{X}(T)) = \{\mathbf{ps}_{i-1}, \mathbf{ns}_{i-1}, \mathbf{ps}_i, \mathbf{ns}_i, \mathbf{ps}_{i+1}, \mathbf{ns}_{i+1}\} \ ,$$

and

$$((s_1, s_2), (s_3, s_4), (s_5, s_6)) \in T$$

if and only if $\mathcal{X}(T)$ is satisfied by the assignment:

$$\begin{aligned} \mathbf{ps}_{i-1} &= s_1 \ , & \mathbf{ns}_{i-1} &= s_2 \ , \\ \mathbf{ps}_i &= s_3 \ , & \mathbf{ns}_i &= s_4 \ , \\ \mathbf{ps}_{i+1} &= s_5 \ , & \mathbf{ns}_{i+1} &= s_6 \ . \end{aligned}$$

Occasionally, we will omit the subscript of a cell if it is implied by the context.

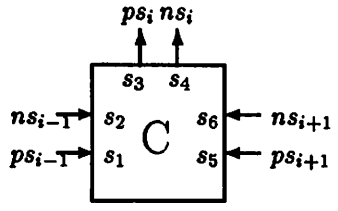


Figure 7.2: A basic cell.

Intuitively, I/O values of an instance consist of three parts: $\mathbf{ps}_{i-1}$ and $\mathbf{ns}_{i-1}$ are the present and the next state of the left neighbor, $\mathbf{ps}_i$ and $\mathbf{ns}_i$ are the present and the next state of the instance itself, and finally $\mathbf{ps}_{i+1}$ and $\mathbf{ns}_{i+1}$ are the present and the next state of the right neighbor, as shown in Fig. 7.2.

To formalize connecting cells into larger units we define *concatenation* "·". Concatenation is similar to automata composition except that the automaton $A \cdot B$ is well defined only if the I/O variable of A is a vector:

$$(\mathbf{ps}_{i-j}, \mathbf{ns}_{i-j}, \ldots, \mathbf{ps}_i, \mathbf{ns}_i, \mathbf{ps}_{i+1}, \mathbf{ns}_{i+1}) \text{ for some } j \in \{0, 1, \ldots, i\} \ ,$$

and the I/O variable of B is a vector:

$$(\mathbf{ps}_i, \mathbf{ns}_i, \mathbf{ps}_{i+1}, \mathbf{ns}_{i+1}, \ldots, \mathbf{ps}_{i+k}, \mathbf{ns}_{i+k}) \text{ for some } k \geq 0 \ .$$

The I/O variable of $A \cdot B$ is then:

$$(\mathbf{ps}_{i-j}, \mathbf{ns}_{i-j}, \ldots \mathbf{ps}_{i+k}, \mathbf{ns}_{i+k}) \ ,$$

and it is defined by:

$$A \cdot B = (S_A \times S_B, \mathbf{I}_A * \mathbf{I}_B, \mathbf{T}_A * \mathbf{T}_B, \mathbf{F}_A * \mathbf{F}_B) \ . \tag{7.2}$$

Given a cell with instances $C_1, C_2, \ldots$ and an automaton E (called an *environment*) with the vector I/O variable $(\mathbf{ps}_0, \mathbf{ns}_0, \mathbf{ps}_1, \mathbf{ns}_1)$ a *network* of length n is defined by:

$$N_n = E \cdot C_1 \cdot C_2 \cdot \ldots \cdot C_n \ .$$

If the behavior of the environment is unrestricted, i.e. if $\mathcal{L}(E) = (S^4)^*$ we say that the network is open. Otherwise, we say that the network is left-closed. We consider only open and left-closed networks, but the results are easily dualized for right-closed networks.

To compose networks into larger ones, only "peripheral" components of their languages need to be considered (see Fig. 7.1). Therefore, we introduce a notion of an *observable part*, first for elements of S^{2n+4}:

$$O((s_1, s_2, \ldots, s_{2n+4})) = (s_1, s_2, s_3, s_4, s_{2n+1}, s_{2n+2}, s_{2n+3}, s_{2n+4}) \ ,$$

and then, we extend the notion naturally to strings $s_1 s_2 \ldots s_k \in (S^{2n+4})^*$ by:

$$O(s_1 s_2 \ldots s_k) = O(s_1)O(s_2)\ldots O(s_k) \ ,$$

and languages $\mathcal{L} \subseteq (S^{2n+4})^*$ by:

$$O(\mathcal{L}) = \{x | x = O(y) \text{ for some } y \in \mathcal{L}\} \ .$$

For simplicity, we write $\mathcal{L}_n$ instead of $O(\mathcal{L}(N_n))$.

Given an automaton, we will denote with $O(A)$ an automaton satisfying:

$$\mathcal{L}(O(A)) = O(\mathcal{L}(A)) \ .$$

It is easy to see that $O(\mathcal{L}(\mathcal{A}))$ is just a projection of the language of A to its peripheral parts. It follows from the presentation in section 2.4.3 that $O(A)$ can always be constructed.

An *iterative system* $\{N_n | n \geq 1\}$ is the class of all networks of different length generated by the same cell and environment. If $\{N_n | n \geq 1\}$ is an iterative system, and if $\mathcal{L}_\infty = \bigcup_{n=1}^{\infty} \mathcal{L}_n$, then an *invariant* is any finite-state automaton A that satisfies $\mathcal{L}(A) \supseteq \mathcal{L}_\infty$. If $\mathcal{L}(A) = \mathcal{L}_\infty$, we say that A is a *tight invariant*. If in addition $\mathcal{L}(A) = \bigcup_{n=1}^{n^*} \mathcal{L}_n$ for some $n^* < \infty$, we say that A is a *finite invariant*.

Obviously, a tight invariant exists if and only if $\mathcal{L}_\infty$ is regular. One may try to find a "tightest regular invariant", i.e. an invariant that is not tight, but that has the language that is contained in the languages of all other invariants. Unfortunately, if $\mathcal{L}_\infty$ is not regular, such an invariant does not exist. To see this assume that A is such an invariant. Let x be some string in $\mathcal{L}(A) - \mathcal{L}_\infty$ (since $\mathcal{L}(A)$ is regular and $\mathcal{L}_\infty$ is not, such a string always exists). Let A' be such that $\mathcal{L}(A') = \mathcal{L}(A) - \{x\}$. Now, A' is an invariant and $\mathcal{L}(A) \not\subseteq \mathcal{L}(A')$, contradicting the assumption that A is the tightest regular invariant.

7.3 Examples

The following three examples illustrate three possible cases: when a finite invariant exists (Example 7.1), when a tight invariant exists, but a finite one does not (Example 7.2),

and finally when a tight invariant does not exist (Example 7.3). All three examples are abstractions of buffers with different disciplines of passing a token. In all three cases cells are initially in the *idle* state. The state *token* indicates that a particular cell holds a token. In Examples 1 and 2, a cell moves to a special *dead* state once it has delivered a token. In all examples, all states are final and all systems are open.

Example 7.1 *In this example a cell can hold a token for any (possibly infinite) number of steps before delivering it to its neighbor. A cell can deliver only one token. More precisely the transition relation of the i-th cell is defined by:*

$$\begin{aligned}
\mathbf{T} \equiv\ & \mathbf{ps}_i = \mathit{idle} \ * \ \mathbf{ns}_i = \mathit{idle} \ * \ (\mathbf{ps}_{1-1} \neq \mathit{token} + \mathbf{ns}_{i-1} \neq \mathit{dead}) \\
+\ & \mathbf{ps}_i = \mathit{idle} \ * \ \mathbf{ns}_i = \mathit{token} \ * \ \mathbf{ps}_{i-1} = \mathit{token} * \mathbf{ns}_{i-1} = \mathit{dead} \\
+\ & \mathbf{ps}_i = \mathit{token} \ * \ \mathbf{ns}_i = \mathit{token} \\
+\ & \mathbf{ps}_i = \mathit{token} \ * \ \mathbf{ns}_i = \mathit{dead} \\
+\ & \mathbf{ps}_i = \mathit{dead} \ * \ \mathbf{ns}_i = \mathit{dead} \qquad .
\end{aligned}$$

In this case, a finite invariant exists. In fact, it is achieved for $n^* = 3$. For $n \geq 3$, a language $\mathcal{L}_n$ can be described by the languages of the leftmost and the rightmost cell and the following additional constraint:

> "If the rightmost cell ever moves from *idle* to *token* it will happen *at least* $n-2$ steps after the leftmost cell leaves the *token* state."

Clearly, $\mathcal{L}_3$ (strictly) contains all $\mathcal{L}_n$'s, $n > 3$.

Example 7.2 *In this example a cell holds a token for exactly one step. Again, once it delivers a single token, a cell will move to the dead state. The transition relation of the basic cell is:*

$$\begin{aligned}
\mathbf{T} \equiv\ & \mathbf{ps}_i = \mathit{idle} \ * \ \mathbf{ns}_i = \mathit{idle} \ * \ \mathbf{ps}_{i-1} = \mathit{idle} \\
+\ & \mathbf{ps}_i = \mathit{idle} \ * \ \mathbf{ns}_i = \mathit{token} \ * \ \mathbf{ps}_{i-1} = \mathit{token} \\
+\ & \mathbf{ps}_i = \mathit{token} \ * \ \mathbf{ns}_i = \mathit{dead} \ * \ \mathbf{ps}_{i-1} = \mathit{dead} \\
+\ & \mathbf{ps}_i = \mathit{dead} \ * \ \mathbf{ns}_i = \mathit{dead} \ * \ \mathbf{ps}_{i-1} = \mathit{dead} \qquad .
\end{aligned}$$

In this case a finite invariant does not exist. All strings in $\mathcal{L}_n$ ($n \geq 2$) that are long enough must satisfy the following constraint:

> "The rightmost cell moves *idle* to *token exactly* $n-2$ step after the leftmost cell leaves the *token* state."

Obviously, for all $n \geq 3$ there are some strings in $\mathcal{L}_{n+1}$ which are not in $\mathcal{L}_n$. However, a tight invariant exists, and it is similar to the one in the previous example, except that the peripheral cells are restricted to remain in the *token* state for one step only.

Example 7.3 *This example is similar to the previous one, except that once a cell delivers a token it will move back to the idle state and become ready to accept a new token.*

$$\begin{aligned} \mathbf{T} \equiv\ & \mathbf{ps}_i = \mathit{idle} * \mathbf{ns}_i = \mathit{idle} * \mathbf{ps}_{i-1} \neq \mathit{token} \\ +\ & \mathbf{ps}_i = \mathit{idle} * \mathbf{ns}_i = \mathit{token} * \mathbf{ps}_{i-1} = \mathit{token} \\ +\ & \mathbf{ps}_i = \mathit{token} * \mathbf{ns}_i = \mathit{idle} \end{aligned} .$$

In this case $\mathcal{L}_\infty$ is not regular, so a tight invariant cannot exist. Indeed, $\mathcal{L}_\infty$ must include $\mathcal{L}_1$ and all strings for which there exists $k \geq 0$ such that the rightmost cell moves from *idle* to *token* exactly k steps after the leftmost cell leaves the *token* state. Notice that for any given string k must be constant. It is straightforward to show that such a language is not regular.

However, if we include in the description of the system an environment which allows at most one token in the system, a tight invariant exists and is similar to the one in Example 2.

7.4 Computing a finite invariant

7.4.1 Decidability and existence

First, we describe a particular iterative system that we use for proving the key result of this section. For an arbitrary (one-way tape, deterministic) Turing machine TM with an empty tape, let the iterative system $IS(TM)$ be generated by the cell whose instances are composition of two finite-state components ($TAPE$ and $CNTRL$) described below.

The states of the component denoted by $TAPE$ are in 1–1 correspondence to the tape alphabet of TM. The initial state is blank. The state of the $TAPE$ component of the i-th instance is at all times equal to the value of the i-th letter on the tape.

The component $CNTRL$ is a copy of the finite-state control of TM augmented by two special states L and R. Intuitively, the i-th instance of $CNTRL$ is associated with the i-th tape position. It is in the L state if the TM's head is at some position to the left, and

it in the R state if the head is somewhere to the right. If $CNTRL$ is any other state, we say that the cell is active. The initial state of $CNTRL$ is L. While in it, a cell waits until its left neighbor is active and have to move the head to the right. At that point, $CNTRL$ moves to appropriate state of TM's control, possibly changing the state of $TAPE$ and after that moves to L or R depending on whether a head has to be moved left or right. Similarly to L, in state R an instance monitor its right neighbor.

An environment E of $IS(TM)$ is almost identical to the basic cell, except:

- it does not have an L state, hence it does not monitor its left neighbor,
- the component $CNTRL$ must be modified to halt whenever TM's control calls for move to the left,
- the initial state of $CNTRL$ must be equal to the initial state of TM's control.

It should be clear from the description that $IS(TM)$ is left-closed and that it simulates TM. In particular, the following result holds:

Lemma 7.1 *A left-closed iterative system $IS(TM)$ has a finite invariant if and only if TM uses finite memory.*

Proof. Assume that TM uses at most M tape symbols. Then $\mathcal{L}_n$ is the same for all $n > M$. Hence, an automaton with the language $\bigcup_{n=1}^{M+1} \mathcal{L}_n$ is a finite invariant. On the other hand, if TM uses infinitely many tape locations for all n there must be a string in $\mathcal{L}_n$ not in $\mathcal{L}_k$ for any $k < n$, because cell n must be activated for the first time at least one step after the k-th cell is. □

Theorem 7.1 *The existence of a finite invariant for a left-closed iterative system is undecidable.*

Proof. It is undecidable whether an arbitrary one-way tape, deterministic Turing machine with empty tape uses finite memory, and by Lemma 7.1 that problem is reducible to deciding the existence of a finite invariant for a left-closed iterative system. □

At present, it is not clear whether this proof can be extended to open systems. In fact, this result is similar to Theorem 4.3 in [WL90] and Theorem 4 in [Hen61]. In all cases, the proofs substantially rely on the ability to distinguish one cell in the network: in our case, it is the environment, in [WL90] the first cell is explicitly distinguished, and in [Hen61]

one cell is distinguished by different boundary condition. Therefore, the decidability of the existence of a finite invariant for open systems is still an open problem.

Theorem 7.2 *Let* $\{N_n = E \cdot C_1 \cdot \ldots \cdot C_n | n \geq 1\}$ *be an iterative system, and let* A *be some automaton satisfying:*

$$\mathcal{L}_1 \subseteq \mathcal{L}(A) , \tag{7.3}$$

$$O(\mathcal{L}(A \cdot C)) \subseteq \mathcal{L}(A) , \tag{7.4}$$

Then A *is an invariant of* $\{N_n | n \geq 1\}$.

Proof. We prove by induction that $\mathcal{L}_n \subseteq \mathcal{L}(A)$ for all $n \geq 1$. The base case ($n = 1$) holds by (7.3). It is straightforward to show that concatenation and projection are monotone with respect to language containment, thus the inductive assumption:

$$\mathcal{L}_n = \mathcal{L}(O(N_n)) \subseteq \mathcal{L}(A) ,$$

implies:

$$\mathcal{L}_{n+1} = O(\mathcal{L}(O(N_n) \cdot C) \subseteq O(\mathcal{L}(A \cdot C)) .$$

Applying (7.4) to the relation above completes the proof. □

If an automaton satisfies (7.3) and (7.4) we say that it is an *inductive* invariant. Both Kurshan and McMillan [KM89] and Wolper and Lovinfosse [WL90] require, by definition, for an invariant to be inductive. We have adopted a broader definition, motivated by the application to language containment. Still, Theorem 7.2 provides the only finite procedure known to us, for verifying that a given automaton with non-trivial language is indeed an invariant.

Note that Theorem 7.2 provides only one way implication: it is quite possible that an automaton is an invariant, but not an inductive one. In fact, Wolper and Lovinfosse [WL90] have shown that there exists an iterative system $\{N_n | n \geq 1\}$ and an automaton P such that P is an invariant of $\{N_n | n \geq 1\}$, but there does not exist an inductive invariant of $\{N_n | n \geq 1\}$ whose language is contained in $\mathcal{L}(P)$. Fortunately, a finite invariant is also an inductive one, as shown by the following theorem.

Theorem 7.3 *A finite invariant of an iterative system* $\{N_n = E \cdot C_1 \cdot \ldots \cdot C_n | n \geq 1\}$ *exists if and only if there exists* $n^* < \infty$ *such that:*

$$\mathcal{L}_{n^*+1} \subseteq \bigcup_{n=1}^{n^*} \mathcal{L}_n . \tag{7.5}$$

```
        function finite_invariant (E, C)
          /* E, C - an environment and a basic cell of an iterative array */
step 1:   let A be such that L(A) = L_1;
          for n = 2, to ∞
step 2:     if O(L(A · C_n)) ⊆ L(A) then return A;
step 3:     modify A such that L(A_new) = L(A_old) ∪ O(L(A_old · C_n));
          end for
        end
```

Figure 7.3: A semi-decision procedure for finding a finite invariant.

If (7.5) holds, then an automaton A satisfying $\mathcal{L}(A) = \bigcup_{n=1}^{n^*} \mathcal{L}_n$ is an invariant of $\{N_n | n \geq 1\}$.

Proof. The "only if" part is obvious. It is also straightforward to show that $O(\mathcal{L}(A \cdot C)) = \bigcup_{n=2}^{n^*+1} \mathcal{L}_n$, thus we have:

$$\mathcal{L}_{n^*+1} \subseteq \bigcup_{n=1}^{n^*} \mathcal{L}_n \iff \bigcup_{n=2}^{n^*} \mathcal{L}_{n+1} \subseteq \bigcup_{n=1}^{n^*} \mathcal{L}_n \iff O(\mathcal{L}(A \cdot C)) \subseteq \mathcal{L}(A) \ .$$

From the definition of A, $\mathcal{L}_1 \subseteq \mathcal{L}(A)$ follows trivially, so by Theorem 7.2 A is an invariant. □

This immediately gives us the semi-decision procedure shown in Figure 7.3. If the procedure terminates, it will return a finite invariant. However, if a finite invariant does not exist, the procedure will not terminate.

7.4.2 Proving non-existence of a finite invariant

In this section, we show a sufficient condition for non-existence of a finite invariant. The condition can be checked algorithmically, and, if satisfied, it provides useful information on sets of strings that every invariant must include in its language.

Our results hold for a generic *open* iterative system induced by a cell (S, I, T, F). Unless stated otherwise, we assume that s is some string in $(S^{2n})^*$. We use $|\cdot|$ to denote the length of a string. To refer to parts of a string we use the naming scheme detailed in

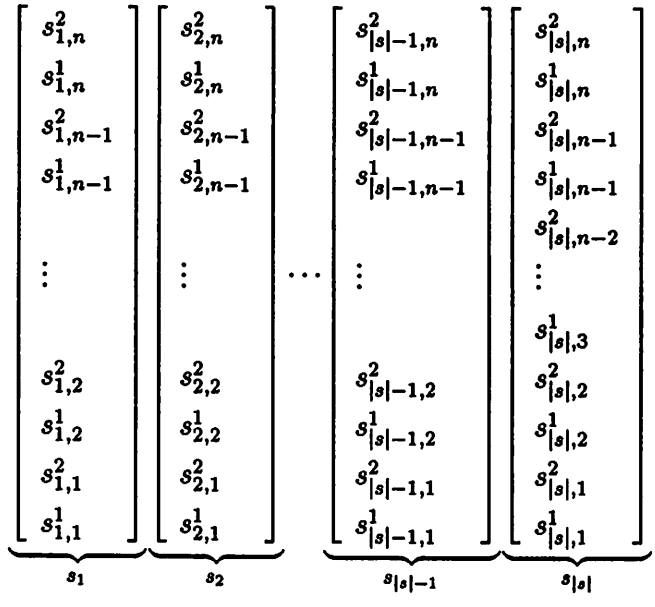


Figure 7.4: A naming scheme for parts of a string.

Fig. 7.4. We call a pair $s_{x,y} = (s^1_{x,y}, s^2_{x,y})$ a *transition*. If s, t and u are transitions, for simplicity we write $(s,t,u) \in T$ instead of:

$$((s^1,s^2),(t^1,t^2),(u^1,u^2)) \in T \ .$$

The reader might find it useful to visualize claims in this section, as suggested in Figure 7.5. It shows a symbolic representation of a string $s \in (S^{2n})^*$, with $n = 6$ and $|s| = 6$. Each transition is represented by an arrow. Conditions for s to be in $\mathcal{L}(N_{n-2})$ can be restated in terms of this symbolic representation as follows:

1. in every row (except the top and the bottom one) any two neighboring transitions must be consecutive, i.e.:

$$s^2_{x,y} = s^1_{x+1,y} \text{ for all } x = 1,\ldots,|s|-1,\ y = 2,\ldots,n-1 \ ; \tag{7.6}$$

we represent this constraint symbolically by a rectangle like the one marked C in Figure 7.5,

2. present-state components of all transitions in the first column (except the top and the bottom one) must be initial:

$$s^1_{1,x} \in I \text{ for all } x = 2,\ldots,n-1 \ , \tag{7.7}$$

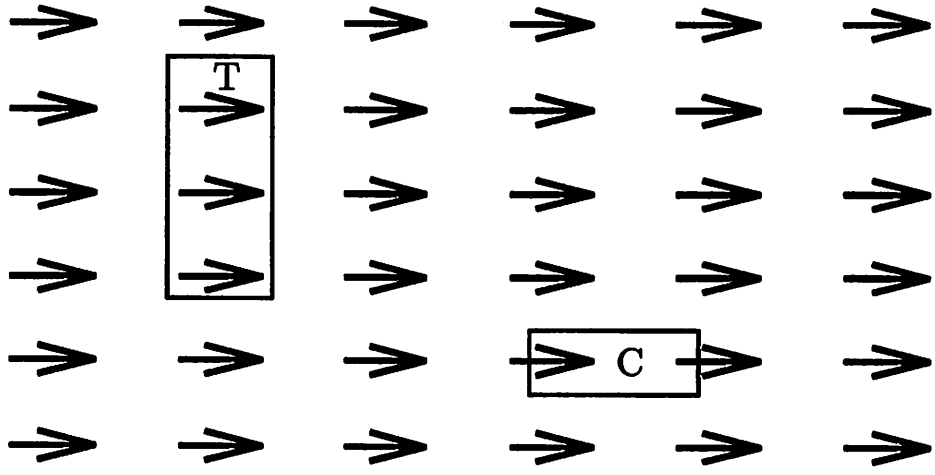


Figure 7.5: A symbolic representation of a string

3. in every column, any three neighboring transitions must satisfy transition relation T:

$$(s_{x,y-1}, s_{x,y}, s_{x,y+1}) \in T \text{ for all } x = 1, \ldots, |s|, y = 2, \ldots, n-1 \ ; \tag{7.8}$$

we represent this constraint symbolically by a rectangle like the one marked T in Figure 7.5,

4. next-state components of all transitions in the last column (except the top and the bottom one) must be final:

$$s^1_{|s|,x} \in F \text{ for all } x = 2, \ldots, n-1 \ , \tag{7.9}$$

An n-row string corresponds to the network N_{n-2}; the first and the last row do not correspond to states of any cell in the network. Therefore, they have to satisfy only transition requirements imposed by the first and the last cell in the networks (i.e. by the cells corresponding to the second and next-to-last row of the string).

To show non-existence of a finite invariant, we search for a sequence of strings: $x_1, x_2, \ldots$ satisfying $O(x_i) \in \mathcal{L}_i$, but $O(x_i) \notin \mathcal{L}_j$ for any $j < i$. We will show that in certain cases these relations can be established in a finite number of steps. We consider only a special case when x_{i+1} is obtained by extending x_i in a certain regular fashion. If that is the case we write $x_{i+1} = \alpha(x_i)$. Next, we define precisely the *extension operator* α.

Given strings $s \in (S^{2n})^*$ and $t \in (S^{2n+2})^*$ such that $|t| = |s| + 1$ we say that $t = \alpha_{ik}(s)$ if the following holds:

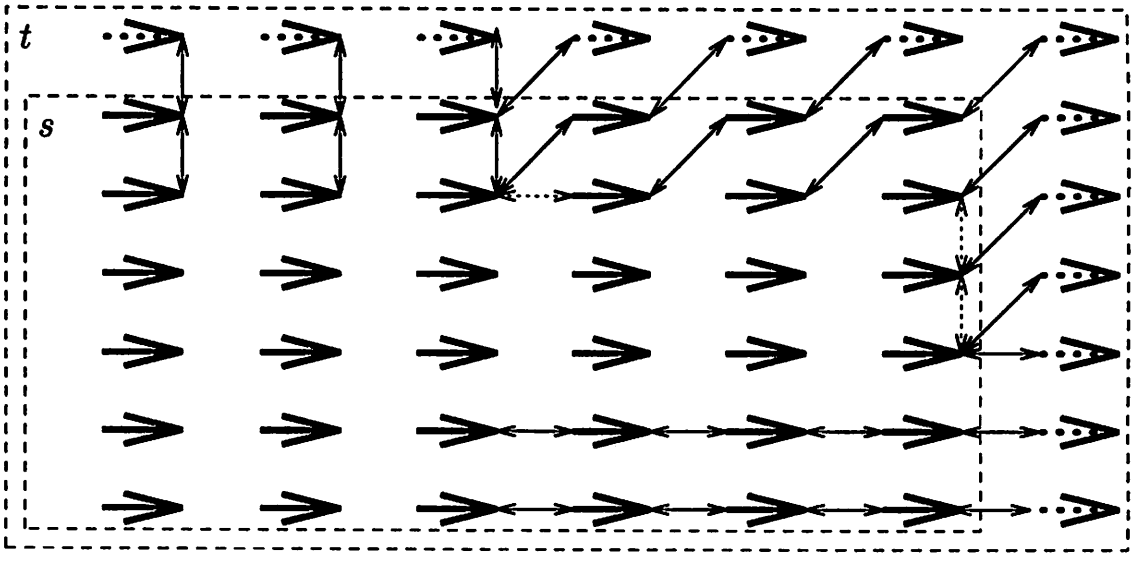


Figure 7.6: Strings satisfying $t = \alpha_{3,3}(s)$.

1. t obtained from s by adding one row and one column of transitions, i.e.:

$$t_{x,y} = s_{x,y} \text{ for all } x = 1,\ldots,|s|,\ y = 1,\ldots,n\ , \tag{7.10}$$

2. the observable part of t is the same as the observable part of s, except that the i'th column is repeated twice, i.e.:

$$O(t_x) = \begin{cases} O(s_x) & \text{for all } x = 1,\ldots,i\ , \\ O(s_{x-1}) & \text{for all } x = i+1,\ldots,|t|\ , \end{cases} \tag{7.11}$$

3. the last column of t is the same as the last column of s, except that the k'th row is repeated twice, i.e.:

$$t_{|t|,x} = \begin{cases} s_{|s|,x} & \text{for all } x = 1,\ldots,k\ , \\ s_{|s|,x-1} & \text{for all } x = k+1,\ldots,n+1\ . \end{cases} \tag{7.12}$$

Figure 7.6 shows an example of such a pair of strings. Full thin lines connect transition that must be equal to satisfy conditions 2 and 3 above. All of these equalities can be satisfied only if s satisfy certain constraints, required by the following proposition.

Proposition 7.1 *A string $\alpha_{ik}(s)$ exists if and only if all of the following hold:*

C1: $s_{x,n} = s_{x,n-1}$, *for all* $x = 1, \ldots, i$,

C2: $s_{x+1,n} = s_{x,n-1}$, *for all* $x = i, \ldots, |s| - 1$,

C3: $s_{x,1} = s_{i,1}$, $s_{x,2} = s_{i,2}$, *for all* $x = i, \ldots, |s|$.

If $\alpha_{ik}(s)$ *exists, then it is unique.*

Henceforth, we will assume that all extensions have common i and k, so without ambiguity we write $\alpha(s)$ for $\alpha_{ik}(s)$. Also, we use the following abbreviation for any $j \geq 0$:

$$\alpha^j(s) = \underbrace{\alpha(\alpha(\ldots\alpha}_{j \text{ times}}(s)\ldots)) \ .$$

Under some mild assumptions, $\alpha(s)$ inherits some interesting properties of s, as stated by the following lemma.

Lemma 7.2 *If* $s \in \mathcal{L}(N_{n-2})$, *and* s *satisfies:*

C4: $s_{i,n-1} = s_{i+1,n-1}$,

C5: $s_{|s|,k} = s_{|s|,k+1} = s_{|s|,k+2}$,

then $t = \alpha(s)$ *satisfies:*

$$t^2_{x,y} = t^1_{x+1,y} \text{ for all } x = 1, \ldots, |s| - 1, \ y = 2, \ldots, n \ , \tag{7.13}$$

$$t^1_{1,x} \in I \text{ for all } x = 2, \ldots, n \ , \tag{7.14}$$

$$t^1_{|t|,x} \in F \text{ for all } x = 2, \ldots, n \ , \tag{7.15}$$

$$(t_{x,y-1}, t_{x,y}, t_{x,y+1}) \in T \text{ for all } x = 1, \ldots, |t|, y = 2, \ldots, n-1 \ . \tag{7.16}$$

Proof.

(7.13) follows from (7.10), (7.12), **C4**, and (7.6);

(7.14) follows from (7.10), (7.12), and (7.7);

(7.15) follows from (7.11), and (7.9);

(7.16) follows from (7.10), (7.11), **C5**, and (7.8).

□

From this point on, we do assume properties **C4** and **C5**. We make this assumption without loss of generality because they are always satisfied by $\alpha^2(s)$, which also satisfies all other restriction on s mentioned in this section. Equalities imposed by **C4** and **C5**, are shown with thin dotted line in Figure 7.6.

The part of our strategy is to find a sequence of strings $x_1, x_2, \ldots$ satisfying $x_i \in \mathcal{L}_i$. The following lemma describes a case when all of these relations are satisfied if one of them is.

Lemma 7.3 *Let s satisfy* **C1–C5**, *and let $t = \alpha(s)$. If:*

C6: $(t_{x,n-1}, t_{x,n}, t_{x,n+1}) \in T$ *for all* $x = 1, \ldots, |t|$,

C7: $t^2_{|t|-1,x} = t^1_{|t|,x}$ *for all* $x = 2, \ldots, n$,

then:

$$s \in \mathcal{L}(N_{n-2}) \implies t \in \mathcal{L}(N_{n-1}) \ , \tag{7.17}$$

$$s \in \mathcal{L}(N_{n-2}) \implies \alpha^j(s) \in \mathcal{L}(N_{n-2+j}) \text{ for all } j \geq 0 \ . \tag{7.18}$$

Proof. Conditions (7.13)–(7.16), **C6** and **C7** are exactly the conditions for $t \in \mathcal{L}(N_{n-1})$ to be satisfied. Thus, (7.17) holds. If s satisfies **C6** and **C7** so does $\alpha(s)$. Therefore, we can repeatedly apply (7.17) to get (7.18). □

Figure 7.7 illustrates the requirements imposed by **C6** and **C7**.

The second part of our strategy is to find a sequence of strings $x_1, x_2, \ldots$ that $x_j \notin \mathcal{L}_i$ for any $j < i$. In Lemma 7.4 we establish a condition which enables us to prove this relation in a finite number of steps.

Lemma 7.4 *Let s satisfy* **C1–C5**, *and let $t = \alpha(s)$. If:*

C8: $s_{x,n-1}$ *is the unique element of the set*

$$\{(u^1, u^2) \mid \text{there exist } v, w, z \text{ such that } (v, t_{x,n}, (u^1, u^2)) \in T \text{ and } (w, t_{x+1,n}, (u^2, z)) \in T\}$$

for all $x = 1, \ldots, |s|$, *and*

C9: *there exists a sequence of transition $u_n, \ldots, u_4, u_3$ such that $u_n = t_{|t|,n}$, and u_x (for all $x = n-1, \ldots, 3$) is the unique element of the set:*

$$\{v \mid \text{there exists } w \text{ such that } (w, u_{x+1}, v) \in T\} \ ,$$

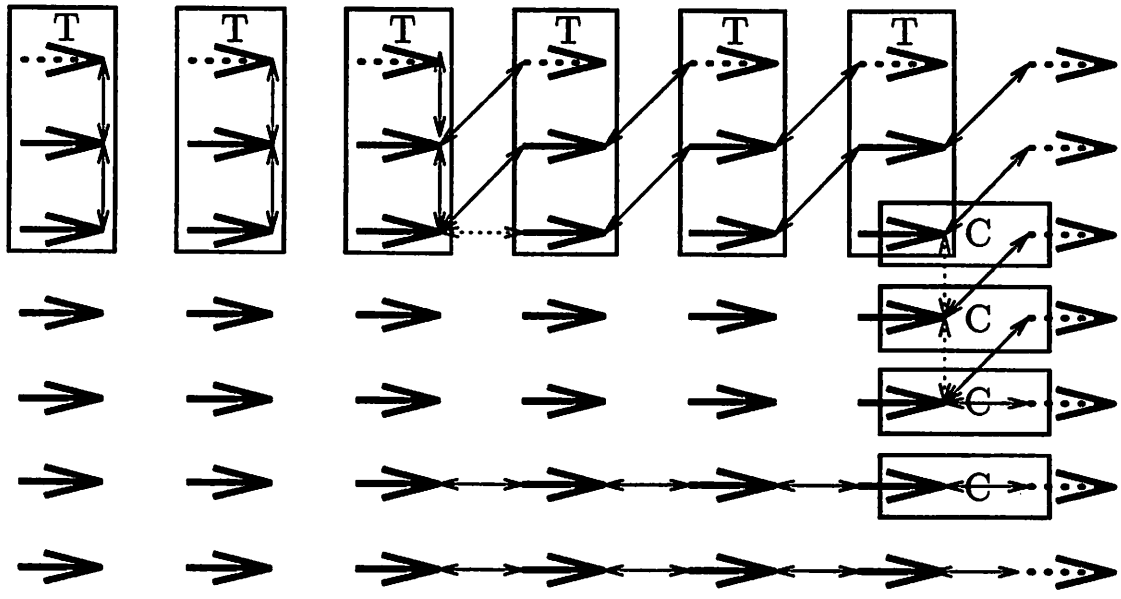


Figure 7.7: An illustration of Lemma 7.3.

then:

$$O(t) \in \mathcal{L}_{n-1} \implies O(s) \in \mathcal{L}_{n-2} \ , \tag{7.19}$$

$$O(s) \notin \mathcal{L}_{n-2} \implies O(\alpha^j(s)) \notin \mathcal{L}_{n-2+j} \text{ for all } j \geq 0 \ . \tag{7.20}$$

Proof. Assume $O(t) \in \mathcal{L}_{n-1}$, let string t' be such that $t' \in \mathcal{L}(N_{n-1})$ and $O(t') = O(t)$. Also, let s' be the string obtained by removing from t' the last row and the last column. It follows from **C8** that the next-to-last row of s' is exactly equal to the next-to-last row of s. Since **C1–C3** also hold, we have that $O(s') = O(s)$. We claim that $s' \in \mathcal{L}(N_{n-2})$ and thus $O(s) \in \mathcal{L}_{n-2}$.

That s' satisfies conditions analog to (7.6), (7.7), and (7.8) follows directly from $t' \in \mathcal{L}(N_{n-1})$. It follows from **C9** that $t'_{|t'|,x} = u_x$ for all $x = n, \ldots, 3$, so $t' \in \mathcal{L}(N_{n-1})$ also implies $u^1_x \in F$, for all $x = 3, \ldots, n$. It follows from **C2** that $s'_{|s'|,n-1} = t'_{|t'|,n} = u_n$, so we can again apply **C9** to obtain $(s'_{|s'|,x})^1 = u^1_{x+1} \in F$, for all $x = n-1, \ldots, 2$.

It is easy to check that $\alpha(s)$ satisfies **C8** and **C9** if s does. Therefore, we can repeatedly apply (7.19) to get (7.20). □

Figure 7.8 illustrates the chain of reasoning in Lemma 7.4. Values of the transition in row 5 (from the bottom) are implied from transitions in row 6 by **C8**. Transitions in the

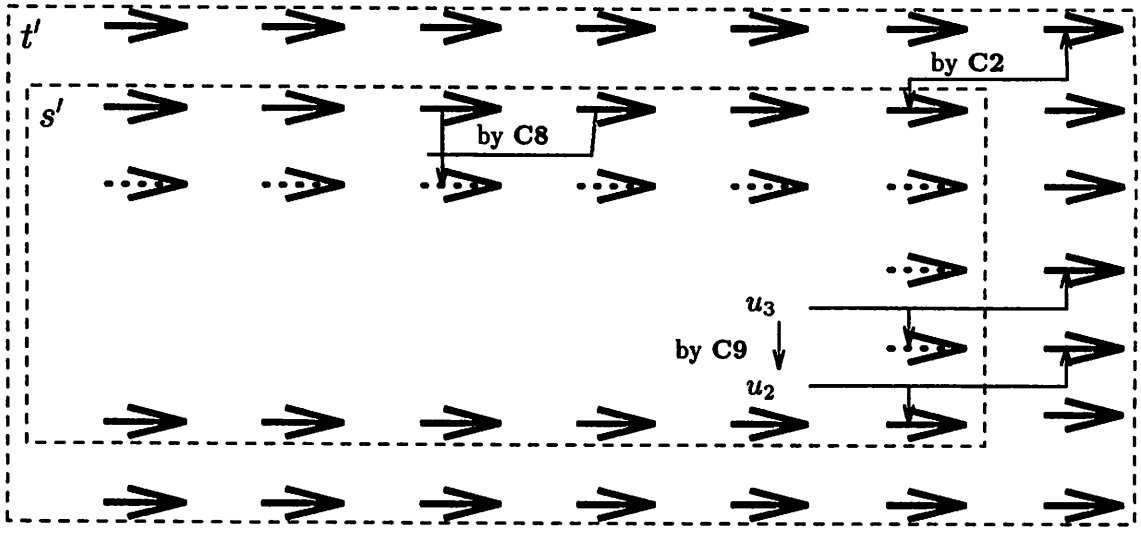


Figure 7.8: An illustration of Lemma 7.4.

last column are implied by **C9**, and they must also appear in column 6 by **C2**.

A reader will notice that the condition **C9** is used only to establish termination. If all the states of the basic cell are final, Lemma 7.4 holds even if **C9** is not satisfied.

We are now ready to postulate sufficient conditions for the non-existence of a finite invariant.

Theorem 7.4 *If s is such that it satisfies* **C1–C9** *as well as:*

C10: $s \in \mathcal{L}(N_{n-2})$,

C11: $O(\alpha^j(s)) \notin \mathcal{L}_{n-2}$, for all $j > 0$,

C12: $O(\alpha^j(s)) \notin \mathcal{L}_m$, for all $j \geq 0$, $m = 1, \ldots, n-3$,

then:

a) *a finite invariant does not exist,*

b) $\mathcal{L}_\infty \supseteq \{O(\alpha^j(s)) | j \geq 0\}$.

Proof. From **C10** and Lemma 7.3 we have:

$$O(\alpha^j(s)) \in \mathcal{L}_{n-2+j} \text{ for all } j \geq 0 \ . \tag{7.21}$$

We can rewrite **C11** as $O(\alpha^{j-m}(s)) \notin \mathcal{L}_{n-2}$ for all $j > 0$, $0 \leq m < j$, and combine it with Lemma 7.4 to get:

$$O(\alpha^j(s)) = O(\alpha^m(\alpha^{j-m}(s))) \notin \mathcal{L}_{n-2+m} \text{ for all } j > 0, \ 0 \leq m \leq j \ . \tag{7.22}$$

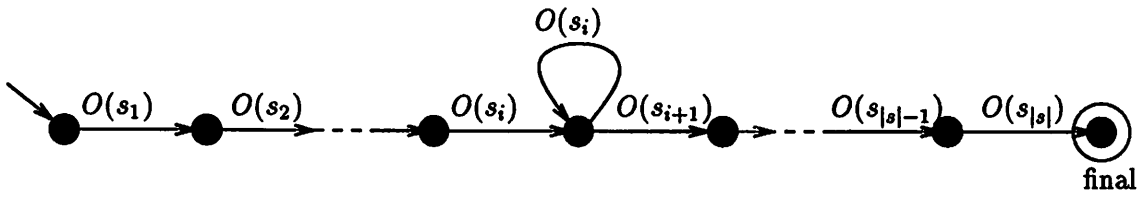


Figure 7.9: An automaton accepting the language $\{O(\alpha_{ik}^j(s))|j \geq 0\}$.

Thus, for every $j \geq 0$ there exists a string (specifically $O(\alpha^j(s))$) in $\mathcal{L}_{n-2+j}$ (by (7.21)), which is not in $\mathcal{L}_m$ for any $m < n-2+j$ (by **C12** and (7.22)), so a finite invariant cannot exist. Also, part **b)** follows from (7.21). □

Given a string s, it is straightforward to check whether conditions **C1–C10** are satisfied for α_{ik}. It is also straightforward to define an automaton accepting $\{O(\alpha_{ik}^j(s))|j \geq 0\}$ (see Figure 7.9), thus **C11** and **C12** can be checked by a language containment tool.

Consider Example 7.2 in section 7.3 and the following string[1] in $\mathcal{L}_3$:

$$s = \begin{bmatrix} idle \\ idle \\ idle \\ idle \\ token \end{bmatrix} \begin{bmatrix} idle \\ idle \\ idle \\ token \\ dead \end{bmatrix} \begin{bmatrix} idle \\ idle \\ token \\ dead \\ dead \end{bmatrix} \begin{bmatrix} idle \rightarrow token \\ token \rightarrow dead \\ dead \rightarrow dead \\ dead \rightarrow dead \\ dead \rightarrow dead \end{bmatrix}.$$

We can easily check that **C1–C12** are all satisfied for $\alpha_{2,1}$, and conclude that a finite invariant does not exist and that $\mathcal{L}_\infty \supseteq (\mathcal{L}_1 \cup \mathcal{L}_2 \cup \{O(\alpha_{2,1}^j(s))|j \geq 0\})$.

We can combine the search for a finite invariant with the search for a string satisfying **C1–C12** as shown in Figure 7.10. If the procedure **tight_invariant** terminates it will generate a tight (but possibly not finite) invariant A. Unfortunately, we can not claim that the procedure will terminate even if a tight invariant exists. It might be more productive to apply this procedure interactively, i.e. to let the user choose a string and then execute other steps automatically.

7.5 Related work and discussion

Although iterative systems have been studied for a long time [Hen61], only recently there has been a significant interest in the formal verification of such systems. Apt

[1] We omit writing $s_{x,y}^2$ for $x < 4$, and assume that $s_{x,y}^2 = s_{x+1,y}^1$.

```
        function tight_invariant (C)
          /* C = (S, I, T, F) - a basic cell of an open iterative array */
step 1:   let A be such that L(A) = L_1;
          for n = 2, to ∞
step 2:     if O(L(A · C_n)) ⊆ L(A) then return A;
step 3:     modify A such that L(A_new) = L(A_old) ∪ O(L(A_old · C_n));
            /* assume all strings in ⋃_{n=3}^{∞} (S^{2n})* are enumerated: s_2, s_3, ... */
step 4:     if s_n satisfies C1–C12 for some α then
step 5:       modify A such that L(A_new) = L(A_old) ∪ {O(α^j(s)) | j ≥ 0};
            end if
          end for
        end
```

Figure 7.10: A procedure for finding a tight invariant.

and Kozen [AK86] have shown that in general the verification of iterative systems is undecidable, so researchers have focused on defining restrictive settings which can be completely automated, or defining more general approaches which depend on user interaction.

Browne, Clarke and Grumberg [BCG89], and Shtadler and Grumberg [SG90] have studied conditions under which the satisfaction of formulas of certain temporal logics is independent of the size of the system. In [SG90] the conditions seem to be quite restrictive, while in [BCG89] the conditions cannot in general be checked automatically. Wolper and Lovinfosse [WL90] have studied formal verification of iterative systems generated by interconnecting identical processes in certain regular fashion. They also present some decidability results for related problems. Kurshan and McMillan [KM89] present slightly more general results which can be applied both to process algebra and automata-based approaches. In both cases, automatic tools are used only to verify that a finite state-system suggested by the user is indeed an invariant. Kurshan and McMillan have hinted that it might be a good idea to check whether a system of some fixed size serves as an invariant. This idea was further developed by Rho and Somenzi [RS92, RS93], who have studied different network topologies and presented several sufficient conditions for the existence of such an invariant.

Our approach builds on results in [KM89, WL90] in a sense that it provides automatic support for generating invariants for a larger class of systems than in [RS92, RS93]. This work can be naturally extended in several ways. From the theoretical point of view, the decidability of existence of a finite invariant for open iterative systems needs to be studied. From the practical point of view, it would be useful to generalize the conditions for non-existence. This can be done by analyzing sequences of strings where not only a single element, but a whole substring is repeated many times. It is also possible to construct cases where the non-existence can be proved by analyzing a sequence of sets of string, rather then just a sequence of strings. Finally, in order to verify liveness properties, these results need to be extended to the automata on infinite sequences.

Chapter 8

Conclusions and future work

In this work we have presented several approaches to automatic abstraction of both finite-state and real-time systems. Common to all approaches is that a useful abstraction is obtained only after several iterations. In every iterations we modify an existing simplification to overcome its shortcomings. The power of a particular algorithm depends on the sophistication of this modification as well as in choosing an initial simplifications.

Overall, the proposed techniques have shown some encouraging experimental results, but they still leave a lot to be desired in term of efficiency, the size of systems they can handle, and particularly robustness. We have showed that user guidance can be used to avoid some of these problems. The down side is that the benefits of formal verification might be offset by the cost of time and effort a designer needs to invest in the verification process. It has been argued many times that only fully automatic verification has a chance of being widely accepted in the design community. We believe that a methodology that falls somewhat short of this ideal still has a chance of being accepted, provided it satisfies the following:

- Users must have a choice of the amount of interaction they want to invest (and receive appropriate efficiency improvements in return). In particular, fully automatic verification must be available for simple systems.

- Additional information that users supply must be an argument on why the system is believed to be correct, and it must be as much as possible independent of the verification tool. It is reasonable to expect of designers to provide reasoning behind their design. It is not reasonable to expect them to understand intricacies of a verification

algorithm.

We believe that iterative approaches similar to ours can be very effective, particularly at a high level of abstraction. At a lower level one has to deal with thousands, if not millions of transitions, and it is often hard to decide which are or are not essential for a given property. This may be easier to guess at a higher level where one deals with fewer, but more complex objects as counters, tokens, integers, buffers, or pipelines, to name a few. This naturally leads to application-specific methods, as high-level objects and typical properties are quite different for different application areas, for example microprocessors and communication protocols. We are already witnessing some contributions in this directions (e.g. [BD94, Bea93, HMAF94, MPS92]), and we can only expect this trend to grow stronger in the future.

There are many other possible approaches to increase the practical value of formal verification. Let us just shortly describe two possibilities that have received virtually no attention in the research community.

The first possibility follows directly from the fact that in practice formal verification is used as a debugging tool. Its real value is in finding bugs. The positive answer certainly increases the level of confidence in the design, but it is by no means the proof that a system will operate correctly, because too many thing can go wrong: the model may not reflect the behavior of the system accurately enough, the set of properties to be verified may not describe all the requirements on the system, and last but not least a verification tool might be faulty. As Vardi [Var94] summed it nicely: *"Verification is successful only if it fails"*. From this point of view, conservative simplifications produce an additional burden of verifying that a reported bug is due to original system and not due to over-simplification. Of more practical value would be their dual, which we call *liberal simplifications*. Liberal simplification preserve the behavior of the original system insofar that if the simplification violates the property, so thus the original system. But, if a liberal simplification is verified, the original system may or may not satisfy the property. The theory of liberal simplifications could probably be derived easily by mirroring the theory of conservative simplifications, but heuristics for finding tractable liberal simplification that are still able to discover some bugs are presently completely unexplored.

Ideally, one could combine liberal and conservative approaches to obtained a sequence of increasingly better approximations of the behavior of a system. Information

obtained by a conservative approximation (e.g. a superset of reachable states) could then be used to simplify a liberal approximation, and vise versa.

The second important practical problem verifying that the properties to be verified indeed describe desired behavior. In a strict sense this problem cannot be solved because one can only prove relations between two formal specification, thus at some point the "desired behavior" needs to be formalized through an error-prone informal process that cannot be verified. But some "sanity checks" on a set of properties can be defined. For example, if a property is a validity (i.e. true of every system) it obviously does not specify anything useful. To take this a step further, if a satisfaction of set of properties does not change even when the system is significantly changed, that might be an indication that the behavior is not precisely specified. This is similar to the notion of *fault coverage* in hardware testing. The fault coverage of a given set of test vectors is roughly a percentage of the design that cannot be changed without changing the response to those test vectors. To define a reasonable notion of a "fault coverage" for a set formally specified properties, and to find efficient algorithms of computing it is another completely unexplored research area that is deemed important in the design community.

In conclusion, formal verification promises to be a valuable design tool. There are still a lot of obstacles to its wide acceptance, but there is little doubt in our mind that those problems will be solved, and that formal verification will exit its niche and become a part of mainstream design methodologies. We hope that research presented in this work is a step toward this bright future.

Bibliography

[AB93] Adnan Aziz and Robert K. Brayton. Verifying interacting finite state machines. Technical Report UCB/ERL M93/52, Electronics Research Laboratory, College of Engineering, Univer sity of California, Berkeley, July 1993.

[ABB+94] A. Aziz, F. Balarin, R. K. Brayton, S.-T. Cheng, R. Hojati, S. C. Krishnan, R. K. Ranjan, A. L. Sangiovanni-Vincentelli, T. R. Shiple, V. Singhal, S. Tasiran, and H.-Y. Wang. HSIS: A BDD-based environment for formal verification. In *Proceedings of the 31th ACM/IEEE Design Automation Conference*, 1994.

[ACD90] Rajeev Alur, Costas Courcoubetis, and David L. Dill. Model-checking for real-time systems. In *Proceedings of 5th LICS*, pages 414–425, June 1990.

[ACD+92] Rajeev Alur, Costas Courcoubetis, David L. Dill, Nicholas Halbwachs, and Howard Wong-Toi. An implementation of three algorithms for timing verification based on automata emptiness. In *Proceedings of IEEE Real-time Systems Symposium*, 1992.

[ACH+92] Rajeev Alur, Costas Courcoubetis, Nicholas Halbwachs, David L. Dill, and Howard Wong-Toi. Minimization of timed transition systems. In *Proceedings of CONCUR'92*, August 1992.

[ACHH93] Rajeev Alur, Costas Courcoubetis, Thomas A. Henzinger, and Pei-Hsin Ho. Hybrid automata: an algorithmic approach to the specification and verification of hybrid systems. In R. L. Grossman, A. Nerode, A. P. Ravn, and H. Rischel, editors, *Hybrid Systems, Lyngby, Denmark, 10-12 Oct. 1991, Proceedings*, pages 209–229. Springer-Verlag, 1993.

[AD90] Rajeev Alur and David L. Dill. Automata for modelling real-time systems. In M.S. Paterson, editor, *ICALP'90 Automata, languages, and programming: 17th international colloquium.* Springer-Verlag, 1990. LNCS vol. 443.

[AH92] Rajeev Alur and Thomas A. Henzinger. Logics and models of real time: a survey. In J.W. de Bakker, C. Huizing, W.P de Roever, and G. Rozenberg, editors, *Proc. Real-Time: Theory in Practice, REX Workshop Proceedings, Mook, Netherlands, 3-7 June 1991*, pages 74–106. Springer-Verlag, 1992.

[AIKY93] Rajeev Alur, Alon Itai, R. P. Kurshan, and M. Yannakakis. Timing verification by successive approximation. In G. v. Bochmann and D.K. Probst, editors, *Proceedings of Computer Aided Verification : 4th International Workshop, CAV '92, Montreal, Canada, June 29-July 1, 1992.* Springer-Verlag, 1993. LNCS vol. 663.

[AK86] K.R. Apt and D.C. Kozen. Limits for automatic verification of finite-state concurrent systems. *Information Processing Letters*, 22(6):307–309, May 1986.

[AL91] Martin Abadi and Leslie Lamport. An old-fashioned recipe for real time. In J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, editors, *Real-Time: Theory in Practice*, volume 600 of *LNCS*. Springer-Verlag, 1991. Proceedings of the REX Workshop Mook, The Netherlands.

[ALG+91] L. M. Augustin, D. C. Luckman, B. A. Gennart, Y. Huh, and A. G. Stanculescu. *Hardware design and simulation in VAL/VHDL.* Kluver Academic Publishers, 1991.

[Alu91] Rajeev Alur. *Techniques for Automatic Verification of Real-time Systems.* PhD thesis, Stanford University, 1991.

[ASB+94] Adnan Aziz, Vigyan Singhal, Felice Balarin, Alberto L. Sangiovanni-Vincentelli, and Robert K. Brayton. Equivalences for fair kripke structures. In *Proceedings of ICALP'94*, 1994.

[ASSSV94] Adnan Aziz, Thomas R. Shiple, Vigyan Singhal, and Alberto L. Sangiovanni-Vincentelli. Formula-dependent equivalence for compositional CTL model checking. In David L. Dill, editor, *Proceedings of Computer Aided Verification:*

6th International Conference, CAV'94, Stanford, CA, June 1994, pages 324–337. Springer-Verlag, 1994. LNCS vol. 818.

[BBC+] Felice Balarin, Robert K. Brayton, Szu-Tse Cheng, Desmond A. Kirkpatrick, Alberto L. Sangiovanni-Vincentelli, and Ephrem C. Wu. A methodology for formal verification of real-time systems. unpublisehd.

[BCG89] M.C. Browne, E.M. Clarke, and O. Grumberg. Reasoning about networks with many identical finite state processes. *Information and Computation*, 81(1):13–31, 1989.

[BD94] Jerry R. Burch and David L. Dill. Automatic verification of pipelined microprocessor control. In David L. Dill, editor, *Proceedings of Computer Aided Verification: 6th International Conference, CAV'94, Stanford, CA, June 1994*, pages 68–80. Springer-Verlag, 1994. LNCS vol. 818.

[Bea93] Derek L. Beatty. *A Methodology for Formal Hradware Verification with Application to Microprocessors.* PhD thesis, School of Computer Science, Carnegie Mellon University, August 1993.

[BFH90] A. Bouajjani, J-C. Fernandez, and N. Halbwachs. Minimal model generation. In Edmund M. Clarke and Robert P. Kurshan, editors, *Proceedings of the Workshop on Computer-Aided Verification*, volume 531 of *LNCS*, pages 197–203. Springer-Verlag, June 1990.

[BRB90] Karl S. Brace, Richard L. Rudell, and Randal E. Bryant. Efficient implementation of a BDD package. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pages 40–45, June 1990.

[BSV93] Felice Balarin and Alberto L. Sangiovanni-Vincentelli. A verification strategy for timing constrained systems. In G. v. Bochmann and D.K. Probst, editors, *Proceedings of Computer Aided Verification : 4th International Workshop, CAV '92, Montreal, Canada, June 29-July 1, 1992*, pages 151–63. Springer-Verlag, 1993. LNCS vol. 663.

[Büc60] J.R. Büchi. On a decision method in restricted second order arithmetic. In E. Nagel, editor, *Proceedings of the 1960 International Congress on Logic,*

Methodology and Philosophy of Science, pages 1–12. Stanford University Press, 1960.

[CDCT93] Costas Courcoubetis, David L. Dill, M. Chatzaki, and Panagiotis Tzounakis. Verification with real-time COSPAN. In G. v. Bochmann and D.K. Probst, editors, *Proceedings of Computer Aided Verification : 4th International Workshop, CAV '92, Montreal, Canada, June 29-July 1, 1992.* Springer-Verlag, 1993. LNCS vol. 663.

[CDK89] Edmund M. Clarke, I. A. Draghiescu, and R. P. Kurshan. A unified approach for showing language containment and equivalence between various type of ω-automata. Technical report, Carnegie Mellon University, 1989.

[CES86] Edmund 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 Systems*, 2(8):244–263, 1986.

[CGH+93] Massimiliano Chiodo, Paolo Giusto, Harry Hsieh, Attila Jurecska, Luciano Lavagno, and Alberto Sangiovanni-Vincentelli. A formal specification model for hardware/software codesign. In *Proceedings of the International Workshop on Hardware-Software Codesign*, 1993.

[CGL92] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. In *Proc. Principles of Programming Languages*, January 1992.

[CLM89] Edmund M. Clarke, David E. Long, and Kenneth L. McMillan. Compositional model checking. In *4th Annual Symposium on Logic in Computer Science*, Asilomar, CA, June 1989.

[CYF94] B. Chen, M. Yamazaki, and M. Fujita. Bug identification of a real chip design by symbolic model checking. In *Proceedings of The European Conference on Design Automation, EDAC-ETC-EUROASIC, Paris, France, 28 Feb.-3 March 1994*, pages 132–136. IEEE Comput. Soc. Press, 1994.

[dBHdRR91] J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, editors. *Real-Time: Theory in Practice*, volume 600 of *LNCS*. Springer-Verlag, 1991. Proceedings of the REX Workshop Mook, The Netherlands.

[DGG93] Dennis Dams, Orna Grumberg, and Rob Gerth. Generation of reduced models for checking fragments of CTL. In Costas Courcoubetis, editor, *Proceedings of the Conference on Computer-Aided Verification*, volume 697 of *LNCS*, pages 479–490. Springer-Verlag, June 1993.

[Dil89] David L. Dill. Timing assumptions and verification of finite-state concurrent systems. In Joseph Sifakis, editor, *Automatic Verification Methods for Finite-State Systems*. Springer-Verlag, 1989. LNCS vol. 407.

[EL86] E. Allan Emerson and C. L. Lei. Efficient model checking in fragments of the propositional mu-calculus (extended abstract). In *Proceedings, Symposium on Logic in Computer Science*, pages 267–278, Cambridge, Massachusetts, 16–18 June 1986. IEEE Computer Society.

[Eme90] E. Allan Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 995–1072. Elsvier Science Publishers B. V., 1990.

[GL91] Orna Grumberg and David E. Long. Model checking and modular verification. In *CONCUR, Amsterdam, The Netherlands*, August 1991. (also submitted to JACM).

[GL93] S. Graf and C. Loiseaux. A tool for symbolic program verification and abstraction. In Costas Courcoubetis, editor, *Computer Aided Verification: 5th International Conference, CAV'93, Elounda, Greece, June/July 1993, Proceedings*, pages 71–84. Springer-Verlag, 1993. LNCS vol. 697.

[Hal93] Nicholas Halbwachs. Delay analysis in synchronous programs. In Costas Courcoubetis, editor, *Computer Aided Verification: 5th International Conference, CAV'93, Elounda, Greece, June/July 1993, Proceedings*, pages 333–346. Springer-Verlag, 1993. LNCS vol. 697.

[HB95] Ramin Hojati and Robert K. Brayton. An environment for formal verification based on symbolic computation. *Formal Methods in System Design: An International Journal*, 5, 1995. (to appear).

[HBK93] Ramin Hojati, Robert K. Brayton, and R. P. Kurshan. BDD-based debugging of designs using language containment an fair CTL. In Costas Courcoubetis, editor, *Computer Aided Verification: 5th International Conference, CAV'93, Elounda, Greece, June/July 1993, Proceedings*, pages 41–58. Springer-Verlag, 1993. LNCS vol. 697.

[Hen61] Frederick C. Hennie. *Iterative Arrays of Logical Circuits.* MIT Press and John Eiley Sons, Inc., 1961.

[HK88] Z. Har'El and R. P. Kurshan. Software for analysis of coordination. In *Proceedings of the International Conference on System Science*, pages 382–385, 1988.

[HMAF94] Y.V. Hoskote, J. Moondanos, J.A. Abraham, and D.S. Fussell. Abstraction of data path registers for multilevel verification of large circuits. In *Proceedings of Fourth Great Lakes Symposium on VLSI, Design Automation of High Performance VLSI Systems GLSV '94*, pages 11–14. IEEE Computer Society Press, 1994.

[HNSY92] Thomas A. Henzinger, Xavier Nicolin, Jospeh Sifakis, and Sergio Yovine. Symbolic model-checking for real-time systems. In *Proceedings of 7th Symposium on Logics in Computer Science.* IEEE Computer Society Press, 1992.

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

[KM89] R. P. Kurshan and K. L. McMillan. A structural induction theorem for processes. In *Proceedings of the 8th ACM Symp. PODC*, 1989.

[Kur90] R. P. Kurshan. Analysis of discrete event coordination. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems : Models, Formalisms, Correctness*, pages 414–453. Springer-Verlag, 1990. LNCS vol. 430.

[Kur94] Robert P. Kurshan. *Formal Verification of Coordinating Processes: The Automata-Theoretic Approach.* Princeton University Press, 1994. To be pudlished.

[Law76] Eugene L. Lawler. *Combinatorial optimization : networks and matroids.* Holt, Rinehart and Winston, 1976.

[LY92] D. Lee and M. Yannakakis. Online minimization of transition systems. In *Proceedings of the Twenty-Fourth Annual ACM Symposium on the Theory of Computing, Victoria, BC, Canada, 4-6 May 1992*, pages 264–274. ACM, 1992.

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

[MP81] Zohar Manna and Anir Pnueli. Verification of concurrent programs: The temporal framework. In R. Boyer and J. Moore, editors, *Correctness Problem in Computer Science*, pages 215–273. Academic Press, 1981.

[MPS92] Enrico Macii, Bernard Plessier, and Fabio Somenzi. Verification of systems containing counters. In *Digest of Technical Papers of the 1992 IEEE International Conference on CAD*, pages 179–182. IEEE Comput. Soc. Press, 1992.

[MV94] Jennifer McManis and Pravin Varaiya. Suspension automata: A decidable class of hybrid automata. In David L. Dill, editor, *Proceedings of Computer Aided Verification: 6th International Conference, CAV'94, Stanford, CA, June 1994*, pages 105–117. Springer-Verlag, 1994. LNCS vol. 818.

[NJK94] B. E. Nelson, R. B. Jones, and D. A. Kirkpatrick. Simulation event pattern checking with proto. In *Proceedings of the SHDL Conference*, 1994.

[NSY92] Xavier Nicolin, Jospeh Sifakis, and Sergio Yovine. Compiling real-time specifications into extended automata. *IEEE TSE Special Issue on Real-Time Systems*, September 1992.

[Pet93] K. Petty. The PATHO Operating System and User's Guide. Technical report, UC Berkeley, 1993.

[Pne92] Amir Pneuli. How vital is liveness? In W. R. Cleaveland, editor, *CONCUR'92: Proc. of the Third International Conference on Concurrency The-*

ory, Stony Brook, NY, USA, August 1992, pages 162–175. Springer-Verlag, 1992. LNCS vol. 630.

[Pnu77] A. Pnueli. The temporal logic of programs. In *18th IEEE Symposium on Foundations of Computer Science*, pages 46–57, October 1977.

[PV94] Anuj Puri and Pravin Varaiya. Verification of hybrid systems using timed abstractions. In *Proc. Workshop on Hybrid Systems*, October 1994.

[RM94] T.G. Rokicki and C.J. Myers. Automatic verification of timed circuits. In David L. Dill, editor, *Proceedings of Computer Aided Verification: 6th International Conference, CAV'94, Stanford, CA, June 1994*, pages 468–480. Springer-Verlag, 1994. LNCS vol. 818.

[RS92] J.K. Rho and F. Somenzi. Inductive verification of iterative systems. In *Proceedings of the 29th ACM/IEEE Design Automation Conference*, pages 628–33, June 1992.

[RS93] J.K. Rho and F. Somenzi. Automatic generation of network invariants for the verification of iterative sequential systems. In Costas Courcoubetis, editor, *Computer Aided Verification: 5th International Conference, CAV'93, Elounda, Greece, June/July 1993, Proceedings*, pages 123–137. Springer-Verlag, 1993. LNCS vol. 697.

[SC85] Aravinda P. Sistla and Edmund M. Clarke. Complexity of propositional linear temporal logics. *Journal of the ACM*, 32(3):733–749, July 1985.

[SCSVB92] Thomas R. Shiple, Massimiliano Chiodo, Alberto L. Sangiovanni-Vincentelli, and Robert K. Brayton. Automatic reduction in CTL compositional model checking. In *Proc. Fourth Workshop on Computer-Aided Verification*, pages 225–238, Montreal, June 1992. Also appeared in Lecture Notes in Computer Science, vol. 663.

[SD93] R. Schlör and W. Damm. Specification and verification of system level hardware designs using timing diagrams. In *Proceedings of The European Conference on Design Automation, Paris, France, February 1993*, 1993.

[SG90] Z. Shtadler and O. Grumberg. Network grammars, communication behaviors and automatic verification. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems, International Workshop Proceedings, Grenoble, France, 12-14 June 1989*, pages 151–65. Springer-Verlag, 1990. LNCS vol. 407.

[Str82] R. S. Streett. Propositional dynamic logic of looping and converse is elementarily decidable. *Information and Control*, 54(no.1-2):121–141, July-Aug. 1982.

[SV89] Shmuel Safra and Moshe Y. Vardi. On ω-automata and temporal logic. In *Proceedings of the Twenty First Annual ACM Symposium on Theory of Computing, Seattle, WA, USA, 15-17 May 1989*, pages 127–137, 1989.

[Tar83] Robert Endre Tarjan. *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

[Tho90] W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 133–191. Elsvier Science Publishers B. V., 1990.

[Var93] Pravin Varaiya. Smart cars on smart roads: Problems of control. *IEEE Trans. on Automatic Control*, 38(2):195–207, February 1993.

[Var94] Moshe Vardi, 1994. A remark during the presentation at Computer Aided Verification: 6th International Conference, CAV'94, Stanford, CA, June 1994.

[VW86] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification. In *Proc. IEEE Symp. on Logic in Computer Science*, pages 332–344, Boston, July 1986.

[VWS83] Moshe Y. Vardi, Pierre Wolper, and Aravinda P. Sistla. Reasoning about infinite computation paths. In *Proceedings of 24th Annual Symposium on Foundations of Computer Science, Tucson, AZ, USA, 7-9 Nov. 1983.* IEEE Comput. Soc. Press, 1983.

[Vyt91] J. Vytopil, editor. *Formal Techiques in Real-Time and Fault-Tolerant Systems*, volume 571 of *LNCS*. Springer-Verlag, 1991. Proceedings of the Second International Symposium, Nijmegen, The Netherlands.

[WL90] P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems, International Workshop Proceedings, Grenoble, France, 12-14 June 1989*, pages 68–80. Springer-Verlag, 1990. LNCS vol. 407.

[YL93] M. Yannakakis and D. Lee. An efficient algorithm for minimizing real-time transitions systems. In Costas Courcoubetis, editor, *Computer Aided Verification: 5th International Conference, CAV'93, Elounda, Greece, June/July 1993, Proceedings*, pages 210–224. Springer-Verlag, 1993. LNCS vol. 697.

[Yov92] Sergio Yovine, 1992. private communication.

[YSSC93] T. Yoneda, A. Shibayama, B.-H. Schlingloff, and E.M. Clarke. Efficient verification of parallel real-time systems. In Costas Courcoubetis, editor, *Computer Aided Verification: 5th International Conference, CAV'93, Elounda, Greece, June/July 1993, Proceedings*, pages 321–332. Springer-Verlag, 1993. LNCS vol. 697.