Ameliorating the State Explosion Problem

by

Kedar Sharad Namjoshi, M.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

May 1998

Ameliorating the State Explosion Problem

Approved by
Dissertation Committee:

_____

_____

_____

_____

_____

To my parents

To Chitra

Acknowledgments

My development as a person and as a scientist has been influenced by many people. I would like to acknowledge them here, and offer thanks for their help and for what I have learnt from them.

My advisor, Professor Allen Emerson, has had a major role in shaping this dissertation. From him, I have learnt most of all to be careful in my research; to formulate definitions and concepts precisely and clearly. He has also been a great friend and mentor, and I have enjoyed our discussions very much, on both technical and non-technical matters. Professor Jayadev Misra's class introduced me to program verification, and his exposition of the subject, and insistence on simplicity and elegance in writing has made a lasting impression. I have learnt to appreciate the fine points of good notation, writing style, and clarity of exposition from Professor Edsger W. Dijkstra. I would like to thank him for inviting me to attend the Austin Tuesday Afternoon Club meetings, from which I have learnt a great deal that will always influence my work. Professors Vijaya Ramachandran, Donald Fussell and Jacob Abraham graciously agreed to be on my dissertation committee. Discussions with them have always served to broaden my perspective. I have particularly enjoyed taking courses on mathematical logic with Professors Vladimir Lifschitz and J. Moore.

I have learnt a lot from discussions with my fellow graduate students Peter Manolios, Carlos Puchol, Robert Sumners, Richard Trefler, Rajeev Joshi, Ashis

Tarafdar and Rajmohan Rajaraman. Long discussions on technical matters, the nature of computer science and the meaning of life have always been enjoyable and illuminating! Jacob Kornerup, G. Neelakantan Kartha, Sanjay Kaluskar, Lyn Pierce, Markus Kaltenbach, Emilio Remolina, Jun Sawada, Nina Amla, Vasilis Samoladas, Emilio Camahort, Yannis Smaragdakis, V. Balayoghan, Sowmya Ramachandran and Ajita John are all good friends and great to be with! My stay in Austin has been enjoyable largely due to my friends here : Sudheer Koneru, Abraham Prasad, Diwakar Vishakhadatta, Venkatesh Sreekantan (Pax), Venkatesh Murty, Ajit and Manju Dingankar and Anand Ramachandran.

My sisters, Neelam and Nirupa, and my brothers-in-law, Rajiv and Mahesh, have helped and encouraged me throughout my education. My parents encouraged me in my early inclination towards science and literature, and have supported and guided me over the years while giving me a lot of freedom in making my own decisions. From my wife, Chitra, I have received constant love, support and encouragement. It is difficult and probably a bit silly to thank your family formally; they provide something that goes beyond what can be "thanked". It is for this reason that this dissertation is dedicated to Chitra and to my parents.

Kedar Sharad Namjoshi

The University of Texas at Austin
May 1998

Ameliorating the State Explosion Problem

Publication No. _____

Kedar Sharad Namjoshi, Ph.D.
The University of Texas at Austin, 1998

Supervisor: E. Allen Emerson

Systems that maintain an ongoing interaction with their environment, such as Operating Systems Network Protocols and Microcontrollers, are commonplace. The complexity of these systems necessitates a rigorous verification of correct behavior. Automatic verification methods such as *Model Checking*, while theoretically efficient, suffer in practice from the large state space of these systems, a phenomenon called *State Explosion*. State explosion often arises when verifying systems parameterized by the number of component processes, and single systems with large data domains. The main contribution of this dissertation is in the development of abstraction methods that serve to ameliorate the state explosion problem for such systems.

The first part of the dissertation presents abstractions for interesting classes of parameterized systems that reduce the *infinite* family of instances to a *finite-state* system, while *exactly* preserving correctness properties. For parameterized ring systems with a synchronizing token, it suffices to examine a few small instances in order to determine the correctness of *every* instance of the system. The method is applicable to protocols such as mutual exclusion and Milner's Cycler. Somewhat surprisingly, the verification problem is undecidable even if the token carries a single

bit of information. For parameterized synchronous systems, an exact abstraction reduces the parameterized system to a finite "abstract" graph. This abstraction method is applied to the verification of the SAE-J1850 industrial standard bus arbitration protocol. We also present a general algorithm schema from which algorithms for model-checking several types of infinite-state systems can be derived.

The second part of the dissertation presents a proof technique for showing that two programs are equivalent up to "stuttering" (repetition) of states. Stuttering arises when comparing programs that are at different levels of abstraction. The new formulation replaces the *global* reasoning of earlier techniques with *local* reasoning, which considerably simplifies abstraction proofs. This new formulation is used in conjunction with a theorem prover to verify a data abstraction for the alternating-bit protocol.

Contents

Chapter 1

Introduction

"What is now proved was once only imagin'd"

– William Blake.

The Marriage of Heaven and Hell, 1790-93.

Systems that maintain an ongoing interaction with their environment are commonplace. Examples of such computer systems include Operating Systems, Network Protocols and micro-controllers. In a landmark paper [Pnueli 77], Pnueli argued that Temporal Logic is the appropriate formalism for reasoning about the correctness of such "reactive" systems. At about the same time, Hoare [Hoare 78] and Milner [Milner 80] introduced the process calculi *CSP* and *CCS*, which provide an alternative means of specifying and verifying such systems.

The essential difference between reactive systems and the class of sequential terminating programs is the property of non-termination. While non-termination of a sequential program usually indicates an error it is the essential feature of a reactive system, which enables the system to maintain a continuous interaction with its environment. Thus, the input-output semantics of terminating programs is inappropriate for reasoning about reactive systems. Methods for reasoning about

reactive systems thus rely on infinite objects, such as computations or computation trees, to define the semantics of these systems.

Proof systems for demonstrating correctness of reactive systems are presented for temporal logic in [CM 88, MP 92] and for the process calculus view in [Hoare 85, Milner 90]. While proof systems have the advantages of generality in specification and flexibility in proof, the complexity of proofs for a large system makes it difficult to use this method. Automated theorem provers (cf. [BM 79, HS 96]) may be used to assist with simplification and proof management, but it still requires substantial human effort to carry out a proof with a mechanical theorem prover.

Model Checking [CE 81] arose from the insight that many interesting systems have a finite number of states, hence the truth of a temporal correctness formula over such a system can be determined by a recursive graph search procedure based on the structure of the formula. In the past decade, Model Checking has been applied to the verification of several circuit designs and protocols [BCMDH 90, McMillan 92]. Correctness properties are typically expressed in the branching temporal logic CTL, for which the model-checking algorithm has time complexity linear in the size of the structure and the formula (linear temporal logics have model-checking time complexity that is exponential in the formula size [SC 85, LP 85]). In the process calculus framework, algorithms have been developed for checking various types of similarity between finite-state process [KS 90, PT 87, GV 90].

An interesting and useful feature of most model-checking algorithms is the generation of counter-examples if the desired property does not hold of the system. This has led to widespread use of Model Checking as a verification method. Although model-checking algorithms have time complexity that is linear in the size of the structure, the size of the structure (i.e., the number of states) may itself be exponential in the size of its description as a program. For instance, a program with n Boolean variables may have a reachable state space of size 2^n. This phenomenon,

referred to as *State Explosion*, is the main obstacle to the application of Model Checking and other automatic verification methods. Currently, the best model-checker implementations can handle programs with at most a few hundred Boolean variables. In addition, the restriction to systems with a finite number of states excludes a number of interesting types of systems, such as distributed protocols, from the purview of Model Checking.

This dissertation presents approaches towards solving both problems. The first part of the dissertation deals with the problem of verifying, fully automatically, parameterized protocols and other types of infinite-state systems. The second part of the dissertation presents a new formulation of similarity between systems, which is used to reduce large structures to similar small ones. The techniques applied in both cases reduce large structures to smaller ones while preserving a number of properties of interest. The contributions of this dissertation are discussed in detail in the following section.

1.1 Overview of the Dissertation

A large part of this dissertation is concerned with determining conditions under which the verification problem for parameterized systems is decidable. Model Checking, being a procedure defined over finite-state systems, cannot be used directly to verify parameterized protocols. Most protocols for distributed systems, multiprocessors and computer networks are, however, parameterized by the number of processes taking part in the protocol and correctness is desired for every instance of the protocol. Thus, a large and interesting class of problems is excluded from the purview of Model Checking. The current practice is to use Model Checking to determine correctness of a few instances of a parameterized protocol. This approach has a strong similarity to testing, and all of the disadvantages that go with it. On the other hand, automated verification of a parameterized system is undecidable

in general [AK 86]. One is thus faced with a dilemma: it seems desirable to have an automated procedure for verifying complex parameterized protocols, but such a procedure cannot exist in general. The approach put forward in this dissertation is to identify classes of parameterized systems for which the verification problem is decidable. Algorithms for two such classes are presented in Chapters 3 and 4. These algorithms have been used to verify protocols for mutual exclusion, Milner's Cycler protocol, and the SAE-J1850 industry standard bus protocol, whose verification is described in Chapter 5.

While the systems considered are quite different, the algorithms developed for parameterized verification have a strong similarity with other algorithms for the model-checking of infinite-state systems. This similarity is explored in Chapter 6, where a general procedure for the verification of infinite-state systems is proposed and analyzed.

Besides parameterized protocols, a common source of state explosion is systems that have a large data domain. It is often the case that the control flow of the system is largely independent of the data. Such a "data insensitivity" property for a particular system is shown by demonstrating that a particular equivalence relation between data values is a bisimulation. In Chapter 7, we present a new definition of bisimulation under stuttering of states, a concept that is often needed for proofs of data insensitivity, and in equivalence proofs between systems at different levels of abstraction. The new definition replaces existing global proof methods with a local proof rule, which simplifies proofs considerably and facilitates the use of mechanical theorem provers in automating such proofs.

In the rest of this section we discuss in some detail the main contributions of the dissertation and their relevance to the general problem of ameliorating state explosion. These results have been published in [EN 95], [EN 96], [Namjoshi 97], [EN 98] and [EN 98a].

1.1.1 Parameterized Token Rings

Concurrent processes are often connected in a unidirectional ring structure, where processes communicate by passing a token with associated information. Usually, correctness properties are expected to hold independent of the size of the ring. We show that if the processes use the token only as a signal, the task of checking many important types of correctness properties for rings of *all* sizes can be reduced to checking these properties for rings of sizes at most a *small* cutoff. These results are applied to automatically verify all instances of a mutual exclusion protocol and Milner's Cycler protocol. We also show that the task of checking correctness of rings of all sizes is undecidable even if the token carries a single bit of information. This delineates rather sharply the boundary between decidability and undecidability of automatic verification of token-ring systems, while also providing a simpler proof of the undecidability of verification for parameterized systems. These results are presented in detail in Chapter 3 and first appeared in [EN 95].

1.1.2 Parameterized Synchronous Systems

We consider here a parameterized system formed of a single *control* process and an arbitrary number of *user* processes. Both processes may test the control process state and the presence of user processes occupying a specific local state. We show that the task of checking properties of such a parameterized system is *decidable* for the synchronous computation model, but undecidable, even for invariance properties, for the interleaving model of computation.

The algorithm for the synchronous model constructs an exact finite-state representation of all instances of the system and performs a special model-checking procedure on it. This procedure works in space polynomial in the size of the input, which is the sum of the sizes of the control and user process descriptions. We prove that this is optimal in a complexity-theoretic sense, by showing that the

verification problem is PSPACE-complete in general, even for invariance properties. These results are presented in Chapter 4.

This algorithm has been applied in the verification of a parameterized, industrial standard bus protocol (SAE-J1850). The SAE-J1850 protocol is an automobile industry standard protocol for data transmission between units connected by a single wire bus. Units may attempt transmission concurrently; the heart of the protocol is a distributed, on-the-fly arbitration mechanism that must ensure a unique result. We have modeled this protocol and verified its correctness for an *arbitrary* number of units, using the algorithms described above and additional abstraction results. This work is described in Chapter 5 and will appear in [EN 98a].

1.1.3 Model Checking Infinite-State Systems

A variety of systems, including parameterized protocols, real-time automata, communication protocols and Petri Nets generate an infinite state space. Although deciding if a general temporal logic formula holds of a system with an infinite state space is undecidable, decidability results have been obtained for many cases, especially for parameterized protocols and real-time automata. We present a general technique for model-checking linear temporal logic specifications for infinite-state systems, which is based on a generalization of the Karp-Miller construction for Petri Nets. Many known decidability results for the verification of infinite-state systems can be derived uniformly from this technique; this is an indication that the technique may be widely applicable.

We demonstrate the application of this general technique to the problem of verifying parameterized broadcast protocols, a class that includes many protocols for maintaining cache coherency in multiprocessor systems. A number of correctness properties of a parameterized invalidation-based cache coherency algorithm are established automatically by the application of this procedure. These results will

be published in [EN 98].

1.1.4 Abstraction by Quotienting

One of the most common situations that lead to state explosion is the verification of programs with large data domains. It is frequently the case that the control behavior of such a system is only weakly dependent on the actual data values. Demonstrating such a proposition usually involves showing that an equivalence relation on data values is a bisimulation. Such a relation is often a bisimulation only up to finite stuttering (repetition) of state propositions. Stuttering also arises in other contexts; especially when showing equivalence of two systems at different levels of abstraction, which may entail mapping a single step in one system to a sequence of steps in the other.

We present a simple new formulation of stuttering bisimulation, in terms of a ranking function over a well-founded set. It has the pleasant property that, like strong bisimulation [Milner 90], it requires checking only *single* transitions. This *local* property of the definition, which is in strong contrast to earlier *global* definitions, leads to shorter and simpler proofs of bisimulation. In addition, it facilitates the use of automated theorem provers in demonstrating the correctness of such bisimulation proofs. In Chapter 7, we present this new formulation, and show its equivalence to an earlier formulation of [BCG 88]. We also present some examples that demonstrate its utility for proving equivalence between systems, and for data abstraction. These results were first presented in [Namjoshi 97].

Chapter 2

Preliminaries

This chapter contains definitions of various concepts that are used throughout this thesis. They include the syntax and semantics of various temporal logics, automata on infinite structures, and structural correspondences such as bisimulation and simulation. These definitions are modified as necessary in subsequent chapters. We also give a short description of two approaches to Model Checking : by fixpoint evaluation [CE 81, EL 86], and using automata theory [VW 86].

2.1 Notation

Quantified expressions are written in the format $(\mathbf{Q}x : r : p)$, where $\mathbf{Q}$ is the quantifier (e.g., $\exists, \forall, min, max$), x the bound variable, r the range, and p the expression being quantified [DS 90]. When the range of x is clear from the context, it may be dropped, in which case the quantified expression has the form $(\mathbf{Q}x :: p)$. Sets defined by set comprehension are written in the format $\{x|P(x)\}$, which represents the set of all elements x for which the predicate P holds. The powerset of a set S is denoted by $\mathcal{P}(S)$. For a binary relation R, we often write $s\ R\ t$ for readability in place of $(s,t) \in R$.

Proofs are often given in the calculational style [DS 90]. A proof in this format is constructed out of the following basic unit :

$$\begin{array}{l} \quad P \\ op\,(\ \text{hint justifying } P\ op\ Q\ ) \\ \quad Q \end{array}$$

The binary operator op is a transitive relation on the expressions, which allows the chaining together (using transitivity) of a number of such proof units. Typical operators are $\Rightarrow$ (implies), $\Leftarrow$ (follows-from), iff (if-and-only-if), and $\leq$.

2.2 Labeled Transition Systems

The types of structures over which temporal logics are interpreted are called Labeled Transition Systems (LTS's) [Keller 76]. Informally, a LTS is a directed graph, where each vertex (a state) is labeled with the atomic propositions true at that state, and each edge (an action) is labeled with an action symbol.

Definition 2.1 (LTS) *A Labeled Transition System (or LTS) A is a structure $(Q, \Sigma, \delta, \lambda, L, I)$ where*

- *Q is a non-empty set of states,*
- *Σ is a transition (or action) alphabet, possibly containing the distinguished symbol τ (the silent action),*
- *$\delta \subseteq Q \times \Sigma \times Q$ is the transition relation,*
- *$\lambda : Q \rightarrow L$ is a labelling function on the states, with the condition that if $(s, \tau, t) \in \delta$ then $\lambda(s) = \lambda(t)$.*
- *L is a non-empty set of labels,*

- $I \subseteq Q$ *is the set of initial states.*

For clarity, we write $s \xrightarrow{a}_A t$ instead of $(s, a, t) \in \delta$. We write $s \xrightarrow{A} t$ for $(\exists a : a \in \Sigma : s \xrightarrow{a}_A t)$. The subscript A is often elided when a specific structure is under consideration. A *path* σ of A (with $length(\sigma)$ states) is a (finite or infinite) sequence of states such that for every i such that $i + 1 < length(\sigma)$, $\sigma_i \xrightarrow{A} \sigma_{i+1}$. A *fullpath* is a maximal path; i.e., either it is infinite or the last state has no successor. A *computation* is a path that starts at an initial state of A.

The semantics of most temporal logics do not take edge labels into account. They are defined over a special kind of LTS known as a Kripke structure.

Definition 2.2 (Kripke Structure) *A Kripke Structure is defined by a structure* $(Q, \delta, \lambda, AP, I)$*, where*

- Q *is a non-empty set of states,*
- $\delta \subseteq Q \times Q$ *is the transition relation,*
- $\lambda : Q \to \mathcal{P}(AP)$ *is a labelling function on the states,*
- AP *is a non-empty set of* atomic *propositions,*
- $I \subseteq Q$ *is the set of initial states.*

Sometimes, fairness constraints are added to an LTS. We consider the simplest form of a fair LTS.

Definition 2.3 (Fair LTS) *A fair LTS is a structure* (A, F) *where* A *is a LTS and* F *is a subset of the states of* A*. A computation of a fair LTS is a computation of* A *where a state from* F *appears infinitely often.*

2.2.1 LTS Composition

The parallel composition of two LTS's $A = (Q_A, \Sigma_A, \delta_A, \lambda_A, L_A, I_A)$ and $B = (Q_B, \Sigma_B, \delta_B, \lambda_B, L_B, I_B)$, such that $L_A \cap L_B = \emptyset$, is defined with respect to a *synchronizer* structure $\Gamma = (\Gamma_A, \Gamma_B, \bar{\ })$, where $\Gamma_A \subseteq \Sigma_A \backslash \{\tau\}$, $\Gamma_B \subseteq \Sigma_B \backslash \{\tau\}$, and $\bar{\ } : \Gamma_A \to \Gamma_B$ is a bijection. Informally, the sets in the synchronizer represent the actions that the processes synchronize on. The composition, denoted by $A \parallel_\Gamma B$, defines the LTS $AB = (Q_{AB}, \Sigma_{AB}, \delta_{AB}, \lambda_{AB}, L_{AB}, I_{AB})$:

1. $Q_{AB} = Q_A \times Q_B$,
2. $\Sigma_{AB} = (\Sigma_A \cup \Sigma_B) \backslash (\Gamma_A \cup \Gamma_B)$,
3. δ_{AB} is defined by :

   $(s,t) \stackrel{a}{\to}_{AB} (u,v)$ iff

   (a) $a \notin \Gamma_A \wedge s \stackrel{a}{\to}_A u \wedge t = v$ (a move by A only), or

   (b) $a \notin \Gamma_B \wedge t \stackrel{a}{\to}_B v \wedge s = u$ (a move by B only), or

   (c) $a = \tau \wedge (\exists b :: s \stackrel{b}{\to}_A u \wedge t \stackrel{\bar{b}}{\to}_B v)$ (a synchronized move)
4. $\lambda_{AB}(s,t) = \lambda_A(s) \cup \lambda_B(t)$,
5. $L_{AB} = L_A \cup L_B$, and
6. $q^0_{AB} = I_A \times I_B$.

It is easily shown that $\parallel$ is associative. It is also symmetric up to strong bisimulation.

2.2.2 LTS Projection

For two LTS's $A = (Q_A, \Sigma_A, \delta_A, \lambda_A, L_A, I_A)$ and $B = (Q_B, \Sigma_B, \delta_B, \lambda_B, L_B, I_B)$, composed to form $C = A \parallel B$, the projection of C on A is defined as the LTS $C|_A$ given by $C|_A = (Q, \Sigma, \delta, \lambda, L, I)$, where

1. $Q = Q_C$,

2. $\Sigma = \Sigma_A$,

3. δ is defined by :

   $s \xrightarrow{a} t$ iff

   (a) $a \in \Sigma_A \wedge s \xrightarrow{a}_C t$, or

   (b) $a = \tau \wedge (\exists b : b \in \Sigma_B : s \xrightarrow{b}_C t)$

4. $\lambda(s) = \lambda_C(s) \cap L_A$,

5. $L = L_A$, and

6. $I = I_C$.

The effect of projecting on to A is just to elide the labels on actions performed by the B-component, replacing them with the silent action τ.

2.3 Automata on Infinite Strings

A Büchi automaton $\mathcal{B}$ is given by a structure $(Q, \Sigma, \delta, q_0, F)$, where Q is a *finite* set of states, Σ is a finite alphabet, $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation, q_0 is the initial state, and F is a non-empty subset of accepting states. Büchi automata recognize infinite strings over Σ. A *run* r of $\mathcal{B}$ over an infinite string w is a function $r : \mathbf{N} \to Q$ such that $r(0) = q_0$, and $(r(i), w_i, r(i+1)) \in \delta$ for every $i \in \mathbf{N}$. Let $\inf(r) = \{q || r^{-1}(q)| = \omega\}$ denote the states from Q that appear infinitely often in r. A run r is *accepting* iff $\inf(r) \cap F \neq \emptyset$; i.e., a state from F appears infinitely often along r. A word w is accepted by the automaton iff there is an accepting run over w. The *language* of $\mathcal{B}$, $\mathcal{L}(\mathcal{B})$, is the set of words accepted by it.

Büchi automata are used to specify temporal correctness properties. Propositional linear temporal formulae can be translated into equivalent Büchi automata

(cf. [Thomas 90]). We sometimes adopt the automata-theoretic approach to Model Checking [VW 86], in which the negation of the correctness property is expressed by a Büchi automaton $\mathcal{B}$, and every computation of an LTS A satisfies the correctness property iff the set of computations of the "product" $\mathcal{C} = \mathcal{B} \times A$ is empty.

2.3.1 Products with Büchi Automata

Let A be the LTS $(S, \Sigma, \delta, \lambda, L, I)$, and let $\mathcal{B}$ be a Büchi automaton (Q, L, ρ, q_0, F). Their product, denoted $\mathcal{B} \times A$, is a fair LTS (A', F'), where :

1. $S' = Q \times S$,
2. $\Sigma' = \Sigma$,
3. $((b, s), a, (c, t)) \in \delta'$ iff $(s, a, t) \in \delta$ and $(b, \lambda(s), c) \in \rho$,
4. $\lambda'(b, s) = (b, \lambda(s))$,
5. $L' = Q \times L$,
6. $I' = \{q_0\} \times I$, and
7. $F' = \{(b, s) | b \in F, s \in S\}$.

2.4 Temporal Logics and their Semantics

We will define the syntax and semantics of the *Full Branching Time Logic* CTL*. The other logics considered (CTL and LTL) are sublogics of CTL*, and will be defined as such.

CTL* (read as "Computation Tree Logic - star") [EH 82] derives its expressive power from the freedom of combining modalities which quantify over paths and the modalities which quantify states along a particular path. These modalities are $\mathsf{A}, \mathsf{E}, \mathsf{F}, \mathsf{G}, \mathsf{X}_s$, and U_w ("for all futures", "for some future", "sometime", "always",

"strong next-time", and "weak until", respectively), and they are allowed to appear in virtually arbitrary combinations. Formally, we inductively define a class of state formulas (true or false of states) and a class of path formulas (true or false of paths), which is the least set of formulas satisfying :

(S1) Any atomic proposition P is a state formula.

(S2) If p, q are state formulas, then so are $p \wedge q$ and $\neg p$.

(S3) If p is a path formula then $\mathsf{E}p$ is a state formula.

(P1) Any state formula p is also a path formula.

(P2) If p, q are path formulas, then so are $p \wedge q$ and $\neg p$.

(P3) If p, q are path formulas then so are $\mathsf{X}_s p$ and $p \mathsf{U}_w q$.

The semantics of a formula is defined with respect to a Kripke Structure $M = (S, R, \lambda, AP, I)$. We write $M, s \models p$ ($M, x \models p$) to mean that state formula p (path formula p) is true in structure M at state s (of fullpath x, resp.). When M is understood, we write simply $s \models p$ ($x \models p$). We define $\models$ inductively using the convention that x denotes a fullpath and x^i denotes the suffix fullpath starting at the ith position in x, provided $i < \mathit{length}(x)$. For a state s :

(S1) $s \models P$ iff $P \in \lambda(s)$ for atomic proposition P

(S2) $s \models p \wedge q$ iff $s \models p$ and $s \models q$,
$s \models \neg p$ iff not $(s \models p)$

(S3) $s \models \mathsf{E}p$ iff for some fullpath x starting at s, $x \models p$

For a fullpath x :

(P1) $x \models p$ iff $x_0 \models p$, for any state formula p

(P2) $x \models p \wedge q$ iff $x \models p$ and $x \models q$,

$x \models \neg p$ iff not $(x \models p)$

(P3) $x \models \mathsf{X}_s p$ iff x^1 is defined and $x^1 \models p$,

$x \models (p \,\mathsf{U}_w\, q)$ iff $(\forall i : i < |x| : (\forall j : j \leq i : x^j \models \neg q) \Rightarrow x^i \models p)$.

We say that state formula p is *valid*, and write $\models p$, if, for every structure M and every state s in M, it is the case that $M, s \models p$. We say that state formula p is satisfiable iff for some structure M and some state s in M, it is the case that $M, s \models p$. In this case we also say that M defines a *model* of p. We define validity and satisfiability for path formulas similarly.

Other connectives can then be defined as abbreviations in the usual way: $p \vee q \equiv \neg(\neg p \wedge \neg q)$, $p \Rightarrow q \equiv \neg p \vee q$, $p \Leftrightarrow q \equiv (p \Rightarrow q) \wedge (q \Rightarrow p)$, $\mathsf{A}p \equiv \neg\,\mathsf{E}\neg p$, $\mathsf{G}p \equiv p\,\mathsf{U}_w\, false$, and $\mathsf{F}p \equiv \neg\,\mathsf{G}\neg p$. Further operators may also be defined as follows:

$\mathsf{X}_w p \equiv \neg\mathsf{X}_s\neg p$ is the weak next-time,

$p\mathsf{U}_s q \equiv (p\mathsf{U}_w q) \wedge \mathsf{F}q$ is the strong until,

$\overset{\infty}{\mathsf{F}} p \equiv \mathsf{GFX}_s p$ means infinitely often p,

$\overset{\infty}{\mathsf{G}} p \equiv \mathsf{FGX}_s p$ means almost everywhere p,

$inf \equiv \mathsf{GX}_s true$ means the path is infinite, and

$fin \equiv \mathsf{FX}_w false$ means the path is finite.

Sublogics of CTL^* are defined either by restrictions on the operators allowed, or by restrictions on the ways in which the operators may be combined.

LTL (Linear Temporal Logic) [Pnueli 77] is obtained by leaving out (S2) and (S3), and by redefining $M, s \models f$ to be true iff for every fullpath x that starts at s, $M, x \models f$.

ACTL^* is the subset of CTL^* where in each formula, every occurrence of A is under an even number of negations, and every occurrence of E is under an odd number of negations. The dual logic is known as ECTL^*.

$\mathsf{CTL}^*\backslash\mathsf{X}$ is obtained by leaving out the operator X_s. This makes the logic

insensitive to *stuttering* (finite repetitions of the same state).

CTL (Computation Tree Logic) is obtained by restricting the kinds of path formulas possible. The new rules are :

(S1) Any atomic proposition P is a state formula.

(S2) If p, q are state formulas, then so are $p \wedge q$, $\neg p$.

(S3) If p is a path formula then $\mathsf{E}p$ is a state formula.

(P1) Any state formula p is also a path formula.

(P2) If p, q are path formulas then so are $\mathsf{X}_s p$ and $p\mathsf{U}_w q$.

The rules ensure that path quantifiers are always associated with a unique temporal operator. Thus, $\mathsf{AG}(p \Rightarrow \mathsf{AF}q)$ is a CTL formula, while $\mathsf{AG}(p \Rightarrow \mathsf{F}q)$ is not.

CTL* itself is a sublogic of the μ-calculus, which is defined as the least set of formulas that satisfy :

- Any atomic proposition P, and any variable symbol Z is a formula.
- If p, q are formulas, then so are $p \wedge q$, $\neg p$.
- If p is a formula, then $\mathsf{EX}p$ is a formula.
- If p is a formula, then $(\mu Z :: p)$ is a formula.

The paper by Emerson [Emerson 90] contains a detailed survey of these logics and comparisons of their expressive power.

2.4.1 Indexed Logics

For systems that are formed by the composition of many similar systems, it is often the case that the correctness properties are of the form : "for every process P

holds", or "for every pair of processes P holds". To express these properties concisely, it is convenient to use an indexed proposition set and quantification over the index set [RS 85, BCG 89]. Quantified properties are represented here in the form $\bigwedge_{\bar{x}:R(\bar{x})} f(\bar{x})$, where $\bar{x}$ is a non-empty list of bound names, R is an expression denoting the range of these names, and f is a temporal logic formula with propositions indexed by these names. For example, mutual exclusion among a collection of n processes is expressed by $\bigwedge_{i,j:i\neq j} \mathsf{AG}(\neg(C_i \wedge C_j))$, where C_k is an indexed proposition denoting the condition that process k is in its critical region.

The interpretation of an indexed formula is performed with respect to an index set A over a Kripke Structure $M = (S, R, L, AP, I)$, where AP may be partitioned into "global" propositions GP and indexed local propositions, $LP \times A$. In this structure, $M, s \models \bigwedge_{\bar{x}:R(\bar{x})} f(\bar{x})$ iff for every vector of values $\bar{a}$ over $A^{|\bar{x}|}$ such that $R(\bar{a})$ holds, $M, s \models f(\bar{a})$. The interpretation of other connectives is as defined earlier.

2.5 Model Checking

Model Checking [CE 81] (cf. [QS 82]) is an algorithmic procedure for determining the truth of a temporal formula over a finite-state system. For the logic CTL, this procedure uses the fixpoint formulation of CTL operators. For instance, the operator $\mathsf{E}(f\mathsf{U}g)$ may be expressed as $(\mu Z :: g \vee (f \wedge \mathsf{EX}Z))$. Fixpoint evaluation may be done using the Knaster-Tarski theorem [Tarski 55], by which the set of states defining a least fixpoint $(\mu Z : f(Z))$ can be computed as $(\bigcup i :: f^i(\emptyset))$, where the range of i is over the ordinals that are at most the cardinal number of the structure. For finite-state systems, this computation terminates in at most n iterations, where n is the number of states. This procedure for CTL has been extended to the logics FairCTL and the μ-calculus [EL 86].

An alternative view of Model Checking uses automata [VW 86]. Here, the

set of structures satisfying a formula f is encoded using a finite automaton, $\mathcal{A}_f$. This may be done using either automata on infinite strings or automata on infinite trees. If the string approach is taken, then Model Checking reduces to the problem of determining whether every computation of M is a computation accepted by $\mathcal{A}_f$, i.e., $\mathcal{L}(M) \subseteq \mathcal{L}(\mathcal{A}_f)$. This language inclusion problem may be decided in PSPACE . For the tree approach, Model Checking can be phrased as $M \in \mathcal{L}(\mathcal{A}_f)$, which is a membership problem. The complexity of determining membership depends on the acceptance conditions of the tree automaton [EJ 88].

The model-checking approach has by now been applied to a wide variety of logics such as branching-time logics [CE 81, QS 82, EL 86] and linear-time logic [LP 85], and to a variety of programming models such as finite-state programs [CE 81, QS 82] and real-time systems [ACD 90].

2.6 Equivalences on Transition Systems

The notions of *simulation* [Milner 71] and *bisimulation* [Park 81, Milner 90] are the basic ways of comparing the structure of transition systems. There are several variants; an important one being bisimulation under stuttering (finite repetition) of state labelings [BCG 88, dNV 90, Milner 90].

We present here the three main notions cited above. For simplicity, the definitions are with respect to a single structure; comparisons between structures can be reduced to this case by forming a single structure that is the disjoint union of the structures. In the following, let $A = (Q, \Sigma, \delta, \lambda, L, I)$ be a LTS.

Definition 2.4 (Simulation) *A relation $R \subseteq Q \times Q$ is a simulation relation on A iff for any s, t such that $s\ R\ t$,*

1. $\lambda(s) = \lambda(t)$,
2. $(\forall a, u : s \xrightarrow{a}_A u : (\exists v : t \xrightarrow{a}_A v : u\ R\ v))$.

This concept can be formulated in other ways. A alternative formulation is the following : R is a simulation relation iff $R \subseteq F(R)$, where F is a function mapping relations to relations that is defined by the the two items above. It is straightforward to show that F is monotone over the complete lattice of relations on the LTS; hence, by the Knaster-Tarski theorem [Tarski 55], it has a greatest fixpoint. This is the greatest simulation relation on the LTS. If $s\ R\ t$ for some simulation relation R, we say that *s simulates t*. Equivalently, s simulates t iff (s,t) is in the greatest simulation relation.

Theorem 2.1 *The greatest simulation relation is reflexive and transitive.*

For *total* transition systems, there is a nice connection between the logical and structural properties of the transition system; the first of many such connections to be discussed here.

Theorem 2.2 *For a total, finite-branching Kripke structure M, if s simulates t, then for every* ACTL* *formula f, $M,s \models f$ implies that $M,t \models f$.*

Definition 2.5 (Bisimulation [Park 81]) *A relation $R \subseteq Q \times Q$ is a bisimulation relation on A iff it is symmetric, and for any s,t such that $s\ R\ t$,*

1. $\lambda(s) = \lambda(t)$,
2. $(\forall a, u : s \xrightarrow{a}_A u : (\exists v : t \xrightarrow{a}_A v : u\ R\ v))$.

Note that the only additional requirement here, over the simulation relation definition, is that R be symmetric. This concept can also be formulated as the following : R is a bisimulation relation iff $R \subseteq F(R)$, where F is a function mapping relations to relations that is defined by the the two items above. It is straightforward to show that F is monotone over the complete lattice of relations on the LTS; hence, by the Knaster-Tarski theorem [Tarski 55], it has a greatest fixpoint. This is the

greatest bisimulation relation on the LTS. If $s\ R\ t$ for some bisimulation relation R, we say that s is *bisimilar to* t. Equivalently, s is bisimilar to t iff (s,t) is in the greatest bisimulation relation.

Theorem 2.3 *[Milner 90] The greatest bisimulation relation is an equivalence relation.*

The connection between bisimulation and CTL^* is stronger than that for simulation.

Theorem 2.4 *[HM 85, BCG 88] For a finite-branching Kripke structure M, s is bisimilar to t iff for every CTL^* formula f, $M,s \models f$ iff $M,t \models f$.*

The definition above is often referred to as "strong bisimulation" since it requires that single actions are matched by bisimilar states. This is often too restrictive a notion, especially when stuttering (repetition of the same state) and τ (silent) actions are introduced by hiding details of processes. To take such stuttering into account, several variants of bisimulation have been proposed [BCG 88, dNV 90, Milner 90]. We present here the definition of *stuttering bisimulation*, modified from the original definition in [BCG 88] which applies only to finite total structures.

Definition 2.6 (Stuttering Bisimulation) *A relation $R \subseteq S \times S$ is a stuttering bisimulation on A iff R is symmetric and for every s,t such that $s\ R\ t$,*

1. $\lambda(s) = \lambda(t)$,
2. $(\forall \sigma : fp(s,\sigma) : (\exists \delta : fp(t,\delta) : match_R(\sigma,\delta)))$.

where $fp(s,\sigma)$ is true iff σ is a fullpath starting at s ($\sigma_0 = s$), and $match_R(\sigma,\delta)$ is true iff σ and δ can be divided into an equal number of non-empty, finite, segments such that any pair of states from segments with the same index is in the relation R. The formal definition of *match* is given below:

Let INC be the set of strictly increasing sequences of natural numbers starting at 0. Precisely, $INC = \{\pi \mid \pi : \mathbf{N} \to \mathbf{N} \wedge \pi(0) = 0 \wedge (\forall i : i \in \mathbf{N} : \pi(i) < \pi(i+1))\}$. Let σ be a path, and π a member of INC. For $i \in \mathbf{N}$, let $intv(i, \sigma, \pi) = [\pi(i), min(\pi(i+1), length(\sigma)))$. The ith segment of σ w.r.t. π, $seg(i, \sigma, \pi)$, is defined by the sequence of states of σ with indices in $intv(i, \sigma, \pi)$.

Let σ and δ, under partitions π and ξ respectively, correspond w.r.t. R iff they are subdivided into the same number of segments, and any pair of states in segments with the same index are related by R. Precisely, $corr_B((\sigma, \pi), (\delta, \xi)) \equiv (\forall i : i \in \mathbf{N} : intv(i, \sigma, \pi) \neq \emptyset \equiv intv(i, \delta, \xi) \neq \emptyset \wedge (\forall m, n : m \in intv(i, \sigma, \pi) \wedge n \in intv(i, \delta, \xi) : (\sigma_m, \delta_n) \in R)))$.

Paths σ and δ match iff there exist partitions that make them correspond. Precisely, $match_R(\sigma, \delta) \equiv (\exists \pi, \xi : \pi, \xi \in INC : corr_R((\sigma, \pi), (\delta, \xi)))$.

It can be shown, applying the Knaster-Tarski theorem, that the greatest stuttering bisimulation exists.

Theorem 2.5 *The greatest stuttering bisimulation is an equivalence relation.*

Theorem 2.6 *[BCG 88] For a finite-branching Kripke structure M, if s is stuttering bisimilar to t, then for every* $\mathsf{CTL}^*\backslash\mathsf{X}$ *formula f, $M, s \models f$ iff $M, t \models f$.*

Recall that $\mathsf{CTL}^*\backslash\mathsf{X}$ is the sublogic of CTL^* that is insensitive to stuttering. Many qualitative properties of systems can be expressed using $\mathsf{CTL}^*\backslash\mathsf{X}$; e.g., absence of starvation ($\mathsf{AG}(trying \Rightarrow \mathsf{AF}\, critical)$) and mutual exclusion ($\mathsf{AG}(\neg(critical_0 \wedge critical_1))$).

Chapter 3

Reasoning about Rings

3.1 Introduction

The ring is one of the most useful ways for structuring systems of concurrently executing processes. Well known examples include protocols for mutual exclusion, leader election, scheduling, and the dining philosophers problem. These have two features in common : the individual processes of the ring are isomorphic (i.e., the "code" of one process can be transformed into that of another by a simple renaming), and the desired correctness properties are expected to be satisfied by instances of arbitrary size. The protocols are thus parameterized by the number of processes. The usual method of verifying that such a parameterized system satisfies a specification is by a direct proof by hand [CM 88, MP 94] which requires considerable ingenuity and can, in practice, be done only for reasonably simple protocols.

In this chapter we show that for *any* system of many isomorphic processes organized in a ring which communicate through a token used as a signal, a property holds for *every* instance of the system iff it holds for instances up to a small *cutoff* size. Thus, for systems composed of isomorphic finite state processes, a fully automated technique, such as Model Checking [CE 81, CES 86] may be applied for the

verification of the parameterized system.

The logic in which correctness properties are expressed is the branching time logic CTL^* without the *next-time* operator X, which we denote by $\mathsf{CTL}^*\backslash\mathsf{X}$ [BCG 89]. The semantics of this logic is presented in Chapter 2. Formulas of this logic are insensitive to "stuttering" (repeated occurrences of the same state). Since the formulas have to hold for rings of various sizes, it seems reasonable to make them free of next-time requirements, which in general vary among rings of various sizes. It is, however, possible to handle a "next-action by process i" modality.

Correctness properties of parameterized systems are typically expressed as properties in an indexed temporal logic. Chapter 2 contains a description of the syntax and semantics of such logics. We consider several types of indexed temporal properties, where the temporal modalities are from $\mathsf{CTL}^*\backslash\mathsf{X}$. For instance, mutual exclusion can be expressed as $\bigwedge_{i,j:i\neq j} \mathsf{AG}\neg(\mathit{critical}_i \wedge \mathit{critical}_j)$, and absence of starvation by $\bigwedge_i \mathsf{AG}(\mathit{trying}_i \Rightarrow \mathit{critical}_i)$. The results for specific types of indexed formulas are as follows, where g is a quantifier-free formula of indexed $\mathsf{CTL}^*\backslash\mathsf{X}$.

- $\bigwedge_i g(i)$ has a cutoff of 2.
- $\bigwedge_i g(i, i+1)$ has a cutoff of 3.
- $\bigwedge_{i,j:i\neq j} g(i,j)$ has a cutoff of 4.
- $\bigwedge_{i,j:i\neq j} g(i,i+1,j)$ has a cutoff of 5.

The rotational symmetry of the ring plays an important role in the proofs of these results. It allows us to reduce the check of an indexed formula such as $\bigwedge_i g(i)$, which ranges over all possible process indices in an instance of the system, to that of one with a particular index, say $g(0)$. We then establish the existence of a correspondence between the state transition graphs of an instance of the system with n processes, and one with the number of processes equal to the cutoff. The

correspondence established is for the projection of the state transition graph on the particular process index; the symmetry then allows us to establish the result. The main observation of the correspondence proof is that a segment of the ring not containing any observable processes (e.g., for $\bigwedge_i g(i)$, one not containing process 0) has behaviour similar to that of a single process.

Section 3.2 defines the notation and constructions we need. Section 3.3 contains the definition of the system model. We prove the main results in Section 3.4. Section 3.5 contains applications of these results to two protocols. Section 3.7 contains a discussion of related work. The technical details of proofs are provided in Section 3.8.

3.2 Preliminaries

Each process in the ring is modeled as a Labeled Transition System (LTS). Chapter 2 contains the precise definition of LTS's and of operators on LTS's such as composition and projection. To compare the instances of the ring system, we use a notion of equivalence under stuttering that is simpler than the original definition that is reproduced in Chapter 3.2. In Section 3.8, we show that the new definition gives rise to the same greatest solution as the original one; thus, they enjoy many common properties.

3.2.1 Block Bisimulation

Let A be an LTS, $A = (Q, \Sigma, \delta, \lambda, L, I)$. Let $\approx$ be an equivalence relation on Q. $MAXF_{\approx}(s)$ is the set of finite paths of the form $\sigma; v$, where σ begins at s ($\sigma_0 = s$), every state in σ is in the same $\approx$-class as s ($(\forall i : i < \mathit{length}.\sigma : \sigma_i \approx s)$), and v is in a different class $\neg(v \approx s)$. $MAXI_{\approx}(s)$ is the set of all fullpaths σ, where σ begins at s and every state in σ is in the same $\approx$-class as s.

Definition 3.1 (Block Bisimulation) *Let $\approx$ be an equivalence relation on Q, and let h be a bijection on AP. A relation B on Q is a block bisimulation on A w.r.t. $(\approx, h)$ iff*

1. *B is symmetric,*
2. *B is monotonic w.r.t. $\approx$: For any s, t, u, v if $(s,t) \in B$, $s \approx u$, and $t \approx v$, then $(u,v) \in B$.*
3. *Whenever $(s,t) \in B$ then*
   (a) $h(\lambda(s)) = \lambda(t)$,
   (b) $(\forall \sigma, v : \sigma; v \in MAXF_{\approx}(s) : (\exists \delta, w : \delta; w \in MAXF_{\approx}(t) : vBw))$
   (c) $(\forall \sigma : \sigma \in MAXI_{\approx}(s) :$
   $(\exists \delta : \delta \in MAXI_{\approx}(t) : length(\delta) = \omega \equiv length(\sigma) = \omega))$

□

The term "block" bisimulation is used as the definition essentially compares paths consisting of states in the same $\approx$-class (block). In order to compare two LTS's A and B, form the disjoint union C of A and B and define a bisimulation on C. For a formula f, let $h(f)$ be the formula obtained by replacing each proposition P of f by $h(P)$. The following theorem is shown in Section 3.8 :

Theorem 3.1 *Let B be a block bisimulation on LTS A w.r.t. $(\approx, h)$. If $(s,t) \in B$, then for any formula f of $\mathsf{CTL}^*\backslash\mathsf{X}$, $A, s \models f$ iff $A, t \models h(f)$.* □

We say that states s and t are block equivalent, if there is some block bisimulation on A that includes the pair (s,t). We write $A \sim B$, for LTS's A and B iff every initial state in A has a block bisimilar initial state of B and vice-versa.

3.2.2 Group Theoretic notions

Some simple notation from Group Theory is needed to define the symmetries of structures. For an natural number n, let $[n]$ represent the set $\{x \mid 0 \leq x \leq n-1\}$. Let $Sym\ I$ be the full symmetry group on the index set I. Let $\mathcal{C}_n$ be the permutation group of rotations on a ring of size n. For groups $\mathcal{C}$ and $\mathcal{D}$, we write $\mathcal{C} \preceq \mathcal{D}$ to mean that $\mathcal{C}$ is a subgroup of $\mathcal{D}$. Clearly, $\mathcal{C}_n \preceq Sym[n]$.

For a formula f indexed over a set I, $Aut\ f = \{\pi | \pi \in Sym\ I \wedge \pi(f) \equiv f\}$. For an LTG M composed of n processes in parallel, where the states are indexed by $[n]$, $Aut\ M$ is the group of permutations in $Sym\ [n]$ that when applied to every state and transition of M, map M to an isomorphic copy [ES 93]. Abusing notation slightly, we define $Aut\ s$ for a state s to be the set of permutations that when applied to s, map it to itself.

3.3 System Model

Informally, the token passing model is defined by the following :

- Initially, a nondeterministically chosen process has the unique token.
- A process has two types of transitions : those that are enabled only if the process has the token, and others which can be enabled without the process possessing the token. We let the system evolve according to pure nondeterminism; i.e., no fairness constraint is assumed.
- The process with the token must eventually transmit it in a clockwise direction.

Formally, individual processes of a ring are constructed from a template process T, which is an LTS defined by :

1. The set of states, $Q \times \mathbf{B}$. The boolean component indicates possession of the token.

2. The set of actions, Σ, which is partitioned into Σ_f, the set of "free" actions, Σ_d, the set of "token dependent" actions, and $\{snd, rcv\}$, the set of token transfer actions.

3. The transition relation δ, where

   - For every $(q,b) \xrightarrow{a} (q',b') \in \delta$,
     (a) $a \in \Sigma_f \Rightarrow b \equiv b'$ (A free transition cannot change possession.)
     (b) $a \in \Sigma_d \Rightarrow b \wedge b'$ (A token dependent transition can execute only if the process possesses the token.)
     (c) $a = rcv \Rightarrow \neg b \wedge b'$ (A receive establishes possession of the token.)
     (d) $a = snd \Rightarrow b \wedge \neg b'$ (A send revokes possession of the token.)
   - For every (q,b) such that $(q^0, false) \xrightarrow{a} (q,b) \in \delta$, $a = rcv$. (The only possible initial action is a receive.)
   - rcv and snd actions alternate along every path in T.
   - δ is *total* in the first component. (The process is nonterminating)

4. The set of propositions is $Q \times \mathbf{B}$, and the labelling function is the identity function.

5. The initial state is $(q^0, false)$.

For a finite-state template process, it is possible to check the conditions on δ automatically, by model-checking a certain CTL formula.

An individual process K_i $(i \in [n], n \geq 1)$ in an instance of the system with n processes is defined by $K_i = \tau_i(T)$, where τ_i is the re-labelling defined by the action renamings $\{rcv_i/rcv, snd_{i+1}/snd\} \cup \{a_i/a : a \in \Sigma_f \cup \Sigma_d\}$ and the state renamings s_i/s for every state s. The instance is defined by $Ring(n) = K_0 \parallel K_1 \parallel \ldots \parallel K_{n-1} \parallel D_n$, where D_n is the initial token distribution process that synchronizes

with a non-deterministically chosen K_i to pass it the token. For each composition $K_i \parallel K_{i+1}$, the synchronized actions are snd_{i+1} and rcv_{i+1} (all arithmetic in a ring of size n is done modulo n). Let $M(n)$ denote the LTS induced by $Ring(n)$. For a set of indices $I \subseteq [n]$, we let $M(n) \downarrow_I$ denote the projection of $M(n)$ on to the processes indexed by I. For a set P of propositions indexed by $[n]$, P_I denotes the subset that is indexed with elements from I.

Definition 3.2 (Ring Intervals) *Let $[i:j]_n$ denote the indices on the clockwise segment from i to j on a ring of size n. Precisely, $[i:j]_n = \{i+k \mid k \in [n] \wedge (\forall l : l < k : i+l \neq j)\}$. Addition is modulo n. Let $[i:j)_n = [i:j]_n - \{j\}$, $(i:j]_n = [i:j]_n \backslash \{i\}$, and $(i:j)_n = [i:j]_n \backslash \{i,j\}$. Note that from the definitions, $(i:i)_n$, $[i:i)_n$, and $(i:i]_n$ are all empty.* □

3.4 Property specific abstractions

We show here that for specific types of properties, it suffices to consider instances of size at most a small *cutoff* size in order to show that the property holds for *all* instances.

The properties that we consider are of the forms $\bigwedge_i g(i)$, $\bigwedge_{i,j:i\neq j} g(i,j)$, and $\bigwedge_{i,j:i\neq j} g(i,i+1,j)$. In each case, we first show by symmetry arguments, that in each instance of size n, it is sufficient to instantiate the quantifier $\bigwedge_i$ with some process index, say 0. We then prove that a system of size greater than the cutoff size satisfies this smaller formula iff the system of cutoff size does. This is shown by exhibiting a block bisimulation between the large and the small system, using the following key theorem.

Theorem 3.2 (Reduction Theorem) *Let $I \subseteq [n]$, and $J \subseteq [k]$ be sets of indices such that there is a bijection $h : I \to J$ such that for any $i, j \in I$,*

1. *$i \leq j$ iff $h.i \leq h.j$, and*

2. $(i:j)_n \neq \emptyset$ *iff* $(h(i):h(j))_k \neq \emptyset$.

Then $M(n)\downarrow_I$ *is block equivalent to* $M(k)\downarrow_J$.

Proof. Let h^A be the derived renaming function for indexed actions from $M(n)$ defined by $h^A(a_i) = a_{h(i)}$. Let h^P be the derived renaming function for indexed propositions from $M(n)$ defined by $h^P(P_i) = P_{h(i)}$. We define a relation that is a block bisimulation on $M(n)$ and $M(k)$ w.r.t. h^P.

For a state s of $M(i)$, let $tok_i(s)$ be the index of the process that possesses the token in state s. Let R_{nk} be the relation between states of $M(n)$ and $M(k)$, defined as follows:
$s\ R_{nk}\ t$ iff

1. The local states of D_n in s and D_k in t are identical (recall that D_n is the initial token distribution process),
2. $h^P(\lambda(s)\downarrow_I) = \lambda(t)\downarrow_J$,
3. $(\forall i : i \in I : tok_n.s = i \ \equiv\ tok_k.t = h.i)$, and
4. $(\forall i,j : i,j \in I : tok_n(s) \in (i:j)_n \ \equiv\ tok_k(t) \in (h(i):h(j))_k)$.

The relation R_{nn} is similar but for (2), which is modified to $\lambda(s)\downarrow_I = \lambda(t)\downarrow_I$. Define the relation $\approx$ to be $R_{nn} \cup R_{kk}$. Let $B = R_{nk} \cup R_{kn}$. B is symmetric. It is straightforward to check that $\approx$ is an equivalence relation, and that B is monotonic w.r.t. $\approx$. Section 3.8 contains the proof that B is a block bisimulation w.r.t. $(\approx, h^P)$.

Informally, the bisimulation treats states as equivalent if corresponding processes (w.r.t. h) in I and J have the same local state (conditions 1,2), the token is at some process in I iff it is at the corresponding process in J (condition 3), and the token is in between two processes in I iff it is in between the corresponding processes in J (condition 4).

□

3.4.1 Properties of the form $\bigwedge_i g(i)$

Properties of the form $\bigwedge_i g(i)$ refer to properties that every individual process in a system of processes must satisfy. They are typically used to specify progress requirements, as in absence of starvation ($\bigwedge_i \mathsf{AG}(\mathit{trying}_i \Rightarrow \mathit{critical}_i)$). We show first that for a symmetric system, it is possible to reduce this property to its instantiation with a single index. The following lemma is a refinement of one in [ES 93].

Lemma 3.1 *If A is the LTS of a system with n isomorphic processes, $\mathcal{C}_n \preceq Aut\ A$, and the start state q^0 is symmetric (i.e $Aut\ q^0 = Sym\ [n]$) then, $A, q^0 \models \bigwedge_i g(i)$ iff $A, q^0 \models g(0)$.*

Proof.

($LHS \Rightarrow RHS$) This is immediate from the definition of $\models$.

($LHS \Leftarrow RHS$)

$A, q^0 \models g(0)$

$\Rightarrow$ (π is a permutation)

$(\forall \pi :\ \pi \in Aut\ A :\ \pi(A), \pi(q^0) \models g(\pi(0)))$

iff ($\pi(A) = A$ as $\pi \in Aut\ A$, $\pi(q^0) = q^0$, as q^0 is symmetric)

$(\forall \pi :\ \pi \in Aut\ A :\ A, q^0 \models g(\pi(0)))$

$\Rightarrow$ (as $\mathcal{C}_n \leq Aut\ A$)

$(\forall \pi :\ \pi \in \mathcal{C}_n :\ A, q^0 \models g(\pi(0)))$

$\Rightarrow$ (The cyclic shifts in $\mathcal{C}_n$ drive 0 to i for every $i \in [n]$)

$(\forall i : i \in [n] : A, q^0 \models g(i))$

iff (by definition)

$A, q^0 \models \bigwedge_i g(i)$

□

Theorem 3.3 *Let f be a formula of the form $\bigwedge_i g(i)$, where $g(i)$ is a* $\mathsf{CTL}^*\backslash\mathsf{X}$ *formula that refers only to propositions indexed by i. Then, $(\forall n : n \geq 2 : M(n), q_n^0 \models f \equiv M(2), q_2^0 \models f)$.*

Proof.

For any $n \geq 2$,

$M(n), q_n^0 \models f$

iff (the initial state is symmetric, lemma 3.1)

$M(n), q_n^0 \models g(0)$

iff (as $g(0)$ refers only to propositions indexed by 0)

$M(n)\downarrow_{\{0\}}, q_n^0 \models g(0)$

iff (By Theorem 3.2, $M(n)\downarrow_{\{0\}}$ and $M(2)\downarrow_{\{0\}}$ are block equivalent)

$M(2)\downarrow_{\{0\}}, q_2^0 \models g(0)$

iff (as $g(0)$ refers only to propositions indexed by 0)

$M(2), q_2^0 \models g(0)$

iff (by lemma 3.1)

$M(2), q_2^0 \models f$

□

3.4.2 Properties of the form $\bigwedge_i g(i, i+1)$

Formulas with the form $\bigwedge_i g(i, i+1)$ can express properties of neighboring pairs of processes. For instance, the exclusion property for the Dining Philosophers problem may be expressed as $\bigwedge_i \mathsf{AG}(\neg(\mathit{eating}_i \wedge \mathit{eating}_{i+1}))$. By a proof analogous to that for Lemma 3.1, we have the following lemma.

Lemma 3.2 *If A is the global state transition graph of a system with n isomorphic processes, $\mathcal{C}_n \preceq Aut\ A$, and the start state q^0 is symmetric, then $A, q^0 \models \bigwedge_i g(i, i+1)$ iff $A, q^0 \models g(0, 1)$.* □

We have to take into account the various situations that process K_0 and K_1 can be in. Intuitively, either K_0 or K_1 can be given the token initially, or some other process could be assigned the token. This suggests that a three process system may be sufficient. And in fact we have, by an argument analogous to that for Theorem 3.3,

Theorem 3.4 *Let f be a formula of the form $\bigwedge_i g(i, i+1)$, where $g(i, i+1)$ is a* $\mathsf{CTL}^*\backslash\mathsf{X}$ *formula. Then,* $(\forall n : n \geq 3 : M(n), q_n^0 \models f \equiv M(3), q_3^0 \models f)$ □

3.4.3 Properties of the form $\bigwedge_{i,j:i\neq j} g(i,j)$

We consider next formulas of the form $\bigwedge_{i,j:i\neq j} g(i,j)$. These are used to express properties that involve distinct pairs of processes. Mutual exclusion, for instance, can be written as $\bigwedge_{i,j:i\neq j} AG(\neg(critical_i \wedge critical_j))$.

Lemma 3.3 *If A is the LTS of a system with n isomorphic processes, $\mathcal{C}_n \preceq Aut\ A$, and the start state q^0 is symmetric then,*
$A, q^0 \models \bigwedge_{i,j:i\neq j} g(i,j)$ *iff* $A, q^0 \models \bigwedge_{jj\neq 0} g(0,j)$.

Proof.
$(LHS \Rightarrow RHS)$ This is immediate from the definition of $\models$.
$(LHS \Leftarrow RHS)$

$A, q^0 \models \bigwedge_{jj\neq 0} g(0,j)$
$\Rightarrow$ (π is a permutation)
$(\forall \pi :\ \pi \in Aut\ A :\ \pi(A), \pi(q^0) \models \bigwedge_{jj\neq\pi(0)} g(\pi(0), j))$
iff ($\pi(A) = A$ as $\pi \in Aut\ A$, $\pi(q^0) = q^0$, as q^0 is symmetric)
$(\forall \pi :\ \pi \in Aut\ A :\ A, q^0 \models \bigwedge_{jj\neq\pi(0)} g(\pi(0), j))$
$\Rightarrow$ (as $\mathcal{C}_n \leq Aut\ A$)
$(\forall \pi :\ \pi \in \mathcal{C}_n :\ A, q^0 \models \bigwedge_{jj\neq\pi(0)} g(\pi(0), j))$

$\Rightarrow$ (The cyclic shifts in $\mathcal{C}_n$ drive 0 to i for every $i \in [n]$)

$(\forall i : i \in [n] : A, q^0 \models \bigwedge_{jj \neq i} g(i,j))$

iff (by definition)

$A, q^0 \models \bigwedge_{i,j:i \neq j} g(i,j)$

□

From this lemma, we need to consider only the pairs of processes $(0, j)$, such that $j \neq 0$. Applying the Reduction Theorem, we obtain the following theorem :

Theorem 3.5 *For any $n \geq 4$,*

1. $M(n) \downarrow_{(0,1)} \sim M(4) \downarrow_{(0,1)}$,
2. $M(n) \downarrow_{(0,n-1)} \sim M(4) \downarrow_{(0,3)}$, *and*
3. $M(n) \downarrow_{(0,j)} \sim M(4) \downarrow_{(0,2)}$, *for $j \notin \{0, 1, n-1\}$.*

Proof. Let $h : [n] \to [4]$ be defined by $h(0) = 0$, $h(1) = 1$, $h(n-1) = 3$, and $h(j) = 2$, for $j \notin \{0, 1, n-1\}$. Each of the claims above follows from Theorem 3.2 by observing that h restricted to the pairs mentioned in each claim is a bijection that satisfies both preconditions of the theorem. □

Corollary 3.1 *For any $n \geq 4$, and a formula $g(i,j)$ of $\mathsf{CTL}^*\backslash\mathsf{X}$,*

1. $M(n), q_n^0 \models g(0,1)$ *iff* $M(4), q_4^0 \models g(0,1)$,
2. $M(n), q_n^0 \models g(0, n-1)$ *iff* $M(4), q_4^0 \models g(0,3)$, *and*
3. $M(n), q_n^0 \models g(0,j)$ *iff* $M(4), q_4^0 \models g(0,2)$, *for $j \notin \{0, 1, n-1\}$.*

Proof. The corollary follows from Theorems 3.1 and 3.5. □

The main theorem results from applying this corollary:

Theorem 3.6 *Let f be a formula of the form $\bigwedge_{i,j:i \neq j} g(i,j)$, where $g(i,j)$ is a $\mathsf{CTL}^*\backslash\mathsf{X}$ formula. Then,* $(\forall n : n \geq 4 : M(n), q_n^0 \models f \equiv M(4), q_4^0 \models f)$. □

Proof.

For any $n \geq 4$,

$M(n), q_n^0 \models f$

iff (the initial state is symmetric, lemma 3.3)

$M(n), q_n^0 \models \bigwedge_{jj\neq 0} g(0,j)$

iff (splitting up the formula)

$M(n), q_n^0 \models g(0,1) \wedge M(n), q_n^0 \models g(0,n-1) \wedge$

$M(n), q_n^0 \models \bigwedge_{jj\notin\{0,1,n-1\}} g(0,j)$

iff (by corollary 3.2)

$M(4), q_4^0 \models g(0,1) \wedge M(4), q_4^0 \models g(0,3) \wedge$

$M(4), q_4^0 \models g(0,2)$

iff (recombining the formula)

$M(4), q_4^0 \models \bigwedge_{jj\neq 0} g(0,j)$

iff (by lemma 3.3)

$M(4), q_4^0 \models f$

□

As a particular case, the formula $\bigwedge_{i,j:i\neq j} AG\neg(critical_i \wedge critical_j)$, which expresses mutual exclusion, can be checked in a 4-process system.

3.4.4 Properties of the form $\bigwedge_{i,j:i\neq j} g(i,i+1,j)$

Properties of the form $\bigwedge_{i,j:i\neq j} g(i,i+1,j)$ are used to express global properties that indicate the relationship between a pair of neighboring processes and an arbitrary process. An example of such a property is presented in Section 3.5.

Lemma 3.4 *If A is the LTS of a system with n isomorphic processes, $n > 1$. $\mathcal{C}_n \preceq Aut\ A$, and the start state q^0 is symmetric, then, $A, q^0 \models \bigwedge_{i,j:i\neq j} g(i,i+1,j)$ iff $A, q^0 \models \bigwedge_{jj\neq 0} g(0,1,j)$.*

Proof.

$(LHS \Rightarrow RHS)$ This is immediate from the definition of $\models$.

$(LHS \Leftarrow RHS)$

$\quad A, q^0 \models \bigwedge_{jj \neq 0} g(0,1,j)$

$\Rightarrow$ (π is a permutation)

$\quad (\forall \pi : \ \pi \in Aut\ A : \ \pi(A), \pi(q^0) \models \bigwedge_{jj \neq \pi(0)} g(\pi(0), \pi(1), j))$

iff ($\pi(A) = A$ as $\pi \in Aut\ A$, $\pi(q^0) = q^0$, as q^0 is symmetric)

$\quad (\forall \pi : \ \pi \in Aut\ A : \ A, q^0 \models \bigwedge_{jj \neq \pi(0)} g(\pi(0), \pi(1), j))$

$\Rightarrow$ (as $\mathcal{C}_n \leq Aut\ A$)

$\quad (\forall \pi : \ \pi \in \mathcal{C}_n : \ A, q^0 \models \bigwedge_{jj \neq \pi(0)} g(\pi(0), \pi(1), j))$

$\Rightarrow$ (The cyclic shifts in $\mathcal{C}_n$ drive 0 to i for every $i \in [n]$)

$\quad (\forall i : i \in [n] : A, q^0 \models \bigwedge_{jj \neq i} g(i, i+1, j))$

iff (by definition)

$\quad A, q^0 \models \bigwedge_{i,j : i \neq j} g(i, i+1, j)$

□

From this lemma, we need to consider only the triples of process indices of the form $(0,1,j)$, such that $j \neq 0$. Application of the Reduction Theorem leads to the following theorem.

Theorem 3.7 *For any $n \geq 5$,*

1. $M(n) \downarrow_{(0,1)} \sim M(5) \downarrow_{(0,1)}$,
2. $M(n) \downarrow_{(0,1,2)} \sim M(5) \downarrow_{(0,1,2)}$,
3. $M(n) \downarrow_{(0,1,n-1)} \sim M(5) \downarrow_{(0,1,4)}$, *and*
4. $M(n) \downarrow_{(0,1,j)} \sim M(5) \downarrow_{(0,1,3)}$, *for $j \notin \{0,1,2,n-1\}$.*

Proof. Define $h : [n] \to [5]$ by $h(0) = 0$, $h(1) = 1$, $h(2) = 2$, $h(n-1) = 3$, and $h(j) = 3$, for $j \notin \{0,1,2,n-1\}$. Each of the claims above follows from Theorem 3.2

by observing that h restricted to the pairs mentioned in each claim is a bijection that satisfies both preconditions of the theorem. □

Corollary 3.2 *For any $n \geq 5$ and any formula $g(i, i+1, j)$ of $\mathsf{CTL}^*\backslash\mathsf{X}$,*

1. $M(n), q_n^0 \models g(0,1,1)$ *iff* $M(5), q_5^0 \models g(0,1,1)$,
2. $M(n), q_n^0 \models g(0,1,2)$ *iff* $M(5), q_5^0 \models g(0,1,2)$,
3. $M(n), q_n^0 \models g(0,1,n-1)$ *iff* $M(5), q_5^0 \models g(0,1,4)$, *and*
4. $M(n), q_n^0 \models g(0,1,j)$ *iff* $M(5), q_5^0 \models g(0,1,3)$, *for* $j \notin \{0,1,2,n-1\}$.

Proof. The corollary follows from Theorems 3.1 and 3.7. □

Theorem 3.8 *Let f be a formula of the form $\bigwedge_{i,j:i\neq j} g(i,i+1,j)$, where $g(i,i+1,j)$ is a $\mathsf{CTL}^*\backslash\mathsf{X}$ formula. Then,* $(\forall n : n \geq 5, M(n), q_n^0 \models f \equiv M(5), q_5^0 \models f)$. □

Proof.

For any $n \geq 5$,

$M(n), q_n^0 \models f$

iff (the initial state is symmetric, lemma 3.4)

$M(n), q_n^0 \models \bigwedge_{jj\neq 0} g(0,1,j)$

iff (splitting up the formula)

$M(n), q_n^0 \models g(0,1) \wedge M(n), q_n^0 \models g(0,1,2) \wedge$

$M(n), q_n^0 \models g(0,1,n-1) \wedge M(n), q_n^0 \models \bigwedge_{jj\notin\{0,1,2,n-1\}} g(0,j)$

iff (by corollary 3.2)

$M(5), q_5^0 \models g(0,1) \wedge M(5), q_5^0 \models g(0,1,2) \wedge$

$M(5), q_5^0 \models g(0,1,4) \wedge M(5), q_5^0 \models g(0,1,3)$

iff (recombining the formula)

$M(5), q_5^0 \models \bigwedge_{jj\neq 0} g(0,1,j)$

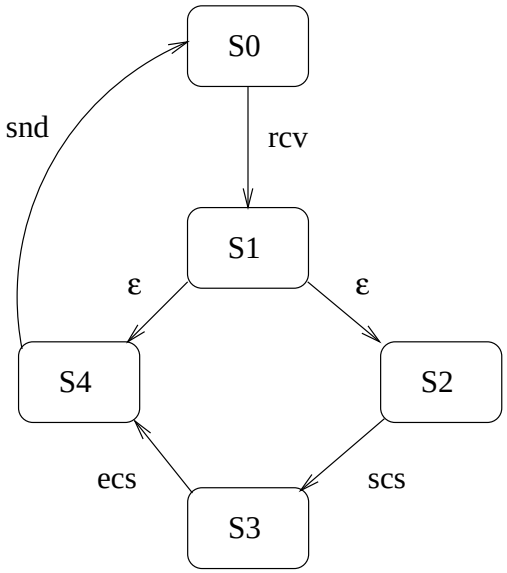


Figure 3.1: Mutual Exclusion Protocol Template.

iff (by lemma 3.4)

$M(5), q_5^0 \models f$

□

3.5 Applications

To illustrate the use of the results, we look at two protocols, one for distributed mutual exclusion protocol [WL 89] and (a slight variation on) Milner's Cycler Protocol [Milner 90].

3.5.1 Distributed Mutual Exclusion

The template process for the protocol in [WL 89] is given in Fig. 3.1. Initially, every process is in state $S0$. Here ϵ is used to indicate a local action. It is easily checked

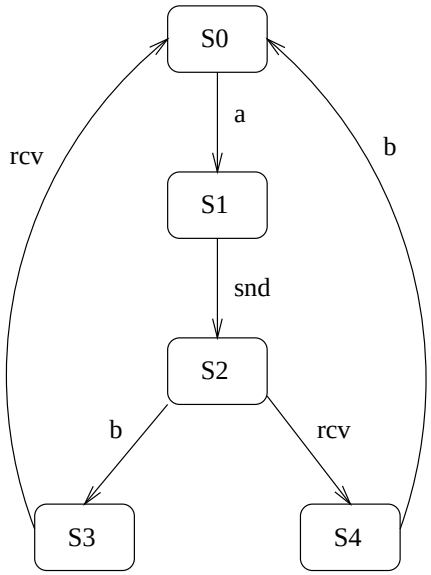


Figure 3.2: Cycler Protocol Template.

that the process description satisfies the conditions for the token passing model (cf. Section 3.3). So properties such as

- If a process requests the token, it will receive it. $\bigwedge_i \mathsf{AG}(S0_i \Rightarrow \mathsf{AF}(S1_i))$
- Every trying process eventually enters its critical section.

  $\bigwedge_i \mathsf{AG}(S0_i \Rightarrow \mathsf{AF}(S2_i \vee S4_i))$

can be checked using a 2-process system by theorem 3.3. Mutual exclusion, i.e $\bigwedge_{i,j:i\neq j} \mathsf{AG}\neg(S3_i \wedge S3_j)$ can be checked in a 4-process system by Theorem 3.6.

3.5.2 Milner's Cycler Protocol

The template process for the protocol in [Milner 90] is given in Fig. 3.2. The initial state is $S3$. The correctness properties are as follows:

- Along every computation, $a_0, a_1 \ldots a_{n-1}$ must be performed cyclically, starting with whichever process is enabled first. The following formula expresses the cyclic order. This has a cutoff of 5, by theorem 3.8.

  $\bigwedge_{i,j:i\neq j} \mathsf{AG}\ (a_i \Rightarrow \mathsf{AXA}((\neg a_i \wedge \neg a_j)\ \mathsf{U}\ a_{i+1}))$
  $\bigwedge_{i,j:i\neq j} \mathsf{AG}\ (\neg(a_i \wedge a_j))$

- Every process performs a_i and b_i alternately. The following formula expresses this property and can be checked in a 2-process system by Theorem 3.3.

  $\bigwedge_i\ \mathsf{AG}((a_i \Rightarrow \neg a_i\ \mathsf{U}\ b_i)\ \wedge (b_i \Rightarrow \neg b_i\ \mathsf{U}\ a_i))$

These formulas are defined with atomic propositions referring to actions instead of states, but it is easily seen that the results apply to such formulas as well, as the block bisimulation of Theorem 3.2 matches computation paths with respect to non-τ actions.

3.6 Undecidability on Rings

Apt and Kozen [AK 86] show that determining if a temporal property holds for all instances of a parameterized network is undecidable, by a reduction from the non-halting problem for Turing machines. The essential idea is to have a ring of size n simulate the computation of a Turing machine for n steps. Suzuki [Suzuki 88] proved a sharper result by showing that the undecidability holds even when the individual programs are finite-state and the ring is unidirectional.

We give a simpler proof of this result, that also delineates sharply the boundary between decidable and undecidable token-ring systems. The proof is by a reduction from the non-halting problem for 2-counter machines. We show that undecidability arises even if the token takes values from a binary domain. The decidability results in this chapter hold for a token with a single value. Thus the information

carrying capacity of the token defines the boundary between decidability and undecidability.

A 2-counter machine [Minsky 61] has four types of instructions : an increment and decrement for each counter, a zero-test, and a halt instruction. The halting problem for 2-counter machines is known to be undecidable. We reduce this problem to the parameterized model-checking problem by using a ring of size n to simulate n steps of a 2-counter machine. W.l.o.g., we suppose that the 2-counter machine ignores its input tape, and initially has both counter values equal to zero.

Each counter is implemented in unary by a single bit in each node of the ring. The bit takes values $\{up, down\}$ with the number of up's in the global state giving the value of the counter. Each node runs a copy of the same program. Initially a single node is chosen nondeterministically, by giving it the token. That node executes the program of the 2-counter machine. A counter increment is done by circulating a token with an increment instruction. The node with a $down$ bit for that counter that first receives this token changes its bit to up, then forwards the token with a "neutral" value. The decrement of a counter value occurs in a similar way (by changing the first up to $down$). The test for zero is done by circulating the token with value "zero"; if it is received by a process which has the corresponding bit set to up, then the token value is changed to "non-zero". If the increment token returns without the neutral value, this implies that the value of that counter is n, so the process simulating the 2-counter machine enters a "dead" state, which has no outgoing transitions.

This simple simulation uses seven values for the token : increment and decrement instructions for each counter, zero/non-zero values, and a neutral value. We can reduce this value domain to a binary one by encoding each of these values in unary. A processes passes one of these values to its neighbor by sending its unary code to the neighbor as a sequence of tokens with value 1. The neighbor accumu-

lates the unary count on receiving such tokens, but retransmits them with value 0, which returns to the code sender, without effecting any change on the local states of the processes in between. This unary transmission is terminated by sending a token with value 0 to the neighbor.

Consider the property $\bigwedge_i(initially\_token_i \Rightarrow \mathsf{AG}\neg halt_i)$, which expresses the condition that the node executing the 2-counter machine program does not reach a halt state. If the 2-counter machine halts, then it must do so in n steps, for some n, so by the simulation, this property is false for a ring of size at least n. If it does not halt, then this property is true for rings of all sizes. Thus, the property is true for all instances iff the machine does not halt. Hence, parameterized verification is undecidable, specifically co-RE.

3.7 Related Work and Conclusions

Among related work, [AK 86, Suzuki 88] show that the problem of automatically checking a specification for every instance of a parameterized system is in general undecidable. Positive results include those of Clarke, Grumberg and Browne [CG 87, BCG 89]; however, their method requires the manual construction of bisimulations or that of a *closure process* which represents computations of an arbitrary number of processes. [KM 89] and [WL 89] introduce the related notion of a *process invariant*. All these methods rely on human ingenuity to manually construct a suitable process closure or invariant. [GS 92] use automata-theoretic methods to construct process closures for processes connected in a complete network, and use them to establish single index properties. Multi-index properties can be indirectly catered for, but the complexity then becomes multi-exponential. In any case, this does not provide an algorithm for ring networks. Vernier [Vernier 93] has developed a model-checking algorithm for a class of parameterized systems, however, the complexity is high.

The closest results are those of Shtadler and Grumberg [SG 89], who use

a network grammar to specify a communication topology, and those of [LSY 94] which deal with ring networks of Petri nets. [SG 89] show that if certain sufficient conditions are satisfied, then every network generated by the grammar satisfies specifications written in linear-time temporal logic. The sufficiency check, however, may require time exponential in the size of an individual process. [LSY 94] contains another (exponential-time) sufficiency test which is used to show that certain parameterized protocols on rings satisfy a single-index linear-time specification. This has recently been extended to show global safety properties in [CGJ 95].

The advantage of the approach presented here is that, to check whether a ring comprised of isomorphic processes satisfies a specification for all instances, it is both necessary and sufficient to check only the small rings of size less than or equal to the cutoff. This result is independent of the actual program executing on each process in the ring, provided that it follows the token-passing discipline. The ring of size equal to the cutoff is analogous to a closure process, but is trivial to construct.

In addition, the result holds even if the transition graph of the template process is not finite, provided that it is finite-branching. If it is finite, then an automated tool such as SMV [McMillan 92], or the Concurrency Workbench [CPS 89] can be used to model-check the desired property for the small ring. This check can be done in time polynomial in the size of a process.

An interesting problem to consider is whether there are conditions under which a similar result holds for systems with multiple-valued tokens. The simplicity of the 2-counter machine simulation suggests that imposing syntactic restrictions will not lead to decidability, unless coupled with some semantic constraints.

3.8 Technical Details

3.8.1 Proof of Theorem 3.2

Theorem 3.2 *Let $I \subseteq [n]$, and $J \subseteq [k]$ be sets of indices such that there is a bijection $h : I \to J$ such that for any $i, j \in I$,*

1. *$i \leq j$ iff $h.i \leq h.j$, and*
2. *$(i : j)_n \neq \emptyset$ iff $(h(i) : h(j))_k \neq \emptyset$.*

Then $M(n)\downarrow_I$ is block equivalent to $M(k)\downarrow_J$.

Proof. Let h^A be the derived renaming function for indexed actions from $M(n)$ defined by $h^A(a_i) = a_{h(i)}$. Let h^P be the derived renaming function for indexed propositions from $M(n)$ defined by $h^P(P_i) = P_{h(i)}$. We define a relation that is a block bisimulation on $M(n)$ and $M(k)$ w.r.t. h^P.

For a state s of $M(i)$, let $tok_i(s)$ be the index of the process that possesses the token in state s. Let R_{nk} be the relation between states of $M(n)$ and $M(k)$, defined as follows:
$s\ R_{nk}\ t$ iff

1. The local states of D_n in s and D_k in t are identical,
2. $h(\lambda(s)\downarrow_I) = \lambda(t)\downarrow_J$,
3. $(\forall i : i \in I : tok_n.s = i \ \equiv\ tok_k.t = h.i)$, and
4. $(\forall i, j : i, j \in I : tok_n(s) \in (i : j)_n \ \equiv\ tok_k(t) \in (h(i) : h(j))_k)$.

The relation R_{nn} is similar but for (2), which is modified to $\lambda(s)\downarrow_I = \lambda(t)\downarrow_I$. Define the relation $\approx$ to be $R_{nn} \cup R_{kk}$. Let $B = R_{nk} \cup R_{kn}$. B is symmetric. It is straightforward to check that $\approx$ is an equivalence relation, and that B is monotonic w.r.t. $\approx$. We show below that B is a block bisimulation w.r.t. $(\approx, h^P)$.

Suppose sBt and s is a state in $M(n)$, and t is a state in $M(k)$. By definition of B, $h^P(\lambda(s)\downarrow_I) = \lambda(t)\downarrow_J$,

1. s is the initial state of $M(n)$.

The only enabled action is the token transfer to some process i. As sBt, t is the initial state of $M(k)$, hence the token transfer action is enabled at t. If $i \in I$, let the transfer from t be to process $h(i)$. Otherwise, there is a pair k, l of indices in I such that $i \in (k : l)$, and $(k : l)$ does not contain an index in I. As sBt, $(h(k) : h(l)) \neq \emptyset$, so let the transfer from t be to some process with index in $(h(k) : h(l))$. It is straightforward to check that the resulting states are related. These single step transitions are the only sequences in $MAXF_{\approx}(s)$.

2. s is not the initial state of $M(n)$.

(a) Let $\sigma; v$ be a sequence starting at s, that is in $MAXF_{\approx}(s)$. Hence $v \not\approx s$, and by the definition of $\approx$, this must be because of either a local move by some process in I, or a token sent from some process in I, or because of a token received by some process in I.

- The action is an internal move, or a send of the token, by process i. As process $h(i)$ has the same local state, the same move can be performed at that process, and the resulting states are related by B. Note that the token is at process i iff it is at process $h(i)$ in t. This creates the matching sequence $\delta; w$.

- The action is a receipt of the token by process i from process $i-1$, for $i \in I$ (arithmetic is modulo n). As sBt, the token in t is not at process $h(i)$. Hence, if process $h(i)-1$ is ready to send the token, the receive action may be executed at process $h(i)$.

  Otherwise, the token is in the interval $(h(j) : h(i))$, for some j. As the process with the token must eventually reach a blocking send action, and processes without the token reach a blocking receive action, there is a sequence of events, δ, starting at state t, such that after δ, the token is at process $h(i) - 1$, which

is blocked on a send action. At this point, the token may be transferred to process $h(i)$ and the system moves to state w. Note that every state along δ is related to t by $\approx$, and $w \not\approx t$; hence, $\delta; w$ is a member of $MAXF_{\approx}(t)$. The final state w is related to v.

(b) The other case, that of infinite sequences from s that are in $MAXI_{\approx}(s)$, does not arise because every process alternates between send and receive actions. This induces the token to move from one process to the next. Thus, the token eventually either reaches or leaves a process in I, which induces a change to a state not equivalent to s.

The proof above is for the case where s is a state in $M(n)$ and t a state in $M(k)$. A similar proof shows the other case. It follows that B is a block bisimulation w.r.t. $(\approx, h^P)$ between $M(n) \downarrow_I$ and $M(k) \downarrow_J$. Notice that this proof shows also that the non-τ actions are matched by B up to h. □

3.8.2 Block Bisimulation

Let A be an LTS, $A = (Q, \Sigma, R, \lambda, L, q^0)$. [BCG 88] define stuttering bisimulation only over finite state systems. Their definition can be generalized to the following :

Definition 3.3 (Stuttering Bisimulation) *A relation B on A is a stuttering bisimulation iff B is symmetric, and for any s, t such that sBt,*

1. *$\lambda(s) = \lambda(t)$, and*
2. *For every fullpath σ starting at s, there is a fullpath δ starting at t such that σ matches δ by B.*

where fullpaths σ and δ match by B iff they can be partitioned into an equal number of segments such that, for each segment number i, every state in the ith segment of σ is related by B to every state in the ith segment of δ. The formal definition of "match" is given below :

Matching paths

Let INC be the set of strictly increasing sequences of natural numbers starting at 0. Precisely, $INC = \{\pi \mid \pi : \mathbf{N} \to \mathbf{N} \wedge \pi(0) = 0 \wedge (\forall i \; i \in \mathbf{N} \Rightarrow \pi(i) < \pi(i+1))\}$. Let σ be a path, and π a member of INC. For $i \in \mathbf{N}$, let $intv(i, \sigma, \pi) = [\pi(i), min(\pi(i+1), length(\sigma)))$. The ith segment of σ w.r.t. π, $seg(i, \sigma, \pi)$, is defined by the sequence of states of σ with indices in $intv(i, \sigma, \pi)$.

Let σ and δ, under partitions π and ξ respectively, correspond w.r.t. stuttering bisimulation B iff they are subdivided into the same number of segments, and any pair of states in segments with the same index are related by B. Precisely, $corr_B((\sigma, \pi), (\delta, \xi)) \equiv (\forall i : i \in \mathbf{N} : intv(i, \sigma, \pi) \neq \emptyset \equiv intv(i, \delta, \xi) \neq \emptyset \wedge (\forall m, n : m \in intv(i, \sigma, \pi) \wedge n \in intv(i, \delta, \xi) : (\sigma_m, \delta_n) \in B))$.

Paths σ and δ match iff there exist partitions that make them correspond. Precisely, $match_B(\sigma, \delta) \equiv (\exists \pi, \xi : \pi, \xi \in INC : corr_B((\sigma, \pi), (\delta, \xi)))$. □

It can be shown, using the Knaster-Tarski theorem with the above definition (as in [Milner 90]), that the greatest stuttering bisimulation exists, and is an equivalence relation. Two states are stuttering equivalent iff there is a stuttering bisimulation that includes the pair.

Theorem 3.3 *Let B be a block bisimulation w.r.t. $(\approx, id)$. Then B is also a stuttering bisimulation.*

Proof. B is symmetric by (1) of Defn. 3.1. Let (s, t) be an arbitrary pair in B. Then, $\lambda(s) = \lambda(t)$ by 3(a). Consider any fullpath σ starting at s. If $\sigma \in MAXI_{\approx}(s)$, then by 3(c) of Defn. 3.1, there is a fullpath δ starting at t such that $\delta \in MAXI_{\approx}(t)$, which is infinite if and only if σ is infinite. So σ and δ may be partitioned into the same number of segments, which by (2), consist of states related by B. If $\sigma \notin MAXI_{\approx}(s)$, there is a maximal prefix σ' of σ with a following state v such that $\sigma; v \in MAXF_{\approx}(s)$. By 3(b), there is a finite path $\delta'; v$ in $MAXF_{\approx}(t)$ such that $(u, v) \in B$. σ' and δ' form a pair of matching segments, and (u, v) the start of a

new pair of corresponding segments. Continuing in this manner, one can inductively define a fullpath δ starting at t, and partitions of σ and δ such that σ and δ match w.r.t. B. Hence, B is a stuttering bisimulation. □

Theorem 3.4 *Let B be the greatest stuttering bisimulation. Then B is a block bisimulation w.r.t. (B, id).*

Proof. The greatest stuttering bisimulation is an equivalence relation; hence conditions (1) and (2) of the block bisimulation definition are satisfied. Let (s, t) be an arbitrary pair in B. As $\lambda(s) = \lambda(t)$, by (1) of Defn. 3.3, it follows that $id(\lambda(s)) = \lambda(t)$.

Consider a path σ in $MAXI_B(s)$. As σ is a fullpath, by (2) of Defn. 3.3, there is a fullpath δ starting at t such that σ matches δ w.r.t. B. This implies that δ is in $MAXI_B(t)$ as B is an equivalence, and δ satisfies condition 3(c) of Defn. 3.1.

Consider a finite path $\sigma; v$ in $MAXF_B(s)$. Let γ be a fullpath starting at v. As $\sigma; \gamma$ is a fullpath starting at s, by 2(b) of Defn. 3.3, there is a matching fullpath δ starting at t. As $\neg(vBs)$, $\neg(vBt)$ holds. Let i be the first position along δ such that $vB\delta_i$ holds. Hence, $i > 0$, and $\delta[0..i)$ consists of states related to t by B. So $\delta[0..i]$ is a member of $MAXF_B(t)$, and satisfies condition 3(b) of Defn. 3.1. Hence, B is a block bisimulation w.r.t. (B, id). □

3.8.3 Proof of Theorem 3.1

Theorem 3.1 *Let B be a block bisimulation on LTS A w.r.t. $(\approx, h)$. If $(s, t) \in B$, then for any formula f of $\mathsf{CTL}^*\backslash\mathsf{X}$, $A, s \models f$ iff $A, t \models h(f)$.* □

Proof.

The proof is by structural induction on formulas of $\mathsf{CTL}^*\backslash\mathsf{X}$.

Basis : $f \in AP$.

$M, s \models f$

iff (definition)

$f \in \lambda(s)$

iff (h is a bijection on AP)

$h(f) \in h(\lambda(s))$

iff (condition 3(a) of Defn. 3.1)

$h(f) \in \lambda(t)$

iff (definition)

$M, t \models h(f)$

Inductive case : If f has the form $g_0 \wedge g_1$ or $\neg g$, then the proof follows directly from the inductive hypothesis. Consider the case where f has the form $\mathsf{E}g$, for a path formula g.

$M, s \models \mathsf{E}g$

iff (definition)

$(\exists \sigma : fp(\sigma) \wedge \sigma_0 = s : M, \sigma \models g)$

iff (by following proof)

$(\exists \delta : fp(\delta) \wedge \delta_0 = t : M, \delta \models h(g))$

iff (definition)

$M, t \models \mathsf{E}h(g)$

iff (definition)

$M, t \models h(f)$

To justify the second step of the proof above, suppose σ is a fullpath starting at s. As $(s, t) \in B$, by Defn. 3.1 and a simple inductive argument involving 3(b) and 3(c), there is a fullpath δ starting at t such that σ and δ may be partitioned into the same number of non-empty segments, where the initial states of corresponding segments are related by B, and states within each segment are in the same $\approx$-class. Since initial states of corresponding segments are related by B, from the

monotonicity of B w.r.t. $\approx$ (property (2)), every pair of states from corresponding segments is in B. Hence, by the inductive claim, states related by B agree on the truth value of sub-formulas of g.

Let σ and δ be such that they can be partitioned as described above. The nested inductive claim is that for any path formula g such that states related by B agree on the truth values of sub-formulas of g modified by h, $M, \sigma \models g$ iff $M, \delta \models h(g)$.

Basis : g is a state formula. As $\sigma_0 B \delta_0$, the claim follows from the nested induction hypothesis.

Inductive case : The proof when g has the form $g_0 \wedge g_1$ or $\neg g_0$ follows directly from the induction hypothesis. Consider the case where g has the form $g_0 \mathsf{U} g_1$.

$M, \sigma \models g_0 \mathsf{U} g_1$

iff (definition)

$(\exists i : M, \sigma^i \models g_1 \wedge (\forall j : j < i : M, \sigma^j \models g_0))$

iff (σ and δ have matching partitions; inductive hypothesis)

$(\exists k : M, \delta^k \models h(g_1) \wedge (\forall l : l < k : M, \sigma^l \models h(g_0)))$

iff (definition)

$M, \delta \models h(g_0) \mathsf{U} h(g_1)$

iff (definition)

$M, \delta \models h(g_0 \mathsf{U} g_1)$

The second step in the proof follows from the observation that since the paths match, for any position i on one path, in the first position k on the corresponding segment of the other path is such that the suffix paths starting at i and k match, and for each suffix path starting at position $l < k$, there is a matching path from some position $j < i$. Hence, the induction hypothesis can be applied to these suffixes. $\square$

Chapter 4

Verifiying Parameterized Synchronous Systems

4.1 Introduction

In this chapter, we consider a class of systems where similar processes execute *synchronously*. Such systems arise typically in hardware; for instance, a bus with some distinguished processes and an arbitrary number of identical processors connected to it. For these systems we show decidability of the parameterized model-checking problem (PMCP). This decidability result has a different flavor than the one in the previous chapter as the cutoff size is strongly dependent on the formula while in the previous result, the cutoff is dependent only on the form of quantification and not the formula itself. Previous work on the PMCP is oriented, with the exception of [KM 89], towards the interleaving composition model. [GS 92] and [EN 95] provide algorithms for some classes of parameterized systems, while other techniques [Lubachevsky 84, SG 89, KM 89, WL 89, Vernier 93, CGJ 95] have only a limited degree of automation.

The approach presented here is fully automated. The class of synchronous

systems is specified by a control and a user process. Each instance of the system consists of a single copy of the control and an arbitrary number of copies of the user processes. The system is thus parameterized by the number of user processes. Processes are specified by finite state programs expressed as transition graphs, where guards on each transition may check both the local control process state as well as certain predicates on the global state. The correctness properties are expressed in an indexed propositional branching temporal logic, and are of the following types:

1. Over the control process : formulas of the form $\mathsf{A}h$ and $\mathsf{E}h$, where h is a linear-time formula with atomic propositions over control process states, and some global predicates.

2. Over all user processes: $\bigwedge_i \mathsf{A}h(i)$ and $\bigwedge_i \mathsf{E}h(i)$, where $h(i)$ is a linear-time formula with atomic propositions over control process states, and over user process states indexed with i, along with some global predicates.

3. Over every distinct pair of user processes : $\bigwedge_{i,j:i\neq j} \mathsf{A}h(i,j)$ and $\bigwedge_{i,j:i\neq j} \mathsf{E}h(i,j)$, where $h(i,j)$ is a linear-time formula with atomic propositions over control process states, and over user process states indexed with either i or j, along with some global predicates.

We show that the PMCP for the first type of formulas is decidable for this class of systems, and is PSPACE-complete . This decidability result is based on constructing a *finite* abstract graph in which *every* computation of *every* size instance of the system is represented by some path in the graph. However, the abstract graph may have "bad" paths that do not correspond to computations of any size instance. The heart of the algorithm is a method for identifying good paths in the abstract graph. The space used by the algorithm is polynomial in the size of the control and user processes. We prove that this is optimal in a complexity-theoretic sense by showing that the problem is PSPACE-hard , even for invariance properties. Using

symmetry arguments, the verification task for the other types of formulas reduces to a verification task of the first type for a modified system. The algorithm for safety properties has been implemented using the SMV verification system [McMillan 92] and used for the verification of an industrial bus protocol [SAE 92]. Our initial experimental results indicate that the algorithm should be useful in practice. Chapter 5 describes this verification in detail.

Section 4.2 defines the system model and the logic used for expressing correctness properties. Section 4.3 describes the abstract graph representation, and Section 4.4 the algorithm for formulas of type (1). Section 4.5 shows the reduction of the PMCP for formulas of types (2) and (3) to the PMCP for formulas of type (1). The implementation of the algorithm, and its application to the SAE-J1850 bus arbitration protocol is described briefly in Section 4.6, and in detail in the following chapter. Section 4.7 generalizes these results to the case of properties that are "almost universal"; i.e., they hold for all but a finite number of processes. In Section 4.8 we present complexity-theoretic lower bounds on the problem. Section 4.9 concludes the chapter with a discussion of related work.

4.2 The system model and logic

We refer to the collection of system instances formed by control process C and copies of a generic user process U as a (C, U) *family*. The control and user processes are specified as finite-state transition graphs. For such a graph A, let S_A denote its set of states, R_A its transition relation, and ι_A its initial state. For simplicity, we consider only graphs with a single initial state; the results carry over with only minor modifications for graphs with a set of initial states. The transition relation of these graphs is total. It is often convenient to specify only some of the transitions out of a state; but, in such a situation, there is an implied self-loop on that state which has as its guard the negation of the disjunction of the guards on the specified

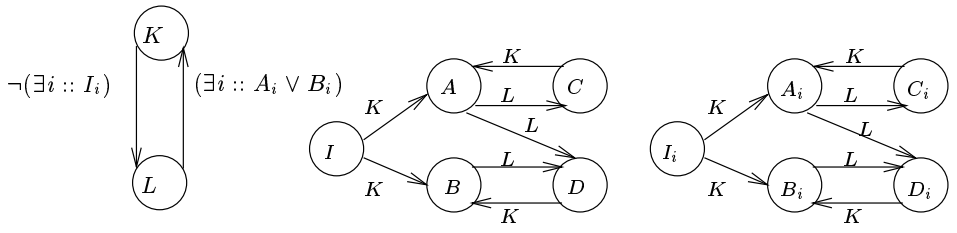


Figure 4.1: A Control-User System : Control, User and the ith User Processes.

transitions.

The system *instance* of size n is a synchronous parallel composition of C with n copies of process U, and is denoted as $C \parallel U^n = C \parallel U_1 \parallel U_2 \parallel \ldots \parallel U_n$. U_i is the ith copy of U, which is obtained from U by uniformly subscripting the states of U with i as shown in the example system in Fig. 4.1. In this example, C has initial state K, and U has initial state I. Atomic propositions are identified with state names.

Thus, for any i, j, U_i and U_j are isomorphic up to re-indexing. Transitions in both C and U_i are labeled with *guards*. Every guard is a boolean combination of *global predicates* of the form $(\exists i :: \mathcal{E}(i))$, where $\mathcal{E}(i)$ is a boolean expression formed from atomic propositions over the states of C and U_i. There are two interesting special cases : (a) The guards in U_i involve only propositions over states of C. The control process may then be viewed as controlling the execution of the user processes. (b) The control process is a copy of the user process, and can be written as U_0. Then $C \parallel U^n$ is isomorphic to U^{n+1}. Our method applies in general, but often finds interesting application in these special cases.

$\mathcal{G}_n$ denotes the global state transition graph of the instance of size n. A state s of $\mathcal{G}_n$ is written as an $(n+1)$-tuple, where $s(0)$ is the local state of C and $s(i)$ is the local state of U_i (for $i \in [1..n]$). The initial state of $\mathcal{G}_n$ is $(\iota_C, (\iota_U)_1, \ldots, (\iota_U)_n)$.

A transition (s,t) is in $\mathcal{G}_n$ iff

1. A transition of C from $s(0)$ to $t(0)$ is enabled in s, and
2. For all $i \in [1..n]$, a transition of U_i from $s(i)$ to $t(i)$ is enabled in s.

where a transition in a process is said to be enabled in a global state iff the corresponding guard is true when evaluated in that global state. $\mathcal{G}_n, s \models (\exists i :: \mathcal{E}(i))$ iff for some $k \in [1..n]$, $\mathcal{E}(k)$ is true given the propositions that hold at $s(0)$ (the control state), and $s(k)$ (the state of process U_k). Boolean operators are handled in the standard manner. For a global state s, and state $a \in S_U$, we let $\#a(s) = |\{i \mid i \in [1..n] \wedge s(i) = a_i\}|$ ($\#a(s)$ is the number of user processes with local state a of the generic user process).

LTL is the standard propositional linear temporal logic built up from atomic propositions, boolean connectives, and temporal operators G (always), F (sometime), X (next time), and U (until) [Pnueli 77]. CTL* is a branching temporal logic which extends LTL by allowing the path quantifiers A (for all fullpaths) and E (for some fullpath). Many interesting correctness properties of parameterized systems can be expressed in one of the following forms:

1. Over the control process : formulas of the form $\mathsf{A}h$ and $\mathsf{E}h$, where h is a linear-time formula with atomic propositions over control process states,
2. Over all user processes: $\bigwedge_i \mathsf{A}h(i)$, and $\bigwedge_i \mathsf{E}h(i)$, where $h(i)$ is a linear-time formula with global predicates as the atomic propositions.
3. Over all distinct pairs of user processes : $\bigwedge_{i,j:i\neq j} \mathsf{A}h(i,j)$, and $\bigwedge_{i,j:i\neq j} \mathsf{E}h(i,j)$, where $h(i,j)$ is a linear-time formula with global predicates as the atomic propositions.

The formal semantics of these logics is defined in the usual way [Emerson 90, BCG 88, ES 95]. We write $M, s \models f$ to mean that formula f is true in struc-

ture M at state s. In the previous chapter, we considered formulas without the next-time operator, X. The use of the next-time operator is not problematic here as the system model is that of synchronous execution. Another difference with the results in the previous chapter is that those hold for quantified CTL*\X formulas, while these are for CTL* formulas with a single outermost path quantifier.

4.3 The abstract model

For a given (C, U) family, we construct an abstract process $\mathcal{A}$ that includes all computations of every size instance of the family. An abstract state (c, S) *represents* any concrete state where the control process is in local state c, and only the local user states in the set S are occupied by some process. Transitions from a state (c, S) in the abstract graph represent transitions enabled from the global states that are represented by (c, S). Each such transition has a label which indicates the moves of processes between local states. Each label is a non-empty relation on S_U.

Formally, let $\Lambda = \mathcal{P}(S_U \times S_U)\backslash\{\emptyset\}$ be the set of edge labels. $\mathcal{A}$ is defined by a labeled transition graph, where

1. $S_{\mathcal{A}} = S_C \times (\mathcal{P}(S_U)\backslash\{\emptyset\})$ is the set of states,
2. $R_{\mathcal{A}} \subseteq S_{\mathcal{A}} \times \Lambda \times S_{\mathcal{A}}$ is the set of transitions,
3. $\iota_{\mathcal{A}} = (\iota_C, \{\iota_U\})$ is the initial state.

To make the correspondence between global and abstract states precise, we define families of abstraction functions $\{\phi_i\}$, $\{\psi_i\}$, where $\phi_n : S_{\mathcal{G}_n} \to S_{\mathcal{A}}$ maps a concrete state in the instance of size n to an abstract state, and $\psi_n : S_{\mathcal{G}_n} \times S_{\mathcal{G}_n} \to \Lambda$ maps a concrete transition in the instance of size n to a transition label. For a state $s \in S_{\mathcal{G}_n}$, $\phi_n(s) = (s(0), \{a \mid \#a(s) > 0\})$. For a pair (s, t), $\psi_n(s, t) = \{(a, b) \mid (\exists i : i \in [1..n] : s(i) = a_i \wedge t(i) = b_i)\}$. The abstract state (c, S) *represents* every s in $\mathcal{G}_n$ for which $(c, S) = \phi_n(s)$.

We postulate a fixed total ordering of the user states (i.e., elements of S_U). For any subset S of S_U, let $ord(S)$ denote the vector obtained by sorting S under this ordering. For a guard g, and state (c, S) of $\mathcal{A}$, we define $(c, S) \,||- \, g$ as $\mathcal{G}_{|S|}, (c, ord(S)) \models g$.

Proposition 4.1 *Let s be a state in $\mathcal{G}_n$ and t a state in $\mathcal{G}_m$ such that $\phi_n(s) = \phi_n(t)$. For any guard g, $\mathcal{G}_n, s \models g$ iff $\mathcal{G}_m, t \models g$.*

Proof. A guard expression is a boolean combination of global predicates which are of the form $(\exists i :: \mathcal{E}(i))$. We show the equivalence when the guard expression is a global predicate. The proof for boolean operations is straightforward.

$\mathcal{G}_n, s \models (\exists i :: \mathcal{E}(i))$

iff (definition)

$(\exists i : i \in [1..n] : \mathcal{E}(s(0), s(i)))$

iff ($\phi_n(s) = \phi_n(t)$ implies $s(0) = t(0)$ and user states in s and t are identical)

$(\exists j : j \in [1..m] : \mathcal{E}(t(0), t(j)))$

iff (definition)

$\mathcal{G}_m, t \models (\exists i :: \mathcal{E}(i))$

Corollary 4.1 *For any n, and any $s \in \mathcal{G}_n$, if $(c, S) = \phi_n(s)$, then for every guard expression g, $\mathcal{G}_n, s \models g$ iff $(c, S) \,||- \, g$.*

Proof.

$(c, S) \,||- \, g$

iff (definition)

$\mathcal{G}_{|S|}, (c, ord(S)) \models g$

iff ($\phi_{|S|}((c, ord(S))) = \phi_n(s)$; Lemma 4.1)

$\mathcal{G}_n, s \models g$

□

The set of transitions is defined as follows: A tuple $((c,S),X,(d,T)) \in R_{\mathcal{A}}$ iff

1. $(\exists p : c \xrightarrow{p} d \in R_C : (c,S) \,||- \, p)$ (A transition from c to d is enabled for the control process),

2. $(\forall a,b : (a,b) \in X : a \in S \wedge b \in T \wedge (\exists q : a \xrightarrow{q} b \in R_U : (c,S) \,||- \, q))$. (For every pair (a,b) in X, there is an enabled transition from a to b in the user process).

3. X is total on S, and X^{-1} is total on T, (Every state in S has a successor in T, and every state in T has a predecessor in S).

Definition 4.1 (Abstract Path) *A path in $\mathcal{A}$ is a sequence starting at a state, with alternating states and transition labels such that for every $s,t \in S_{\mathcal{A}}$ and $X \in \Lambda$, sXt occurs in the sequence only if $(s,X,t) \in R_{\mathcal{A}}$.* □

Define a family of functions $\{\gamma_i\}$ such that γ_n maps from paths in $\mathcal{G}_n$ to paths in $\mathcal{A}$ by $(\gamma_n(\sigma))_{2i} = \phi_n(\sigma_i)$ and $(\gamma_n(\sigma))_{2i+1} = \psi_n(\sigma_i, \sigma_{i+1})$ for all $i \in \mathbf{N}$. The following proposition has a straightforward proof, by induction on the length of the path. The complete proof is given in Section 4.10.

Proposition 4.2 *For every path σ in $\mathcal{G}_n$, $\gamma_n(\sigma)$ is a path in $\mathcal{A}$.*

It follows from Proposition 4.2 that if $\mathcal{A}$ satisfies a linear temporal formula over all paths, then so does every size instance of the family. However, if the formula is false for some path in $\mathcal{A}$, it does not follow that it is false for some instance, as the path may not have a corresponding path in some instance. Paths which have this property are called "good".

Definition 4.2 (Good Paths) *A path ρ in $\mathcal{A}$ is good iff $(\exists n, \sigma : \sigma \in \mathcal{G}_n : \gamma_n(\sigma) = \rho)$.* □

Definition 4.3 (State Covering) *A state t in $\mathcal{G}_m$ covers a state s in $\mathcal{G}_n$ iff $\phi_n(s) = \phi_m(t)$, and for every $a \in U$, $\#a(t) \geq \#a(s)$.*

Definition 4.4 (Path Covering) *A path δ in $\mathcal{G}_m$ covers a path σ in $\mathcal{G}_n$ iff $\gamma_m(\delta) = \gamma_n(\sigma)$, and for every $k \in \mathbf{N}$, $a \in U$, $\#a(\delta_k) \geq \#a(\sigma_k)$.* □

Lemma 4.1 *State covering is a simulation relation.*

Proof. Let s, t be global states in $\mathcal{G}_n$ and $\mathcal{G}_m$ respectively, such that t covers s, and let $s \rightarrow u$. Since $\phi_n(s) = \phi_m(t)$, by Proposition 4.1, guards in s and t evaluate to the same value. Hence, the transitions that are enabled at s are also enabled at t. For user states a, b, let k_{ab} be the number of processes that move from local state a to local state b in the transition from s to u. Let $x_{ab} = k_{ab} + \Delta_{ab}$, where $\Delta_{ab} = \#a(t) - \#a(s)$, if b is the first (w.r.t. the total order on user states) state for which $k_{ab} > 0$, otherwise let $\Delta_{ab} = 0$. Clearly, $x_{ab} \geq k_{ab}$, and $x_{ab} = 0$ iff $k_{ab} = 0$. Let v be any successor of t on the same control transition as that from s to u, and for which x_{ab} of the processes in state a in t move to state b in v. As for any a, $\#a(v) = \Sigma_b x_{ba}$, it follows that $\#a(v) \geq \#a(u)$ and $\#a(v) = 0$ iff $\#a(u) = 0$. Thus, v covers u, and is a successor of t by the same transitions that change s to u. □

From this lemma, we can conclude that if σ is a path starting at s, and t covers s, there is a path δ starting at t such that δ covers σ. It also follows that every path in $\mathcal{G}_n$ has a covering path in $\mathcal{G}_m$, for $m \geq n$, as one can find a covering state in $\mathcal{G}_m$ for the first state of the path. The following lemma shows that state covering is also a *backwards* simulation.

Lemma 4.2 *State covering is a backwards simulation relation.*

Proof. Let u, v be global states in $\mathcal{G}_n$ and $\mathcal{G}_m$ respectively such that v covers u. Let s be such that $s \rightarrow u$, i.e., s is a predecessor of u. We will show that there is a predecessor t of v such that t covers s. Construct t as follows:

The control state of t is the control state of s. For local states a, b, let k_{ab} be the number of processes that move from local state a to local state b in the transition from s to u, and let x_{ab} be the corresponding number (to be determined) for the move from t to v.

Let $x_{ab} = k_{ab} + \Delta_{ab}$. The added number, Δ_{ab}, is $\#b(v) - \#b(u)$, if a is the smallest (in the total order on user states) for which $k_{ab} > 0$, otherwise it is 0. It follows that $x_{ab} \geq k_{ab}$, and $x_{ab} = 0$ iff $k_{ab} = 0$. Let t be a state such that $t(0) = s(0)$, and $\Sigma_b x_{ab}$ of the user processes are in local state a. Clearly, $\phi_m(t) = \phi_n(s)$; hence, from Proposition 4.1, the local transitions enabled at s are also enabled at t. Let x_{ab} processes change from local state a in t to local state b in v by the same transition as in the change from s to u. Hence, t covers s, and is a predecessor of v by the same global transition that changes s to u.

□

The structural lemmas above have the following important consequence:

Theorem 4.1 *Every finite path of $\mathcal{A}$ is good.*

Proof. The proof is by induction on the number of states in the path. Suppose the path is a single state s. Let $s = (c, S)$. For the state $r = (c, ord(S))$, $\phi_{|S|}(r) = s$. So the claim is true of paths with one state.

Assume inductively that the claim holds for all paths with at most m states, for $m \geq 1$. Let ρ be a path with $m + 1$ states. Then, ρ may be written as $sX\rho'$, where ρ' has length m. By the inductive hypothesis, for some n', there is a path $\delta' \in \mathcal{G}_{n'}$ such that $\gamma_{n'}(\delta') = \rho'$. Let u be the first state in δ', and let $s = (c, S)$.

Let w be the state where $w(0) = c$, and for every local user state a, $\#a(w) = |\{b|(a, b) \in X\}|$. Let k be the number of processes in w. As $\phi_k(w) = s$, by Corollary 4.1, the guards enabled at s are also enabled at w. Hence there is a successor x of w such that $x(0) = u(0)$ and for any $(a, b) \in X$, a single process moves from state a in w to state b in x. So, $\phi_k(x) = \phi_k(u)$. Let y be a state such that $y(0) = u(0)$, and

for every a, $\#a(y) = max(\#a(u), \#a(x))$. Thus, y covers both x and u. By Lemma 4.2, y has a predecessor z that covers w, and $\psi_n(z, y) = X$. By Lemma 4.1, there is a path δ from y that covers δ'. Hence, the path $z;\delta$ is a path in an instance that maps to ρ. □

4.4 Verifying properties of the control process

The properties of the control process are of the form $\mathsf{A}h$ or $\mathsf{E}h$, where h is a linear-time temporal formula with atomic propositions that are either global predicates of the form $(\exists i :: \mathcal{E}(i))$, or are propositions over the states of C. To model-check such a property, we follow the automata-theoretic approach of [VW 86] : To determine if $M, \iota_M \models \mathsf{E}h$, construct a Büchi automaton $\mathcal{B}_h$ for h, and check that the language of the product Büchi automaton of M and $\mathcal{B}_h$ is non-empty (cf. [LP 85]). $\mathcal{B}$ accepts a computation σ labeled with propositions over states of C iff there is a run of $\mathcal{B}$ on σ such that a "green" (i.e., accepting) state of $\mathcal{B}$ is entered infinitely often. The check for the property $\mathsf{A}h$ is easily reduced to that for the earlier case by noting that $M, \iota_M \models \mathsf{A}h$ iff $M, \iota_M \not\models \mathsf{E}\neg h$.

Definition 4.5 (Universal property) *A property is universal iff it is true for every size instance of the parameterized system.*

To determine if $\mathsf{A}h$ is universal, we model-check it over the abstract graph, by constructing a Büchi automaton $\mathcal{B}$ for $\neg h$, and forming the product Büchi automaton $\mathcal{M}$ of $\mathcal{A}$ and $\mathcal{B}$. An *accepting* path in $\mathcal{M}$ is one which starts in an initial state, and along which a green state occurs infinitely often. For a path δ in $\mathcal{M}$, let $\delta_{\mathcal{A}}$ be its projection on $\mathcal{A}$. A path in $\mathcal{M}$ is *good* iff its projection on $\mathcal{A}$ is a good path in $\mathcal{A}$.

Theorem 4.2 *Formula $\mathsf{A}h$ is not universal iff there is an accepting good path in $\mathcal{M}$.*

Proof. Suppose δ is an accepting good path in $\mathcal{M}$. As $\delta_{\mathcal{A}}$ is good, for some n, there is a path σ in $\mathcal{G}_n$ such that $\gamma_n(\rho) = \delta_{\mathcal{A}}$. By the definition of γ_n, σ matches δ_A on both the valuation of global predicates and the states of C, and is hence accepted by $\mathcal{B}$. Therefore, $\mathsf{A}h$ is false in $\mathcal{G}_n$ and thus is not universal.

In the other direction, if $\mathsf{A}h$ is not universal, then for some n, there is a path σ in $\mathcal{G}_n$ from the initial state that is accepted by $\mathcal{B}$. From Lemma 4.2, $\gamma_n(\sigma)$ is a path in $\mathcal{A}$, which is good by construction. By the definition of γ_n, σ and $\gamma_n(\sigma)$ match on both the valuation of global predicates and the states of C. Hence, there is a run of $\mathcal{B}$ on $\gamma_n(\sigma)$ that forms an accepting good path in $\mathcal{M}$. □

4.4.1 Model-Checking Safety Properties

For safety properties, we can replace Büchi automata with automata over finite strings. The acceptance condition is modified so that a *finite* path is accepting iff it ends in an accepting state of the automaton. From theorem 4.1, we can conclude that such a path exists in an instance iff an accepting state is reachable in the abstract graph $\mathcal{M}$. The reachability test can be performed in space logarithmic in the size of $\mathcal{M}$; i.e., in polynomial space in the size of the user process. The check for liveness properties requires the use of Büchi acceptance conditions, and the algorithms presented in the next part of this section.

4.4.2 Model-Checking Liveness Properties

For liveness properties, we have to check if an infinite path in $\mathcal{M}$ is accepting and good. The following lemmas provide the basis for a PSPACE algorithm to check universality. For a cycle δ in $\mathcal{M}$, we say that δ is good iff the infinite path δ^ω is good.

Lemma 4.3 *There is an accepting good path in $\mathcal{M}$ iff there are finite paths α and β in $\mathcal{M}$, such that*

1. *α is a path from the initial state to a green state s, and*

2. *β is a good cycle starting at s.* □

Proof.

(LHS $\Rightarrow$ RHS) Let ρ be an accepting good path in $\mathcal{M}$. Hence, for some n, there is a path σ in $\mathcal{G}_n$ such that $\gamma_n(\sigma) = \rho_{\mathcal{A}}$. As ρ is accepted by $\mathcal{B}$, so is σ, hence there exists a path δ_1 to a green state, and a cycle δ_2 from that green state in $\mathcal{G}_n \times \mathcal{B}$, such that $\delta_1 \circ \delta_2^\omega$ is an accepting run of $\mathcal{B}$ on $\mathcal{G}_n$ [1]. Let α be the path in $\mathcal{M}$ that δ_1 maps to (by extending the mappings ϕ and ψ to include automaton states), and β the cycle in $\mathcal{M}$ that δ_2 maps to. α and β satisfy the conditions above.

(RHS $\Rightarrow$ LHS) By Lemma 4.1, $\alpha_{\mathcal{A}}$ is good. Hence, for some k, there is a path σ_1 in $\mathcal{G}_k$ such that $\gamma_k(\sigma_1) = \alpha_{\mathcal{A}}$. As β is a good cycle, for some l, there is an infinite path σ_2 in $\mathcal{G}_l$ such that $\gamma_l(\sigma_2) = \beta_{\mathcal{A}}^\omega$. The rest of the proof uses the simulation lemmas proved earlier to "patch together" these paths, which are in different size structures.

Let u be the last state on σ_1, and v be the first state on σ_2. Define the state w so that w has the same control state as v (and hence, as u), and for every user local state a, $\#a(w) = max(\#a(u), \#a(v))$. Hence, w covers both u and v. By Lemma 4.1, there is a path δ_2 from w that covers σ_2, and by Lemma 4.2, there is a path δ_1 to w that covers σ_1. As δ_1 starts at an initial state of $\mathcal{G}_k$, σ_1 starts at an initial state of $\mathcal{G}_n$, where n be the number of processes in w. Let $\delta = \delta_1 \circ \delta_2$. As δ_2 covers σ_2, it has the same sequence of control states, so there is an accepting run of $\mathcal{B}$ on δ. Hence, $\gamma_n(\delta)$ is an accepting good path in $\mathcal{M}$. □

Intuitively, a cycle in $\mathcal{M}$ is good if, starting at some global state which maps to a state in the cycle, there is no transition in that cycle that causes the count of processes in a specific local state to be "drained" (i.e., strictly decreased) as the sequence of transitions along the cycle is repeatedly executed. For example, a self-

[1] $\circ$ concatenates two strings deleting a copy of a common end state, if any. e.g., $ba \circ ac = bac$, while $ba; ac = baac$.

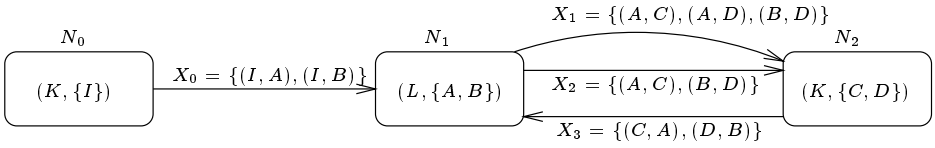


Figure 4.2: A portion of the Abstract Graph for the example system.

loop with the transition label $\{(a, b)\}$ $(a \neq b)$ forces a transfer of at least one process in local state a to local state b, so it decreases the count of processes in state a with every execution of the transition, while a self-loop with label $\{(a, b), (b, a)\}$ does not. Notice that in the latter case, there is a cycle $a \rightarrow b \rightarrow a$ in the graph of transition label. This presence of cycles in transition labels that do not cause draining is the intuition behind the characterization of good cycles of $\mathcal{M}$ that follows.

To determine if such loops are present in a cycle of $\mathcal{M}$, we resolve it into a "threaded graph" (cf. [ES 95]) which shows explicitly which local user state in an abstract state is driven into which other local user state in the next abstract state. This information is obtained from the transition label. The threaded graph is defined below:

Definition 4.6 (Threaded Graph) *Let δ be a finite path in $\mathcal{M}$ with m states, $m > 1$. Let the ith state of δ be called s_{i-1}, and the ith transition be called X_{i-1}. For a state $s = ((c, S), u)$ of $\mathcal{M}$ (u is the automaton state), let $ustates(s) = S$. Define H_δ to be the following graph :*

$V(H_\delta) = \{(x, i) \mid i \in [0..m-1] \wedge x \in ustates(s_i)\}$

$E(H_\delta) = \{((x, i), (y, i+1)) \mid i \in [0..m-1) \wedge (x, y) \in X_i\}$

If δ is a cycle, then $s_0 = s_{m-1}$. Define T_δ to be the "threaded" graph where $V(T_\delta) = V(H_\delta)$, and $E(T_\delta) = E(H_\delta) \cup \{((x, m-1), (x, 0)) \mid x \in ustates(s_0)\}$.

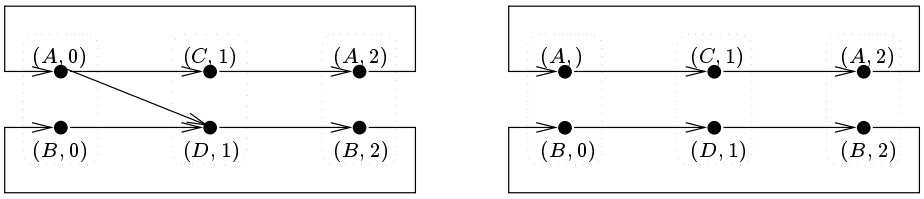


Figure 4.3: Threaded graphs for the cycles $X_1; X_3$ and $X_2; X_3$ respectively.

Note that the threaded graph has a total transition relation, as both the control and user processes have total transition relations.

A graph is *empty* iff its edge set is empty. For any directed graph G, let *sccd(G)* be the graph representing the decomposition of G into its maximal strongly connected components (scc's).

$V(sccd(G)) = \{ C \mid C \textit{ is a maximal strongly connected component of } G\}$

$E(sccd(G)) = \{(C, D) \mid (\exists s : s \in C \wedge t \in D : (s, t) \in E(G))\}$

We refer to vertices of *sccd(G)* as max-scc's. It is a fact that *sccd(G)* is acyclic for any graph G. A max-scc C is said to be *above* another max-scc D iff there is a path in *sccd(G)* from C to D. □

Fig. 4.3 shows the threaded graphs for the cycles $N_1 \stackrel{X_1}{\rightarrow} N_2 \stackrel{X_3}{\rightarrow} N_1$ and $N_1 \stackrel{X_2}{\rightarrow} N_2 \stackrel{X_3}{\rightarrow} N_1$ in Fig. 4.2. From Theorem 4.3, one can conclude that the first cycle is bad while the second is good.

Theorem 4.3 *δ is a good cycle in $\mathcal{M}$ iff $sccd(T_\delta)$ is empty.*

Proof Sketch.

(LHS $\Rightarrow$ RHS): Let m be the number of states in δ. Suppose that $sccd(T_\delta)$ is not empty. Hence, there are max-scc's C and D such that some pair of vertices (x, i) in C and (y, j) in D is connected in T_δ. For any n, consider an infinite path

σ in $\mathcal{G}_n$ such that $\gamma_n(\sigma) = \delta_{\mathcal{A}}^{\omega}$. We say that process l in component F at σ_k iff $(a, k \ mod \ m) \in F$, where $\sigma_k(l) = a_l$.

Starting with the ith transition in σ, at every mth successive transition, at least one of the processes in C, say one with index l, must change its local state from x_l to y_l. Thus, the count of processes in components above D decreases at each such step. As the max-scc decomposition is acyclic, this number cannot increase in subsequent steps. As σ is infinite, eventually the number of processes in components above D must become negative, which is impossible. Hence, no infinite path can map to δ.

(RHS $\Rightarrow$ LHS): Suppose that $sccd(T_\delta)$ is empty. For each max-scc of T_δ, construct a tour in T_δ that includes each edge in that component at least once. For each user state a, let occ_a be the number of occurrences of the vertex $(a, 0)$ in the tour for its max-scc. Let $n = \Sigma_a occ_a$. We will construct a path σ in $\mathcal{G}_n$ such that $\gamma_n(\sigma) = \delta_{\mathcal{A}}^{\omega}$. The idea behind the construction is to allot a set of processes for each constructed tour, and to ensure that each transition of a process is along the tour that it is allotted to.

The inductive assertion is that at the ith step ($i < m$), a path σ has been constructed such that $\gamma_n(\sigma)$ is the prefix of $\delta_{\mathcal{A}}$ up to the ith state, and if s is the last state of σ, then $\#a(s)$ is the number of occurrences of (a, i) in the set of constructed tours. This is possible for $i = 0$ from the definition of n above. Suppose that the ith transition on $\delta_{\mathcal{A}}$ is labeled by X, and that inductively, the assertion is true for the ith state on $\delta_{\mathcal{A}}$. For each pair (a, b) in X, there is an edge from (a, i) to $(b, i+1)$ in the threaded graph, which is therefore part of some tour. By the inductive assumption, there is, in state s, a bijection between processes in local state a and outgoing edges from (a, i). Construct state t from s by performing a transition from a to b for every process associated with the edge (a, b) of X and the same transition among control states that is present in the abstract graph. Such a transition must be enabled by

the abstract graph definition. By this construction, $\phi_n(t)$ is the $(i+1)$st state of $\delta_{\mathcal{A}}$, and for every local state b, the number of processes in local state b in t is the number of occurrences of $(b, i+1)$ in the set of tours. Thus, the inductive assertion holds for the path $\sigma; t$.

Hence, after m steps, the final state σ_{m-1} is a permutation of the first state σ_0. Repeating the construction for k times, for some $k \leq n$, produces a path σ with last state identical to the first, such that $\gamma_n(\sigma) = \delta_{\mathcal{A}}^k$. Thus, $\gamma_n(\sigma^\omega) = \delta_{\mathcal{A}}^\omega$. Hence, δ is a good cycle. □

The previous theorem appears to imply that the entire cycle needs to be stored in order to determine its threaded graph decomposition so that the test for "goodness" can be performed. We show that this is not the case; only a bounded amount of information needs to be stored. In addition, this information about the cycle can be collected during an incremental traversal of the cycle, which is crucial to the PSPACE algorithm presented later.

For a finite path α with m states in $\mathcal{A}$ define $\overline{\alpha}$ to be the relation over $S_U \times S_U$ where $(a, b) \in \overline{\alpha}$ iff there is a path from $(a, 1)$ to (b, m) in H_α. We say that relation R is cyclic iff every edge in the graph of R lies on a cycle.

Lemma 4.4 *For a cycle δ in $\mathcal{M}$, $sccd(T_\delta)$ is empty iff $\overline{\delta}$ is cyclic.* □

(LHS ⇒ RHS) Suppose that $sccd(T_\delta)$ is empty. For any $(a, b) \in \overline{\delta}$, there is a path from $(a, 0)$ to $(b, m-1)$ in H_δ. As $sccd(T_\delta)$ is empty, this path is entirely within some component C. Since this is strongly connected, for some k, there is a path from $(b, m-1)$ to $(a, 0)$ in C of the form $(b, m-1); (b, 0); \ldots; (c_1, m-1); (c_1, 0); \ldots; (c_k, 0)$, with $c_k = a$. This implies that $(b, c_1) \in \overline{\delta}$, and $(c_i, c_{i+1}) \in \overline{\delta}$ for $i \in [1..k-1]$. Hence, there is a cycle containing the edge (a, b) in the graph of $\overline{\delta}$.

(RHS ⇒ LHS) Suppose that $sccd(T_\delta)$ is not empty. So there exist components C and D such that for some x, y, $(x, i) \in C$, $(y, j) \in D$, and $((x, i), (y, j)) \in E(G_\delta)$. For some a, there is a path from $(a, 0)$ to (x, i), and for some b, a path from (y, j) to

$(b, m-1)$, as the threaded graph has a total transition relation. Thus, $\overline{\delta}$ contains the pair (a, b). Since $\overline{\delta}$ is cyclic, (a, b) lies on a cycle in the graph of $\overline{\delta}$. This cycle must correspond to a cycle in the max-scc decomposition of T_δ, which is a contradiction. □

Theorem 4.4 *Formula* Ah *is not universal iff there is a finite path in* $\mathcal{M}$ *from an initial state to a green state and a cycle* δ *from that state such that* $\overline{\delta}$ *is cyclic.*

Proof. Follows from Theorem 4.2 and 4.3 and Lemma 4.4. □

Let L be the maximum length of a guard in C and U processes. Note that $L \leq |C| + |U|$.

Theorem 4.5 *There is a nondeterministic algorithm to decide if a temporal property over computations of* C *is not universal that uses space* $O(|S_U|^2 + log(|S_C||S_\mathcal{B}|) + L)$. *The algorithm uses space logarithmic in the size of* $\mathcal{M}$.

Proof. By Theorem 4.4, a property Ah is not universal iff there is a finite path in $\mathcal{M}$ to a green state and a following cycle δ from that state such that $\overline{\delta}$ is cyclic. The algorithm "guesses" a path to a green state, and a cycle δ from it, recording only the current state of $\mathcal{M}$, and $\overline{\rho}$ for the prefix ρ of δ that has been examined. As $\overline{(\rho; X; s)} = \overline{\rho} \circ X$, ($\circ$ is relational composition) $\overline{\delta}$ can be computed incrementally.

Recording a state of $\mathcal{M}$ takes space $(log(|S_C||S_\mathcal{B}|) + |S_U|)$. Computing a successor state can be done in space proportional to $(log|S_\mathcal{B}| + log|S_C| + log|S_U| + L)$ (as this requires checking if $(c, S) \,||- p$ for guards p). Storing $\overline{\delta}$ takes space $|S_U|^2$, and checking if $\overline{\delta}$ is cyclic can be done deterministically within the same space bound. Thus, the overall space usage is $O(|S_U|^2 + log(|S_C||S_\mathcal{B}|) + L)$. □

Remark 4.1 There are two special cases where the algorithm can be optimized. If the user processes are deterministic, every cycle δ in $\mathcal{M}$ is good (as T_δ must be empty). If the correctness property is a safety property, the algorithm need check

only finite accepting paths, which are good by Lemma 4.1. In both cases, the check for good cycles can be eliminated, which is a substantial saving. □

A reduction from a generic PSPACE Turing Machine shows that checking if $\mathsf{AG}\neg$*accept* is not universal is PSPACE-hard . Section 4.8 contains the details of this proof. The following theorems follow.

Theorem 4.6 *Deciding if a property over computations of C is not universal is complete for PSPACE .*

Corollary 4.2 *Deciding if a property over computations of C is universal is complete for PSPACE .*

Proof Sketch. Follows from the previous theorem, as co-PSPACE = PSPACE . □

The algorithm given above for determining if a property is not universal is nondeterministic and uses polynomial space. Using Savitch's construction, there is a deterministic algorithm with time complexity $O(2^{k(|S_U|^2+log(|S_C||S_\mathcal{B}|)+L)^2})$ for some k. We present below a "natural" deterministic algorithm with the same worst case time complexity in $|S_U|$. Let $K = |S_\mathcal{M}| \times 2^{|S_U|^2}$. The algorithm follows from this observation:

Proposition 4.3 *If ρ is a finite path in $\mathcal{M}$ from s to t of length greater than K, then there is a path δ from s to t in $\mathcal{M}$ of length at most K such that $\overline{\rho} = \overline{\delta}$.*

Proof. Define an equivalence relation on states s of ρ by $s_i \equiv s_j$ iff $s_i = s_j$ and $\overline{X_0 \circ X_1 \circ \ldots \circ X_{i-1}} = \overline{X_0 \circ X_1 \circ \ldots \circ X_{j-1}}$. Clearly there are at most K equivalence classes. So if the length of ρ is greater than K, there must be distinct indices i and j such that $i < j$ and $s_i \equiv s_j$. Then the path η formed by appending the suffix from s_j to the prefix up to s_i is a path in $\mathcal{M}$ that is shorter than the path ρ, and is such that $\overline{\eta} = \overline{\rho}$. Repeating this construction a finite number of times produces a path δ with the desired properties. □

Theorem 4.7 *There is a deterministic algorithm to determine if a property is not universal with exponential worst case time complexity in* $|S_U|$.

From Theorem 4.4, the algorithm has to check that there is a finite path in $\mathcal{M}$ from an initial state to a green state and a cycle δ from that state such that $\overline{\delta}$ is cyclic. From Proposition 4.3, it suffices look for cycles of length at most K. Given the transition relation of $\mathcal{M}$, the algorithm uses iterative squaring to construct a transition relation R such that $(s,t) \in R$ iff there is a path of length at most K from s to t in $\mathcal{M}$. Next, the algorithm uses depth-first search to search for a path from an initial to a green state, and a self loop on that state such that the label associated with the self loop is cyclic.

The iterative squaring requires $log(K) = O(|S_U|^3)$ steps. Each squaring step requires time $T = O(2^{4|S_U|^2})$ in $|S_U|$. Thus, the overall time complexity is $T * log(K)$, which is exponential in $|S_U|$. □

This result justifies the claim made in the introduction that cutoffs on the number of processes depend on the formula (here, the automaton) structure. Since only cycles of bounded length, and paths of bounded length leading to the cycle need be considered, the proof of Lemma 4.3 implies that the instance that contains the corresponding accepting path is of a size that is a function of this bound. So if all instances up to the cutoff do not contain a path accepted by the automaton, no larger instance can contain such a path; thus all larger instances are correct.

4.5 Symmetry reduction

Let π be a permutation over the set $\{1 \ldots n\}$. For a state s in $\mathcal{G}_n$, the permuted state $\pi(s)$ is defined by $(\pi(s))(0) = s(0)$, and for every $i \in [1..n], (\pi(s))(i) = a_i$ iff $s(\pi^{-1}(i)) = a_{\pi^{-1}(i)}$. For example, the state (c, u_1, v_2, w_3) under the permutation $\pi = \{(1 \rightarrow 2), (2 \rightarrow 3), (3 \rightarrow 1)\}$ becomes (c, w_1, u_2, v_3). As $\phi_n(\pi(s)) = \phi_n(s)$, from

Proposition 4.1, the truth value of any guard is the same in both s and $\pi(s)$. Hence there is complete symmetry among the user processes in any size instance of a (C, U) family, and the PMCP for formulas of type (2) and (3) reduces that for formulas of type (1). The following lemmas are based on those in [ES 93, CFJ 93] (cf. [ID 93]) Let $f(i)$ be a CTL^* formula with propositions over the states of C and over the states of U indexed with i, and let $f(i,j)$ be a CTL^* formula with propositions over the states of C and over the states of U indexed with either i or j.

Lemma 4.5 *For $n \geq 1$, $\mathcal{G}_n, \iota_{\mathcal{G}_n} \models \bigwedge_i f(i)$ iff $\mathcal{G}_n, \iota_{\mathcal{G}_n} \models f(1)$.*

Lemma 4.6 *For $n \geq 2$, $\mathcal{G}_n, \iota_{\mathcal{G}_n} \models \bigwedge_{i,j:i\neq j} f(i,j)$ iff $\mathcal{G}_n, \iota_{\mathcal{G}_n} \models f(1,2)$.*

Let $C \lhd_\alpha U$ be the process where a copy of the user process is "merged" into the control in the following manner. Let $S_{C\lhd_\alpha U} = S_C \times S_{U_\alpha}$, and $(c,u) \overset{p^\alpha \wedge q^\alpha}{\rightarrow} (d,v) \in R_{C\lhd_\alpha U}$ iff $c \overset{p}{\rightarrow} d \in R_C$ and $u \overset{q}{\rightarrow} v \in R_{U_\alpha}$, where for any guard p, p^α is p with every global predicate $(\exists i :: \mathcal{E}(i))$ replaced with $\mathcal{E}(\alpha) \vee (\exists i :: \mathcal{E}(i))$.

Theorem 4.8 *A property of the form $\bigwedge_i \mathsf{A}h(i)$ is universal for a (C,U) family iff $\mathsf{A}h(\alpha)$ is universal for the control process in the family $(C \lhd_\alpha U, U)$.*

Theorem 4.9 *A property of the form $\bigwedge_{i,j:i\neq j} \mathsf{A}h(i,j)$ is universal for a (C,U) family iff $\mathsf{A}h(\alpha,\beta)$ is universal for the control process in the family $(((C \lhd_\alpha U) \lhd_\beta U), U)$.*

4.6 Applications

We have implemented this algorithm to verify a bus arbitration protocol based on the SAE J1850 draft standard [SAE 92] for automobile applications. This is a protocol where many micro-controllers can transmit symbols along a shared single-wire bus. As a consequence of this restriction, symbols are encoded by the width of a pulse. Nodes on the bus may begin transmitting different messages simultaneously; only

the node with the highest priority message should complete transmission after the arbitration process. Symbol 0 has priority over symbol 1, and priority between messages over the alphabet $\{0, 1\}$ is determined lexicographically. The micro-controllers are modeled as user processes, and the bus as the control process.

We implemented the PMCP algorithm presented here by generating SMV [McMillan 92] code to describe the abstract process transitions, given a description of the next-state relation of a user and control processes. The correctness property has both a safety and a progress component. For the safety property, we are able to simplify the implementation as described following Theorem 4.5. The progress property holds on the abstract graph, and as the abstract graph simulates each individual instances, it follows that the property holds of all instances.

The SAE-J1850 protocol is quite complex, as it has to take into account nondeterministic, but bounded, delays that may occur at a micro-controller when it is sensing a bus state transition. The protocol has the maximum delay as a parameter, in addition to the number of processes. We were able to show, by an abstraction mapping, that, for n processes, the protocol with maximum delay 2 is trace equivalent to that with a higher delay. This allows us to focus on the number of processes as the single parameter, with the maximum delay being fixed at 2. For this system, each user process has about 180 states, while the control process together with the automaton for the property has about 400 states. This implies that the abstract graph is represented with about 200 boolean variables describing an abstract state, as each abstract state includes a subset of user states. Despite this large number of variables, verification of the correctness properties over the abstract graph took less than a minute on an Intel Pentium running at 200 MHz with 32 MB of memory. We emphasize that this establishes correctness of the bus protocol for an arbitrary number of attached micro-controllers. The following chapter contains a detailed description of this protocol and its verification.

4.7 Almost universal properties

In Section 4, we considered the question of whether a property is universal, i.e., true for every size instance of a (C, U) family. However, it is possible that a property is not true of all instances, but of almost every instance (i.e., all but a finite number of instances). We say that such a property is *almost everywhere true*. Similarly, a property that is false of almost every instance is called *almost everywhere false*[2].

We showed in Section 4 that a property of the form Ah is not universal iff there is an accepting good path in $\mathcal{M}$. To generalize this result, we define another property of paths in $\mathcal{A}$.

Definition 4.7 (Real Path) *A path ρ in $\mathcal{A}$ is real iff* $(\overset{\infty}{\forall} n\ (\exists \sigma : \sigma \in \mathcal{G}_n : \gamma_n(\sigma) = \rho))$. $(\overset{\infty}{\forall}$ *is read as "for all but a finite number")* □

Lemma 4.7 *Every path (finite or infinite) in $\mathcal{A}$ is* real *iff it is* good.

Proof. The direction from left to right follows from the definitions. So suppose ρ is a good path in $\mathcal{A}$. Hence, for some n, there is a path σ in $\mathcal{G}_n$ such that $\gamma_n(\sigma) = \rho$. By the Simulation Lemma (Lemma 4.1), for every $n' \geq n$, there is a path $\delta \in \mathcal{G}_{n'}$ that covers σ. By definition of a covering path, $\gamma_{n'}(\delta) = \rho$. So in every instance of size at least n, there is a path which maps to ρ. Hence, ρ is real. □

Let a path in $\mathcal{M}$ be real iff the projection of the path on $\mathcal{A}$ is real. We get this analogue to Theorem 4.2:

Theorem 4.10 *Formula* Ah *is almost everywhere false iff there is an accepting real path in $\mathcal{M}$.*

Proof. Suppose δ is an accepting real path in $\mathcal{M}$. As $\delta_{\mathcal{A}}$ is real, for almost every n, there is a path in $\mathcal{G}_n$ that matches $\delta_{\mathcal{A}}$ on the sequence of states of C, and is hence accepted by $\mathcal{B}$. Therefore, Ah is false of almost every instance.

[2] A similar notion, *almost always satisfiability*, is considered by [ESr 90] when synthesizing a many process program that satisfies a temporal specification.

In the other direction, if Ah is almost everywhere false, then for some n, there is a path σ in $\mathcal{G}_n$ from the initial state that is accepted by $\mathcal{B}$. From Lemma 4.2, $\gamma_n(\sigma)$ is a path in $\mathcal{A}$, which is good by construction, and hence real by Lemma 4.7. The sequence of states of C in $\gamma_n(\sigma)$ is the same as in σ; hence, there is a run of $\mathcal{B}$ on $\gamma_n(\sigma)$ that forms an accepting real path in $\mathcal{M}$. □

Corollary 4.3 *Formula* Ah *is almost everywhere false iff* Ah *is not universal.*

Proof. By Theorem 4.2, Ah is not universal iff there is an accepting good path in $\mathcal{M}$. As good paths are real by Lemma 4.7, the claim follows from Theorem 4.10. □

This is a strong result, as it implies that if Ah is false of *some* instance, then it is false in *almost every* instance. The following theorem shows the interesting fact that almost everywhere truth and falsity are complementary, which is not true in general.

Theorem 4.11 *Formula* Ah *is almost everywhere false iff* Ah *is not almost everywhere true.*

Proof. Suppose that Ah is not almost everywhere true. Then for some n, there is a path σ in $\mathcal{G}_n$ from an initial state that is accepted by $\mathcal{B}$. From Proposition 4.2, $\delta = \gamma_n(\sigma)$ is a path in $\mathcal{A}$ that has the same sequence of states of C as does σ, and hence is accepted by $\mathcal{B}$. By construction, δ is good, and hence it is real by Lemma 4.7. The accepting run of $\mathcal{B}$ on δ defines an accepting real path in $\mathcal{M}$. Hence, by Theorem 4.10, Ah is almost everywhere false. The other direction holds trivially from the definitions. □

4.8 Hardness Results

4.8.1 PSPACE completeness

Theorem 4.12 *Deciding universality is complete for PSPACE .*

Proof. A PSPACE algorithm for the PMCP is presented in Theorem 4.5. To show PSPACE hardness, suppose that M is a deterministic machine operating in space polynomial in the size of its input. Without loss of generality, assume that there is a k, such that on every input x, M uses exactly $|x|^k$ space.

Given M and input x, we construct processes C and U such that the (C, U) family simulates the computation of M on x. First, construct machine N such that N ignores its input, prints x on the work tape, and simulates M on x. By using a binary valued mark at each tape cell that is toggled on writing to the cell, N is guaranteed not to over-write a tape cell with the same symbol on a transition.

The control process simulates N, while the user processes simulate individual work tape cells. The process C has a counter *head* with range $[0..|x|^k)$ that maintains the current head position of N. The initial state of C is the initial state of N, with $head = 0$. Each user process chooses a state representing a position on the tape. The symbol stored in each user process is initially blank.

In the initial state, C checks, using global predicates, that there is at least one user process at each tape position in the range $[0..|x|^k)$. If this is true, C then proceeds to simulate N. A transition $\delta(p, a) = (q, b, D)$ of N is simulated by C with the following (we write the transitions in a guarded command language for clarity):

$$
\begin{array}{ll}
 & \neg change \wedge state = p \wedge (\exists i : position_i = head : symbol_i = a) \longrightarrow \\
 & \quad change := true; tosymbol := b \\
[] & change \wedge state = p \wedge \neg(\exists i : position_i = head : symbol_i = a) \longrightarrow \\
 & \quad change := false; state := q; head := head + D
\end{array}
$$

while U has the following transition

$$
change_C \wedge tapeposition = head_C \wedge symbol = a \wedge tosymbol_C = b \longrightarrow symbol := b
$$

Informally, for every move $\delta(p, a) = (q, b, D)$ of N, C first sets its *tosymbol* to b, then every process with tape position *head* sets the new value of its stored symbol to b, after which C changes its *head* value. For this last step to execute after the

second one, it is important that $a \neq b$. This is ensured by the marking discipline given above.

In any instance of the family, every computation where there is initially at least one user process at each tape position simulates the computation of N; otherwise, C enters a deadlock state, and the computation deadlocks. Let the property to be verified be $\mathsf{AG}\ \neg accept$. If $x \in L(M)$, then N enters an accepting state. For any instance of size at least $|x|^k$, there is an initial state where each tape cell is assigned to some user process; hence, there is a computation that accepts, so the property is false. If $x \notin L(M)$, then N does not enter an accepting state, so every computation of each instance does not contain an accepting state; hence, the property is true of all instances. Thus, $x \in L(M)$ iff $AG\ \neg accept$ is not universal. All the constructions can be performed in LOGSPACE. Thus, deciding if a temporal property over computations of C is not universal is PSPACE-complete . Note that the property used in the proof is a simple invariance property, and the control process is deterministic. □

4.8.2 Undecidability for interleaving semantics

We show that assuming the process structure for C and U described earlier, but with an *interleaving* parallel composition operator, the simplest form of the PMCP, i.e with type (1) formulas, is undecidable. Essentially, this is so because the $\forall$ quantification can be used to simulate the zero-testing actions of a two-counter machine.

The main idea (cf. [GS 92]) is that C and U are constructed such that $C \parallel U^n$ simulates the given deterministic two counter machine (henceforth, 2CM) for at least n steps. The control process executes the program of the 2CM, while the user processes simulate the two counters. The counters are represented in unary, and each copy of the user process has sufficient storage for one bit of each counter,

and some additional boolean flags. Initially, all the flags are *false*, and all bits are set to 0.

For clarity, we have expressed the simulation in a guarded command format. The control process simulates the instructions of the 2CM with an explicit program counter (PC), used to implement sequential execution in the guarded command model. An outline of the 2CM simulation follows.

1. Zero Testing : The $zero?(x)$ test (x names a counter) is simulated by

$$\begin{array}{ll} & PC = current \wedge \neg(\exists i :: x_i = 1) \longrightarrow zero(x) := true; inc(PC) \\ [] & PC = current \wedge (\exists i :: x_i = 1) \longrightarrow zero(x) := false; inc(PC) \end{array}$$

2. Increment Counter x : This is done by a four way handshake protocol between C and one of the U processes that has $x = 0$. The code for C is :

$$\begin{array}{ll} & PC = current \wedge \neg reset \wedge \neg incr(x) \wedge (\exists i :: x_i = 0) \longrightarrow \\ & \quad incr(x) := true \\ [] & PC = current \wedge incr(x) \wedge (\exists i :: DoneIncr(x)_i) \longrightarrow \\ & \quad incr(x) := false; reset := true \\ [] & PC = current \wedge reset \wedge \neg(\exists i :: DoneIncr(x)_i) \longrightarrow \\ & \quad reset := false; inc(PC) \end{array}$$

   The code for a U process is as follows :

$$\begin{array}{ll} & x = 0 \wedge incr(x)_C \wedge \neg(\exists i :: DoneIncr(x)_i) \longrightarrow x := 1; DoneIncr(x) := true \\ [] & DoneIncr(x) \wedge reset_C \longrightarrow DoneIncr(x) := false \end{array}$$

   Notice that if an increment is not possible because all process bits are set to 1, the control process will deadlock.

3. Decrement Counter x : The simulation is similar to that for increment.

Note that both C and U are deterministic. The non-halting property for C can be expressed as $\mathsf{AG}\neg Halt$.

Theorem 4.13 *The PMCP is undecidable even in the simplest case in an interleaving computation model. More precisely, it is co-RE.*

Proof. The simulation of a non-halting 2CM will, for any instance, either deadlock or execute the program forever, never reaching the *Halt* state. Thus, along every computation in every instance of size n, $\mathsf{G}\neg halt$ holds, hence $\mathsf{AG}\neg Halt$ is universal. Conversely, a halting 2CM halts in k steps, for some value of k, and the simulation will enter a halt state, so $\mathsf{AG}\neg Halt$ is not universal. It follows that the 2CM does not halt iff $\mathsf{AG}\neg Halt$ is universal, so the PMCP is co-RE. □

4.9 Conclusions and Related Work

A variety of positive results on the PMCP have been obtained previously. All of them, however, possess certain limitations, which is perhaps not surprising since the PMCP is undecidable in general (cf. [AK 86],[Suzuki 88]). Many of these methods are only partially automated, requiring human ingenuity to construct, e.g., a process invariant or a closure process (cf. [CG 87, BCG 89, KM 89, WL 89]). Some could be fully automated but do not appear to have a clearly defined class of protocols on which they are guaranteed to succeed (cf. [SG 89], [Vernier 93], [CGJ 95]).

Abstract graphs (for asynchronous systems) are considered in [ESr 90] for synthesis, [Vernier 93] for automatic but incomplete verification, and in [CG 87], where they are called process closures. Interestingly, [CG 87] show (in our notation) that if, for some k, $C \parallel U^k \parallel \mathcal{A}$ is appropriately bisimilar to $C \parallel U^{k+1} \parallel \mathcal{A}$, then it suffices to model-check instances of size at most k to solve the PMCP. However, they do not show that such a cutoff k always exists, and their method is not guaranteed to be complete. Pong and Dubois [PD 95] propose a similar abstract graph construction for verification of safety properties of cache coherence protocols. They consider a synchronous model with broadcast actions. Although sound for verification, their method appears to be incomplete. Lubachevsky [Lubachevsky 84]

makes an interesting early report of the use of an abstract graph similar to a "region graph" for parameterized asynchronous programs using *Fetch-and-Add* primitives; however, while it caters for (partial) automation, the completeness of the method is not established and it is not clear that it can be made fully automatic.

Our approach, in contrast, is a fully automated, sound and complete one (i.e., always generates a correct "yes" or "no" answer to the PMCP). Another such approach appears in [GS 92]. They also consider systems with a single control process and an arbitrary number of user processes, but with asynchronous CCS-type interactions. Unfortunately, their algorithm has exponential space (double exponential time) worst case complexity.

Our framework thus differs from [GS 92] in these significant respects: (a) the parallel composition operator is synchronous; (b) we permit guards with "everywhere" quantification (i.e., of the form $(\forall i :: \mathcal{E}(i))$); (c) it is more tractable (PSPACE vs. EXPSPACE)[3]. Partial synchrony can also be handled in our framework. These factors permit us to represent a wider range of concurrent systems. For example, the bus protocol described in Section 4.6 relies on the ability to test everywhere predicates, which are not permitted in [GS 92]. There is a noteworthy limitation in the modeling power of our present framework. Because of the covering property (Lemma 4.1), an algorithm for mutual exclusion cannot be implemented in our model (cf. [GS 92]'s control process-free model), even with the control process.

4.10 Technical Details

Proposition 4.2 *For every path σ in $\mathcal{G}_n$, $\gamma_n(\sigma)$ is a path in $\mathcal{A}$.*

Proof. The proof is by induction on the number of states in σ. If σ has a single

[3]On the other hand, for their model of computation with all user processes but no control process, there is a polynomial time algorithm [GS 92]. We believe that our PSPACE completeness result is not an insurmountable barrier to practical utility, given BDD-based implementations, as suggested in section 4.6.

state s, $\gamma_n(\sigma) = \phi_n(s)$, which is a path in $\mathcal{A}$.

Assume inductively that the claim is true for all paths of length at most m, with $m \geq 1$. Consider a path σ of length $m+1$. Then σ maybe written as ρt. Let s be the last state in ρ. Let $X = \psi_n(s,t)$, $\phi_n(s) = (c,S)$, and $\phi_n(t) = (d,T)$. As there is a transition from s to t in $\mathcal{G}_n$, there is an enabled transition with guard p for the control process. Furthermore, from Proposition 4.1 $(c,S)\,||-\,p$. From the definition of ψ_n, X is total on S, and X^{-1} is total on T. For any (a,b) in X, $a_i = s(i)$ and $b_i = t(i)$ for some i, so there is an enabled transition with guard q from a_i to b_i in U_i, and $(c,S)\,||-\,q$.

Thus, $(\phi_n(s), \psi_n(s,t), \phi_n(t)) \in R_{\mathcal{A}}$. By the inductive hypothesis, $\gamma_n(\rho)$ is a path in $\mathcal{A}$ ending at the state $\phi_n(s)$, so $\gamma_n(\sigma)$ is a path in $\mathcal{A}$. Thus, the claim is true of all finite paths in $\mathcal{G}_n$, hence it is true for all infinite paths, as an infinite computation of the system is the unique limit of its finite prefixes. □

Lemma 4.5 *For $n \geq 1$, $\mathcal{G}_n, \iota_{\mathcal{G}_n} \models \bigwedge_i f(i)$ iff $\mathcal{G}_n, \iota_{\mathcal{G}_n} \models f(1)$.*

Proof. The left-to-right direction follows directly from the definition of $\bigwedge_i$. For the other direction, if $\mathcal{G}_n, \iota_{\mathcal{G}_n} \models f(1)$, then $\pi(\mathcal{G}_n), \pi(\iota_{\mathcal{G}_n}) \models f(\pi(1))$ for any permutation π over $\{1..n\}$. As the system exhibits complete symmetry among the user processes, $\pi(\mathcal{G}_n) = \mathcal{G}_n$, and $\pi(\iota_{\mathcal{G}_n}) = \iota_{\mathcal{G}_n}$. Hence $\mathcal{G}_n, \iota_{\mathcal{G}_n} \models f(\pi(1))$ for any permutation π. Choosing π's appropriately, we have that $\mathcal{G}_n, \iota_{\mathcal{G}_n} \models f(i)$ for all $i \in [1..n]$. Hence, $\mathcal{G}_n, \iota_{\mathcal{G}_n} \models \bigwedge_i f(i)$ holds. □

For a computation η of $(C \lhd_\alpha U) \parallel U^n$, let $\hat{\eta}$ be the corresponding computation of $C \parallel U^{n+1}$), formed by considering the copy of U in $(C \lhd_\alpha U)$ as a separate process.

Proposition 4.4 *The families $(C \lhd_\alpha U, U)$ and (C, U) are related in the following way:*

(a) *For every computation η in $(C \lhd_\alpha U) \parallel U^n$, $\hat{\eta}$ is a computation in $C \parallel U^{n+1}$.*

(b) *For $n \geq 1$, for every computation σ in $C \parallel U^{n+1}$, there is a computation η in $(C \lhd_\alpha U) \parallel U^n$ such that $\hat{\eta} = \sigma$.*

Theorem 4.8 *A property of the form $\bigwedge_i \mathsf{A}h(i)$ is universal for a (C, U) family iff $\mathsf{A}h(0)$ is universal for the control process in the family $(C \lhd_\alpha U, U)$.*

Proof. Suppose $\bigwedge_i \mathsf{A}h(i)$ is universal for the (C, U) family. From Lemma 4.5, for $n \geq 1$, $\mathcal{G}_n \models \bigwedge_i \mathsf{A}h(i)$ iff $\mathcal{G}_n \models \mathsf{A}h(1)$, thus, $\mathsf{A}h(1)$ is universal. By Proposition 4.4(a), $\mathsf{A}h(\alpha)$ is universal for the control process in the family $(C \lhd_\alpha U, U)$. The argument for the other direction is analogous, and uses proposition 4.4(b). □

Chapter 5

Verification of a Bus Arbitration Protocol

5.1 Introduction

In the previous chapter, we presented an algorithm for the verification of parameterized synchronous systems. While the problem is decidable, it was shown to be PSPACE-complete in the size of the system description. This complexity bound may make it appear that the algorithm is unlikely to be useful in practice. We present a case study on the verification of an parameterized industrial standard protocol. The protocol is called the SAE-J1850 protocol [SAE 92], and is an automobile industry standard for transmitting data between various sensors and controllers in an automobile. The system consists of a single-wire bus, to which several controllers (units) are attached. Since the bus is a single wire, symbols **0** and **1** are transmitted by encoding them by both the length and the value of a bus pulse. For instance, a **0** may be sent with either a long high or a short low pulse.

Several units may transmit concurrently; the protocol incorporates a *distributed, on-the-fly* arbitration mechanism which ensures that only the units trans-

mitting the highest priority message succeed. Priority between messages (strings over $\{\mathbf{0}, \mathbf{1}\}$) is determined by lexicographic order, given that the symbol $\mathbf{0}$ has priority over the symbol $\mathbf{1}$. The protocol is correct if it ensures that the arbitration mechanism functions correctly. We should note here that the protocol as described in [SAE 92] has other higher-level functionality, which we have not considered, in order to concentrate our attention on the core arbitration question. The protocol is further complicated by the presence of arbitrary, but bounded delays in the units while detecting a change in the bus state. These delays have an electrical origin; they arise from delays in the detection circuitry, and the presence of different bias voltages at the units. To accommodate these delays, "long" and "short" are actually time *intervals*, whose length is proportional to the maximum delay. Thus the protocol is parameterized both by the maximum delay, and by the number of units taking part in it.

The verification of the protocol proceeds by two applications of abstraction, one for each parameter. The first abstraction theorem shows a *delay independence* property of the protocol : an instance of the protocol with n processes and maximum delay Δ is correct iff the instance with n units and maximum delay 2 is correct. Thus, correctness need be proved only for the family of instances with maximum delay 2. The second abstraction uses the algorithm in [EN 96] to handle the parameterization over the number of units in a *fully* automated manner; the algorithm constructs a *finite* "abstract graph", which represents the entire family of instances exactly, over which properties can be model-checked. A simple version of this protocol, without the complexity introduced by the delays, was verified in [EN 96]. The modeling of the delay not only introduces complexity into the behavior of the units, but also introduces additional parameterization into the protocol, which is dealt with by the delay independence theorem.

The success of this effort leads us to believe that careful specification of the

computational model underlying other protocols will expose constraints that can be utilized, as in this case, for developing decision procedures for large classes of protocols. It also exposes a dire need for developing and popularizing notation for expressing such protocols. Remarkably, the SAE-J1850 document does not contain a succinct protocol description; the development of such a description was a major component of this project. The successful verification of the protocol using symbolic methods, despite the theoretical result on PSPACE-completeness of the procedure used [EN 96], is reason to believe that fully automated parameterized verification is feasible for reasonably sized protocols.

The rest of the chapter is structured as follows: Section 5.2 describes the various components of the protocol in more detail. Section 5.3 discusses the abstractions used for handling the parameterizations. In Section 5.4, we describe the implementation of the [EN 96] algorithm, and its application to this protocol. Section 5.5 concludes the chapter and provides comparisons with related work.

5.2 Protocol Description

The SAE-J1850 protocol is a data transfer protocol over a single wire bus, which is intended to be used for communication between various sensors and controllers in an automobile. The restriction to a single wire bus reduces wiring complexity. An instance of the parameterized system consists of several *units* connected to a single *bus*. The operation of the protocol can be described at the "interface" and "implementation" levels.

At the interface level, the units communicate by broadcasting messages (sequences of symbols from the set $\{\mathbf{0}, \mathbf{1}\}$) over the bus. Units may transmit concurrently; arbitration takes place during transmission. The arbitration mechanism is defined in terms of priority among symbols; the symbol $\mathbf{0}$ has higher priority than $\mathbf{1}$. The priority order among symbols is extended to messages by lexicographic order-

Length	*Trmin*	*Txmin*	*Txmax*	*Trmax*
Short	2.5Δ	3.5Δ	4.5Δ	5.5Δ
Long	6.5Δ	7.5Δ	8.5Δ	9.5Δ

Figure 5.1: Interval Lengths

ing. The key correctness property of this protocol is that the arbitration mechanism works as follows : whenever several units are sending messages concurrently, the message with the highest priority is placed on the bus.

At the implementation level, since the bus is a single wire, symbols are encoded by pulses of differing length and the bus value during the pulse. For instance, the **0** symbol is encoded by either a "long" high pulse, or by a "short" low pulse. The high and low states on the bus are referred to as *Dominant* and *Passive* respectively in the SAE-J1850 document [SAE 92], so we will use this terminology in the rest of the chapter. The state of the bus is an "or" of the bus states desired by the units. The protocol is further complicated by non-deterministic, but bounded delays in the units while detecting a change in bus value. This delay is caused either by bias voltages, or by delays in the detection circuitry. To account for these delays, "long" and "short" are not fixed numbers, but are instead non-empty intervals, whose length is proportional to the maximum delay parameter, which we term Δ.

We will continue to use the symbolic names "long" and "short". There are four parameters associated with a symbolic length l : $Txmin(l)$, $Trmin(l)$, $Txmax(l)$, $Trmax(l)$. Their values are based on a nominal value $Tnom(l)$ and are given by the formulae : $Txmin(l) = Tnom(l) - \Delta/2$, $Txmax(l) = Tnom(l) + \Delta/2$, $Trmin(l) = Tnom(l) - 3\Delta/2$, $Trmax(l) = Tnom(l) + 3\Delta/2$. $Tnom(l)$ is itself proportional to Δ. $Tnom(Long) = 8 * \Delta$, and $Tnom(Short) = 4 * \Delta$. The values are given explicitly in the table below:

Note that the interval $[Txmin(l), Txmax(l)]$ is properly contained in the in-

terval $[Trmin(l), Trmax(l)]$, and that the least *Long* value exceeds the largest *Short* value by Δ. The core of the protocol is the following procedure followed by each unit to transmit a symbol with symbolic length l at a bus value of b (e.g., $\mathbf{0}$ as a *Short*, *Passive* pulse). At the entry to this procedure, $request = b, localbus = b$, and $counter = 1$.

Informally, the procedure above attempts to maintain the bus at value b for $Txmin(l)$ time units. If this attempt succeeds, then it attempts to change the bus value to $\neg b$ within $Txmax(l)$ time units, so as to terminate the pulse. If that fails, then the procedure switches to a *Passive* request, and waits for some other unit to change the bus value. As the names indicate, $[Trmin(l), Trmax(l)]$ is the interval for successful "reception" of the symbol, while $Txmin(l)$ and $Txmax(l)$ are the time bounds for attempting "transmission" of the symbol. $\mathbf{0}$ is encoded as either a *Short Passive* pulse or as a *Long Dominant* pulse, while $\mathbf{1}$ is encoded by the other two combinations. The asymmetry between *Passive* and *Dominant* is used to enforce the priority order $\mathbf{1} \preceq \mathbf{0}$.

5.2.1 Correctness Properties

The correctness property is stated informally in the protocol document [SAE 92] as: *Whenever several units are transmitting messages concurrently, the message with the highest priority is the one placed on the bus.*

This property can be stated precisely in CTL as follows: Consider n units connected to the bus, indexed by i, $(i \in [1, n])$. Let $\mathcal{M}(k)$ denote the set of message strings (over $\{\mathbf{0}, \mathbf{1}\}$) of length k. For each i in $[1, n]$, let msg_i denote the fixed message string that is associated with unit i. Let B denote the message that is transmitted on the bus (this may be defined as an auxiliary variable that records symbols as they are transmitted on the bus). Let tr_i be a boolean auxiliary variable that records if unit i is transmitting. Let max be the function that determines the

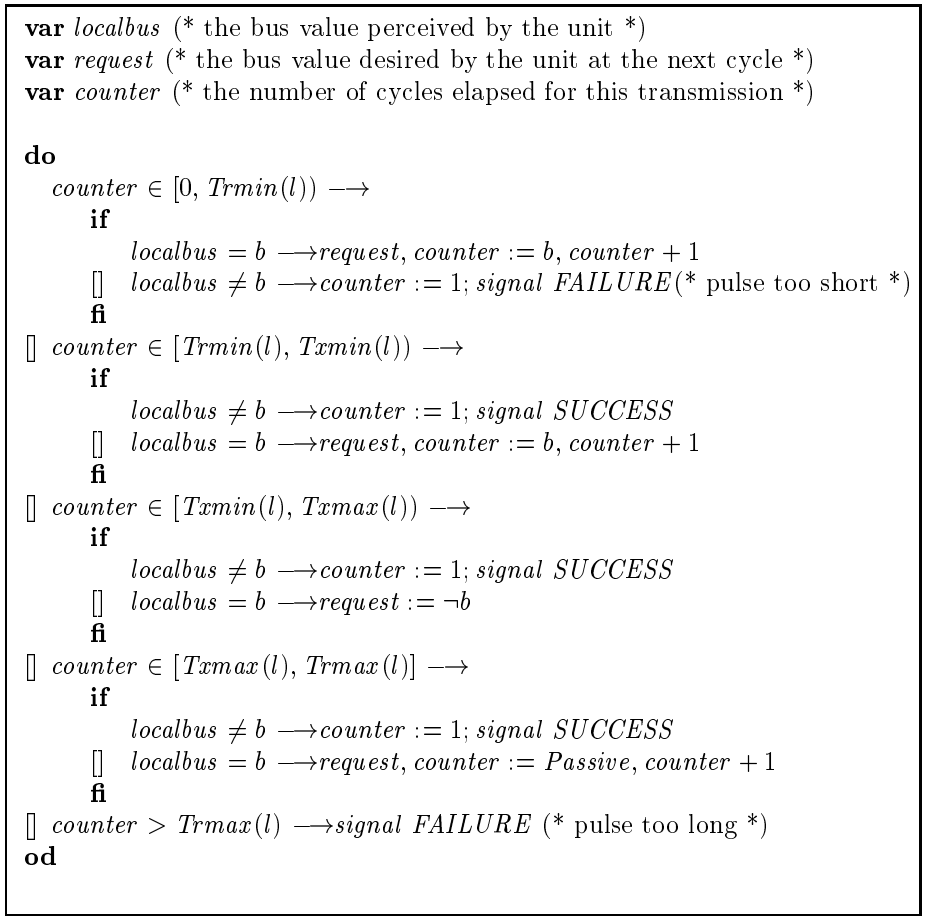


Figure 5.2: Algorithm to transmit a symbol with length l and bus value b.

maximum message of a set of messages, according to the lexicographic priority $\preceq$ on messages. If the set is empty, *max* has value ϵ, the empty string. The following CTL formula expresses the property above:

(C0) $(\forall m : m \in \mathcal{M}(k) : \mathsf{AG}(maxT = m \wedge B = \epsilon \Rightarrow \mathsf{A}(B \preceq m \mathsf{U} B = m)))$,

where $maxT = max\{i : i \in [1,n] \wedge tr_i : msg_i\}$

This expression is of finite length for fixed k. Verification of this property for a fixed k requires adding state to each unit to store message contents, which makes the state space intractably large. To solve this problem, we modify the environment of the protocol so that the message sent by a unit is generated on the fly. At any state, let $sent_i$ denote the message sent by a unit. The modified correctness property is as follows :

(C1) $\mathsf{AG}(maxS = \epsilon \wedge B = \epsilon \Rightarrow \mathsf{AG}(B = maxS))$, where $maxS = max\{i : i \in [1,n] \wedge tr_i : sent_i\}$

Informally, this property states that starting at any state where both the message on the bus and that at the units is empty, at any point of time the message on the bus is equal to the lexicographic maximum of the messages sent by the currently transmitting units. This implies that B must increase (lexicographically) as long as there is a transmitting unit.

While the new environment is simpler, the statement of the property still involves several unbounded auxiliary variables. Instead of checking this property, which refers to the history of a computation, we check several properties that deal with the transmission of a single symbol. We show in Lemma 5.1 that their conjunction implies (C1). The statement of these properties requires some auxiliary propositions : *insym* holds at states where $B = \epsilon$ or the state is at least Δ time units from the last bus state change; *E0sender* holds iff there is a transmitting unit with current symbol **0**; *E1sender* holds iff there is a transmitting unit with current symbol **1**.

Let $before(x) \equiv \mathsf{A}(insym \mathsf{U}(insym \wedge x))$, $at(x) \equiv \mathsf{A}(insym \mathsf{U}(\neg insym \wedge x))$, and $after(x) \equiv \mathsf{A}(insym \mathsf{U}(\neg insym \mathsf{U}(insym \wedge x)))$. Informally, $before(x)$ holds iff x is true at some point before the next bus change, $at(x)$ holds iff x is true at the following bus change, and $after(x)$ holds iff x is true just after the bus change is complete.

A *stable state* on an execution sequence is one where $B = \epsilon$, or the state is at Δ time units after the last bus value change. By the protocol definition, in this state every unit perceives the new bus value. A stable state is the first state for which *insym* is true after a bus change.

(C2a) In any global state where symbol transmission is in progress, and there is a unit sending **0**, the next bus value is **0**. In CTL, this is specified as

$\mathsf{AG}(insym \wedge E0sender \Rightarrow at(value = \mathbf{0}))$

(C2b) In any global state where symbol transmission is in progress, if there is a unit sending **1**, and no unit sending **0**, the next bus value is **1**.

$\mathsf{AG}(insym \wedge \neg E0sender \wedge E1sender \Rightarrow at(value = \mathbf{1}))$

The properties above are global properties. The following are properties of every unit, expressed in an indexed temporal logic (cf. [RS 85],[BCG 89]):

(C2c) In any global state where symbol transmission is in progress, every unit transmitting **0** succeeds, and continues to transmit until the next *insym* state.

$\bigwedge_i \mathsf{AG}(insym \wedge tr_i \wedge (sym_i = \mathbf{0}) \Rightarrow after(tr_i))$

(C2d) In any global state where symbol transmission is in progress, and there is a unit sending **0**, every unit transmitting **1** fails before the bus symbol is determined.

$\bigwedge_i \mathsf{AG}(insym \wedge E0sender \wedge tr_i \wedge (sym_i = \mathbf{1}) \Rightarrow before(\neg tr_i))$

(C2e) In any global state where symbol transmission is in progress and there is no unit sending **0**, every unit transmitting **1** succeeds, and continues to transmit until the next *insym* state.

$$\bigwedge_i \mathsf{AG}(insym \wedge \neg E0sender \wedge tr_i \wedge (sym_i = \mathbf{1}) \Rightarrow after(tr_i))$$

Lemma 5.1 *Properties (C2a)-(C2e) imply Property (C1).*

Proof.

We show by induction on the number of stable states on any computation from a state with $B = \epsilon$ and $maxS = \epsilon$ (the 0th stable state) that the following property holds:

(IH) At the the kth stable state, B is the maximum of the messages sent by units that were transmitting at the start of previous stable bus state if $k > 0$, otherwise it is ϵ. Every transmitting unit has sent B.

Basis : $k = 0$. The message on the bus as well as the message at every transmitting unit are both ϵ, so the claim holds.

Inductive step : Assume that (IH) holds at the kth stable state. If some unit transmits **0** at this state, by (C2a) the next symbol on the bus is **0**. By (C2c), any unit transmitting **0** is transmitting at the next stable state. By (C2d), all units transmitting **1** fail before the next stable state.

If some unit transmits **1**at this state and no unit transmits **0**, then by (C2b), the next bus symbol is **1**, and by (C2e) every unit transmitting **1** is still transmitting at the next stable state. By (IH), at the kth stable state, all units transmit the lexicographic maximum among the sent messages, hence, at the next stable state, the value of B is still the maximum among the messages sent. In either case, the inductive hypothesis holds. □

5.3 Abstractions

The procotol as described is parameterized by both the maximum delay parameter Δ, and the number of units N. Let $P(N, \Delta)$ stand for the instance of the protocol with N units and delay Δ. This parameterization makes the protocol infinite-state,

hence Model Checking cannot be applied directly to determine its correctness. We apply two abstractions that reduce the protocol to an *equivalent* finite-state system. The first abstraction demonstrates a *delay insensitivity* property of the protocol : for every N, $P(N, \Delta)$ is correct iff $P(N, 2)$ is correct. Hence, protocol correctness need be checked only for the set of instances with maximum delay 2. However, this is still a parameterized, infinite-state protocol. This parameterization can be handled with the algorithm presented in [EN 96]. This algorithm abstracts away the number of units, constructing a *finite* "abstract graph", which encodes all instances of the system. Model Checking the abstract graph created by this unit is thus equivalent to checking the doubly parameterized SAE-J1850 protocol. Experimental details are presented in the following section.

5.3.1 Delay Insensitivity

As noted in the protocol description, the timing parameters are proportional to the parameter Δ. In an underlying dense time model, each test of a clock variable x is of the form $x \in \langle l * \Delta, r * \Delta \rangle$ (the angled brackets indicate either a open or a closed end to the interval), and each reset of x is of the form $x := \mathbf{choose}\langle l * \Delta, r * \Delta \rangle$, which assigns to x a nondeterministically chosen value from the interval. It is then straightforward to show that if the intervals $\langle l*\Delta, r*\Delta \rangle$ are changed to $\langle l, r \rangle$ (dividing through by Δ), the resulting un-parameterized system has the same computations w.r.t. the non-clock variables as the original one. This is so since global states with identical non-clock values and clocks related by scaling with Δ are bisimilar. This class of systems thus forms a decidable instance of parameterized real-time reasoning (cf. [AHV 93]).

Since our model of the bus system is over integer time (each transition takes 1 time unit), we cannot use this result. The protocol, however, satisfies additional properties that make a similar reduction possible. We show that any execution of

$P(n,d)$ (d even and at least 2) can be simulated by an execution of $P(n,2)$, in the sense that the sequence of symbols on the bus is the same.

Lemma 5.2 *Let σ be an execution of $P(n,d)$ (d even and at least 2). Let l be the symbolic length of the time interval between successive stable bus states in σ. Then*

1. *Every unit sending a symbol with a different length is aborted by the start of the next stable state, and*

2. *Every unit sending a symbol with the same length is transmitting at the start of the next stable state.* □

Theorem 5.1 *Let σ be an execution of $P(n,d)$ (d even and at least 2) from a stable state. There is an execution γ of $P(n,2)$ such that the sequence of symbols on the bus is identical in σ and γ.*

Proof.

We construct γ inductively. For each i, γ_i ends in the ith stable state, the symbols on the bus in γ_i and in the subsequence of σ up to and including the ith stable state are identical, and the local states of corresponding units in the ith stable states are the same except for, possibly, the *counter* values. The *counter* values, must however, satisfy the relationship : for any pair of units p, q, $counter_p \leq counter_q$ in the ith stable state in σ implies that $counter_p \leq counter_q$ in the ith stable state in γ.

Let γ_0 equal σ_0. Let p be the unit that determines the bus change that results in the $(i+1)$st stable state. For a *Passive* to *Dominant* change, p is the first unit to request a *Dominant* bus state, and for a *Dominant* to *Passive* change, p is the last unit to request a *Passive* bus state. At each stable state, all units begin transmission of their symbol with request identical to the current bus value. Thus, the change by unit p can occur only at $counter_p = Txmin(l)$, where l is the length that p sends its symbol at. $Txmin(l) = (a/2) * \Delta$, for some a.

The order of *counter* values is the same in the ith stable state in γ. As the *counter* value in each unit does not decrease until a bus change or a termination of transmission, in every execution starting at the ith stable state in γ, unit p still is one of the units that determine the bus change. As the change of bus state occurs at the same multiple of Δ, the symbolic length, and hence the symbol on the bus is the same. From the previous Lemma, the units un-aborted at the $(i+1)$st stable states in γ and σ are the same. There exists a execution where within Δ units after the bus change, *counter* values for unaborted units are chosen in the order of *counter* values at the $(i+1)$th stable state of σ. Hence, the inductive hypothesis holds. □

We obtain the following theorem as a corollary:

Theorem 5.2 (Delay Insensitivity) *$P(n,d)$ is correct for every even d, $d \geq 2$, iff $P(n,2)$ is correct.*

Proof.

The direction from left to right follows by instantiating d with 2. For the direction from right to left, note that if $P(n,d)$ is incorrect for some d, then it contains a computation where the sequence of symbols on the bus is not the maximum of the sent messages. By the previous theorem, this computation can be simulated by one in $P(n,2)$, so $P(n,2)$ is incorrect. □

Proof of Lemma 5.2:

Note that at a stable state, all units have the same requested bus state, although they may be transmitting different symbols with differing lengths. In the interval between stable states, for any pair of units p, q, $|counter_p - counter_q| \leq \Delta$.

(i) The length of the interval is *Long*. Let p be the unit determining the new symbol. As the bus change occurs when p's *counter* value equals $Txmin(Long)$,

$Txmin(Long) - \Delta \leq counter_q \leq Txmin(Long) + \Delta$, for any unit q, i.e., $6.5\Delta \leq counter_q \leq 8.5\Delta$.

If q sends a symbol by a short pulse, as $Trmax(Short) < 6.5\Delta$, q aborts by

the time that the bus changes state. If q sends by a long pulse, its *counter* value remains in the interval [$Trmin(Long)$, $Trmax(Long)$] up to the next stable state, by which time the new bus state is perceived by q.

(ii) The length of the interval is *Short*. Let p be the unit determining the new symbol. As the bus change occurs when p's *counter* value equals $Txmin(Short)$,

$Txmin(Short) - \Delta \leq counter_q \leq Txmin(Short) + \Delta$, for any unit q, i.e., $2.5\Delta \leq counter_q \leq 4.5\Delta$.

If q sends by a long pulse, then as $Trmin(Long) = 6.5\Delta$, q aborts by the next stable state (which occurs in the interval $[3.5\Delta, 5.5\Delta]$). If q sends by a short pulse, its *counter* value remains in the interval [$Trmin(Short)$, $Trmax(Short)$] up to the next stable state, by which time the new bus state is perceived by q.

Hence, every unit sending a different length aborts, and every unit sending a symbol with the same length is live at the next stable state. □

5.3.2 Many-Process Verification

The delay insensitivity theorem (Theorem 5.2) shows that it is both necessary and sufficient to check every instance with delay 2 in order to check correctness for instances over all other delay values. While this eliminates consideration of the delay parameter, the reduced system is still infinite-state, as it is parameterized by the number of units taking part in the protocol.

Verification of this parameterized system can be carried out fully automatically using the algorithm described in [EN 96]. This algorithm is based on a synchronous *control-user* model, where the instances of the parameterized system consist of a fixed control process C, and many copies of a fixed user process U. The n-process instance can thus be described by $C \parallel U_1 \parallel \ldots \parallel U_n$, where $\parallel$ denotes synchronous composition. In the SAE-J1850 protocol, the control process models the behavior of the bus, while the user process models the behavior of a single

unit, together with some machinery for modeling the delays in detecting bus value changes.

The algorithm of [EN 96] constructs a finite-state "abstract graph" for such a control-user parameterized system which is an abstraction of the entire *family* of instances. The states of the abstract graph record only the state of the control process, and for each local user state, whether there exists at least one user process in that state. The Lemma below gives a way of checking safety properties of the family. Liveness properties may be checked in two ways : (a) As the abstract graph simulates every instance, if the liveness property holds of the abstract graph, then it holds of the family, (b) An algorithm is provided in [EN 96] for exactly determining whether the liveness property holds of every instance.

Lemma 5.3 *[EN 96] The abstract graph simulates every instance of the family. Every finite path in the abstract graph corresponds to a finite computation of some instance.*

The paper also shows how to check properties of the form $\bigwedge_i \mathsf{A}g(i)$ by reducing them, using symmetry arguments (cf. [ES 93],[CFJ 93]) to checking a property $\mathsf{A}g(0)$ of the control process in a modified control-user system, which has the same user process, but has $C' = C \parallel U$ as the new control process.

5.4 Implementation Details

The behavior of the bus and the units as specified in the protocol is coded as a SMV [McMillan 92] program. The transition relation of the abstract graph is generated automatically by a program which takes the specification of control and user processes (in C), and generates SMV code describing the *transition relation* of the abstract graph. This is done by enumerating the reachable local states for a single user process, then generating each transition of the abstract graph by inspection of

the local transitions in the unit. States of the abstract graph are represented by subsets of the local user state space. Each subset indicates the presence of at least one user process in that local state, as discussed in the previous section. Thus, for a local user transition $s \longrightarrow t$, the corresponding abstract graph transition adds t as a member of a abstract state following one that has s as a member.

For the singly parameterized system with $\Delta = 2$, each unit has 254 reachable states; thus, the number of Boolean variables needed to encode an abstract state is also 254 (subsets are encoded as a boolean membership vector). The correctness properties C2(a) - C2(e) were checked together on the abstract graph. Since some of these properties are liveness properties, they were checked on the abstract graph using the fact that it simulates every instance. Every property succeeds on the abstract graph, so that we can infer that properties C2(a) - C2(e) hold of the parameterized system with delay 2, which by Theorem 5.2 implies that they hold of the completely parameterized system. By Lemma 5.1, this implies that the desired correctness property, (C1), holds of the completely parameterized system. We did not have to invoke the potentially expensive but exact method for checking liveness properties.

These checks take about 8 MB and 35 seconds on an Intel Pentium 133 with 32 MB of main memory. Conjunctive partitioning of the transition relation and pre-computation of the reachable states (the strongest invariant) is used. 24 iterations are needed to compute the reachable state space. Incidentally, checking a 15 unit instance takes roughly the same amount of time but less space.

5.5 Conclusions and Related Work

Verification of parameterized systems is often done by hand, or with the help of a mechanical theorem prover (cf. [CM 88], [MP 92], [HS 96]). Several methods have been proposed that, to various degrees, automate this verification process. Meth-

ods based on manual construction of a process invariant are proposed in [CG 87], [SG 89], [KM 89], [WL 89], [LSY 94], and have been applied for the verification of the Gigamax cache consistency protocol in [McMillan 92]. These constructions have been partially automated in [RS 93], [CGJ 95] (cf. [Vernier 93],[PD 95],[ID 96]); however, as the general problem is undecidable [AK 86], it is not in general possible to obtain a finite-state process invariant. For classes of parameterized systems obeying certain constraints, [GS 92], [EN 95], [EN 96] give algorithms (i.e., decision procedures) for model-checking the parameterized system. These papers demonstrate the methods on simple verification examples; we believe that our case study is one of the few examples of verification of a large and complex parameterized protocol. It is likely that the delay insensitivity theorem is an instance of a general theorem for such types of systems; given such a theorem, the verification of this protocol could be indeed fully automated.

We believe that careful specification of the computational model underlying other protocols will expose constraints that can be utilized, as in this case, for developing decision procedures for large classes of protocols. There is also a need for developing and popularizing notations for expressing such protocols. Remarkably, in the SAE-J1850 document (over 100 pages), there is no succinct protocol description; the description given in Section 5.2 had to be culled from the entire text. The successful verification of the protocol, despite the theoretical result on PSPACE-completeness of the procedure [EN 96], is reason to believe that fully automated parameterized verification is feasible for reasonably sized protocols.

Chapter 6

On Model Checking Infinite State Systems

6.1 Introduction

Model Checking refers to a collection of algorithms for automatically checking temporal properties of *finite* state systems [CE 81, QS 82, CES 86, LP 85]. Model Checking is well established as a verification method in large part because it is fully automated, and because most model checking algorithms produce a counter-example if the correctness property does not hold of the system. Inspired by the success of Model Checking, there is now increased attention to developing such algorithms and procedures for *infinite* state systems.

This effort is motivated by two tasks : the verification of systems parameterized by the number of processes, such as distributed protocols, and the verification of a fixed set of processes communicating over unbounded channels. Such systems are commonplace and generate an infinite state space, in the first case from the infinite number of instances, each with a fixed number of processes, and in the second case from the unboundedness of the communication channels. State explosion,

the limiting factor to the practical application of Model Checking algorithms, often arises in such parameterized systems for large instances. Thus, a solution for the general problem also helps ameliorate state explosion.

Algorithms are known for model-checking many types of infinite-state systems: Petri Nets [Esparza 94], asynchronous parameterized protocols [GS 92], timed automata [AD 91], hybrid systems [Henzinger 95], parameterized token-ring protocols [EN 95] and parameterized synchronous protocols [EN 96]. For other types of systems, semi-algorithmic procedures have been proposed [PD 95, ID 96, BG 96, BGWW 97].

In an interesting recent paper [ACJT 96] (cf. [Finkel 90]), it is pointed out that programs in these formalisms induce "well-quasi-ordered" (*wqo*) transition systems. This condition, together with additional properties, is shown to make the model-checking of *safety* properties decidable. The question of deciding general liveness properties is, however, left open. The algorithm for safety properties in [ACJT 96] computes $\mathsf{EF}\, bad$ as a fixpoint, i.e., in a "backward" direction, starting with the set of states where *bad* holds. More recently, [KMMPS 97] explores the use of automata as representations for infinite sets of states in general fixpoint computations.

Many of the known algorithms, however, are based on a "forward" search, starting with the set of initial states [GS 92, Esparza 94, BG 96, EN 96]. Algorithms based on quotients w.r.t. a bisimulation equivalence [AD 91, Henzinger 95] are also forward searches as the incremental computation of the quotient structure proceeds in a forward direction.

In this chapter, we propose a new type of "covering graph" construction as a general method of forward search for Model Checking. The covering graph construction for Petri Nets is developed by Karp and Miller [KM 69], where it is used to decide covering and boundedness questions. Finkel [Finkel 90] generalized this

construction to arbitrary deterministic *wqo* systems to decide the same properties.

We show that neither the well-quasi-ordering, nor the restriction to deterministic systems is necessary for checking *safety* properties. The essential feature of the covering graph construction is the use of a simulation preorder on the infinite state space. The simulation relation is used to detect potentially infinite paths, and "compress" them by replacing the path with the least upper bound (w.r.t. the simulation preorder) of the set of states occurring on the path. We prove that each node of the covering graph has an associated set of reachable states, which makes it possible to model-check safety properties. For liveness properties, well-quasi-ordering of the simulation relation has an important consequence : we show that there is a *finite* witness for the satisfaction of $\mathsf{E}h$ formulas, where h is a linear-time temporal property. The finite witness can be searched for in the covering graph, given an algorithm for determining if a strongly connected component is "good"; i.e., has the "tail" of the witness path embedded in it. Both results apply to general *non-deterministic* systems, thus providing a framework under which to explore the decidability of model-checking for non-deterministic infinite-state systems.

Although termination is not guaranteed in general (cf. [Finkel 90]), many of the forward search algorithms [AD 91, GS 92, EN 96] can be developed in a simple, uniform fashion from the new construction. Furthermore, the construction exposes the key ideas common to these algorithms and other procedures [PD 95, BG 96]. Despite the wide variety among these formalisms, these model-checking procedures are based on common high-level ideas. This is a strong indication that the covering graph construction is appropriate for analysis of infinite-state systems, even for computation models that are Turing-powerful for which termination cannot be guaranteed. We also consider the new application domain of parameterized broadcast protocols. The new approach is illustrated on the verification of an invalidation-based (MESI) cache coherency protocol, which is shown, fully automatically, to

satisfy correctness properties for an *arbitrary* number of processes.

The rest of the chapter is structured as follows. Section 6.2 contains preliminaries; Section 6.3 introduces the covering graph construction and the model-checking procedure for safety properties. Section 6.4 deals with liveness properties. In Section 6.5, we describe how many known algorithms may be derived uniformly from the construction, and introduce the application domain of parameterized broadcast protocols. Section 6.6 concludes the chapter with a discussion of related work and future directions. Some technical lemmas are presented in Section 6.7.

6.2 Preliminaries

Infinite-state systems are represented as Labeled Transition Systems and linear temporal properties by automata on infinite strings. This section contains the definitions of these concepts and their basic properties. Quantified expressions are written in the format $(\mathbf{Q}x : r : p)$, where $\mathbf{Q}$ is the quantifier, x the bound variable, r the range, and p the expression being quantified. The powerset of a set S is denoted by $\mathcal{P}(S)$.

6.2.1 Quasi orders and Partial orders

A binary relation $\preceq$ on set S is said to be a *quasi-order* (or *preorder*) if it is reflexive and transitive. The pair $(S, \preceq)$ is called a *preset* (for preordered set). If $\preceq$ is symmetric it is an equivalence relation, and if it is antisymmetric it is a *partial order* and $(S, \preceq)$ is called a *poset*.

In a poset $(S, \preceq)$, an element c is an *upper bound* of a subset X iff $(\forall x : x \in X : x \preceq c)$. The *least upper bound* (*lub*) of X, if it exists, is the upper bound of X that is the minimum w.r.t. $\preceq$ among the set of upper bounds of X. A subset X is *directed* iff any pair of elements in X has an upper bound in X. A a function $C : \mathbf{N} \to S$ is a chain iff for every $i \in \mathbf{N}$, $C(i) \preceq C(i+1)$. The set of elements of

a chain C, $\hat{C}$, equals $\{C(i)|i \in \mathbf{N}\}$. The poset is a *complete partial order* (*cpo*) iff every directed subset has a least upper bound.

Definition 6.1 (Well-Partial-Order, *wpo* **)** *For a poset* $(S, \preceq)$, $\preceq$ *is a well-partial-order iff for every infinite sequence* σ *of elements of* S, *there exist positions* $i, j \in \mathbf{N}$ *such that* $i < j$ *and* $\sigma_i \preceq \sigma_j$.

A preorder $(S, \preceq)$ is a *well-quasi-order* iff the induced partial order is a *wpo*.

Proposition 6.1 *(cf. [Fraisse 86]) For a preset* $(S, \preceq)$, $\preceq$ *is a wqo iff every infinite sequence of elements of* S *contains an infinite sub-sequence that is a chain.*

6.2.2 Ordered transition systems

Labeled transition systems (LTS's) are defined in Chapter 2. We will work with LTS's that have a finite number of actions and where the state labels are derived from a finite set of atomic propositions, AP. The label of each state is a subset of AP; i.e., the propositions true at that state. Such an LTS is written as a structure (S, Σ, R, I, AP, L). We write $s \xrightarrow{a} t$ instead of $(s, a, t) \in R$, $s \rightarrow t$ for $(\exists a : a \in \Sigma : s \xrightarrow{a} t)$, $s \xrightarrow{*} t$ if t is reachable from s, and $s \xrightarrow{+} t$ if t is reachable in at least one step from s. For sequences of symbols from Σ, the function $: \mathcal{P}(S) \rightarrow \mathcal{P}(S)$ is defined recursively by $\overline{\epsilon}(X) = X$, $\overline{a}(X) = \{t|(\exists s : s \in X : s \xrightarrow{a} t)\}$, and $\overline{\alpha; a}(X) = \overline{a}(\overline{\alpha}(X))$. Informally, $\overline{\alpha}(X)$ is the set of states reachable from states in X by performing the actions of α in order.

Definition 6.2 (Simulation) *A relation* $\lhd$ *on* S *is a simulation on* A *iff for any states* s, t *such that* $s \lhd t$, $L(s) = L(t)$, *and for every* a, u *such that* $s \xrightarrow{a} u$, *there is* v *such that* $t \xrightarrow{a} v$ *and* $u \lhd v$.

Definition 6.3 (Ordered LTS) *The pair* $(A, \preceq)$ *is called an ordered LTS iff* $\preceq$ *is a* simulation *on* A *and a wqo on* S.

Several important ways of specifying systems give rise to ordered LTS's:

1. Finite LTS's with the identity preorder, as any infinite path contains a repeated state.

2. Vector Addition Systems (VAS) [KM 69], as the component-wise ordering of vectors over $\mathbf{N}^k$ (the state space of a VAS), given by $u \preceq v$ iff $(\forall i : i \in [0,k) : u_i \leq v_i)$, is a *wpo*.

3. Petri Nets and Vector addition systems with states (VASS) [Reisig 85] are equivalent to VAS's.

4. Real-Time Automata [AD 91], as the bisimulation equivalence on clock values has finite index.

5. Finite state machines communicating over restricted FIFO channels [FR 88] or lossy channels [AJ 93], as the orderings on channel words are *wqo*'s.

6. Parameterized protocols [GS 92, EN 96], where the state space is encoded either as a VASS [GS 92] or by constraints [EN 96].

Büchi automata (see Chapter 2) are used to specify temporal correctness properties. Propositional linear temporal formulae can be translated into equivalent Büchi automata (cf. [Thomas 90]). We adopt the automata-theoretic approach to Model Checking [VW 86], in which the negation of the correctness property is expressed by a Büchi automaton $\mathcal{B}$, and every computation of an LTS A satisfies the correctness property iff the set of computations of the "product" $\mathcal{C} = \mathcal{B} \times A$ is empty. Given a simulation preorder $\preceq$ on A, define $\preceq'$ on C by $(b,s) \preceq' (c,t)$ iff $b = c$ and $s \preceq t$.

Proposition 6.2 *If $(A, \preceq)$ is an ordered LTS, then $(\mathcal{C}, \preceq')$ is an ordered fair LTS.*

Proposition 6.3 *Accepting runs of $\mathcal{B}$ over A correspond to computations of $\mathcal{C}$.*

6.3 Model-Checking Safety Properties

The negation of a linear-time safety property can be expressed as an automaton on *finite* strings, instead of a Büchi automaton. The system satisfies the safety property iff no finite path of the system is accepted by the automaton, which holds iff in the combined system there is no path to an accepting state.

The Karp-Miller construction for Petri Nets [KM 69] is a sophisticated version of the naive reachability procedure. The key idea is to "compress" paths that are potentially infinite, using a simulation preorder. We develop below a new generalization of the Karp-Miller construction geared towards Model Checking.

We work with an LTS $A = (S, \Sigma, R, I, AP, L)$, where Σ and AP are both *finite*. As Σ and AP are finite, one can augment the edge labeling to $\Sigma \times AP$, where $s \stackrel{(a,l)}{\rightarrow} t$ in the new labeling iff $s \stackrel{a}{\rightarrow} t$ and $L(t) = l$. This relabeling induces the following *labeling property* : For any subset X of states, and a new action label (a, l), states in $\overline{(a,l)}(X)$ have identical state labels. For the rest of the paper, the system is assumed to have the labeling property.

Let a *uniform* subset of S be a set where any two members have the same state label. The set of uniform subsets of S is denoted by $\mathcal{U}(S)$. The label of a uniform subset X, denoted by $\Lambda(X)$, is the common label of its members if the set is non-empty and a special value $\perp$ if the set is empty. Let $\sqsubseteq$ be a relation on $\mathcal{U}(S)$ with the following properties :

1. $\sqsubseteq$ is a *pre-order*, such that if $X \sqsubseteq Y$ then $\Lambda(X) = \Lambda(Y)$. Let $\approx$ denote the equivalence generated by $\sqsubseteq$. I.e., $X \approx Y$ iff $X \sqsubseteq Y$ and $Y \sqsubseteq X$.

2. For any action symbol a, $\overline{a}$ is monotonic w.r.t. $\sqsubseteq$. I.e., for any X, Y, $X \sqsubseteq Y$ implies $\overline{a}(X) \sqsubseteq \overline{a}(Y)$.

3. $(\mathcal{U}(S)/\approx, \sqsubseteq)$ is a *complete partial order* (it cpo).

4. For any chain C over $\mathcal{U}(S)$, *lub* $\hat{C} \approx \bigcup \hat{C}$.

5. For any s and any uniform subset X, if $s \in X$ then $\{s\} \sqsubseteq X$. If $(A, \preceq)$ is an ordered LTS and $\{s\} \sqsubseteq X$, there exists $t \in X$ such that $s \preceq t$.

Lemma 6.1 *$X \approx Y$ implies $\overline{a}(X) \approx \overline{a}(Y)$.*

Lemma 6.2 *Let C and D be chains such that for every i in $\mathbf{N}$, $C(i) \approx D(i)$. Then lub $\hat{C} \approx$ lub $\hat{D}$.*

The proofs of these Lemmas are quite straightforward and are deferred to the appendix.

6.3.1 The Covering Graph Procedure

The procedure constructs a covering graph incrementally. Each node n of the graph is labeled by a non-empty uniform subset, which is denoted by $\mathsf{L}(n)$. The subsets are usually infinite so, in practice, finite representations and methods of manipulating such representations are needed. We describe some such representations in Section 6.5; the properties of the algorithm are independent of the representation method.

The function *rep* maps a uniform subset to its representative, so that for any subset X, $rep(X) \approx X$. The graph is constructed by the following nondeterministic procedure. *New* is the set of unexamined (node, edge label) pairs.

Begin

- Choose a *finite* partition of the set of initial states into uniform subsets. For each set X in the partition, an initial node n of the covering graph is created with $\mathsf{L}(n) = rep(X)$. For each $a \in \Sigma$, add (n, a) to *New*.
- Repeat the following process as long as *New* is non-empty :

  Choose and remove a pair (n, a) from *New*. Let $Y = \overline{a}(\mathsf{L}(n))$. If $Y \not\approx \emptyset$, perform these actions in order:

(Cover) If there is a node m such that $Y \sqsubseteq \mathsf{L}(m)$: make m the a-successor of n.

(Limit) If there is a predecessor k of n on a path γ, with $\mathsf{L}(k) = Z$, such that $\mathsf{L}(n) \approx \overline{\gamma}(Z)$ and $Z \sqsubseteq Y$:

Let $\beta = \gamma; a$. Define C by $C(i) = \overline{\beta^i}(Z)$ for $i \in \mathbf{N}$. Create a node m labeled with $rep(W)$ as the a-successor of n, where $W = lub\ \hat{C}$. For each $a \in \Sigma$, add (m, a) to *New*.

(Step) Create a node m labeled with $rep(Y)$ as the a-successor of n. For each $a \in \Sigma$, add (m, a) to *New*.

End

The Covering Graph Construction Procedure.

In the second alternative, note that $\overline{\beta^0}(Z) = Z \sqsubseteq Y \approx \overline{\beta^1}(Z)$. From this initial condition and the monotonicity of $\overline{\beta}$ (property (2) of $\sqsubseteq$), it follows that C is indeed a chain, and by property (3), *lub*'s of chains exist. The construction procedure is nondeterministic, so several possible covering graphs may be generated. The theorems below hold for every such graph.

Theorem 6.1 *For every node n, there is a non-empty reachable set of states R*

such that $\mathsf{L}(n) \approx R$.

Proof. We prove that this property is an invariant of the procedure. It holds of the initial nodes by their definition and the property of *rep*.

Assume that the property holds at the beginning of an iteration. Let (n, a) be the choice from *New*.

(1) The first alternative is taken. As the set of nodes of the covering graph is not changed, the invariant holds.

(2) The second alternative is taken and a new node is added with label $rep(W)$. From the invariant, there is a non-empty, reachable subset of states R such that $Z \approx R$. So for the new node m,

$rep(W)$

$\approx$ (by definition of C and property of *rep*)

$lub\ \{\overline{\beta^i}(Z) | i \in \mathbf{N}\}$

$\approx$ (from Lemma 6.1 $\overline{\beta^i}(Z) \approx \overline{\beta^i}(R)$; Lemma 6.2)

$lub\ \{\overline{\beta^i}(R) | i \in \mathbf{N}\}$

$\approx$ (property (4) of $\sqsubseteq$)

$\bigcup\{\overline{\beta^i}(R) | i \in \mathbf{N}\}$

$\bigcup\{\overline{\beta^i}(R) | i \in \mathbf{N}\}$ is a set of reachable states by definition, and is non-empty as it has R as a subset.

(3) The third alternative is taken and a new node is added with label $rep(Y)$. From the invariant, there is a non-empty reachable subset of states R such that $\mathsf{L}(n) \approx R$. So for the new node m,

$rep(Y)$

$\approx$ (by property of *rep*)

Y

$\approx$ (by definition of Y)

$\overline{a}(\mathsf{L}(n))$

$\approx$ (from Lemma 6.1)

$\overline{a}(R)$

$\overline{a}(R)$ is a reachable set of states as R is reachable. As $Y \not\approx \emptyset$, $\overline{a}(R)$ is non-empty. $\square$

Definition 6.4 (Covering simulation) *Let $\lhd$ be the relation defined between the LTS A and the covering graph by $s \lhd n \equiv \{s\} \sqsubseteq \mathsf{L}(n)$.*

Theorem 6.2 *Every Covering Graph simulates the underlying LTS by $\lhd$.*

Proof.

Suppose $s \lhd n$ and $\mathsf{L}(n) = X$. By property (1) of $\sqsubseteq$, $L(s) = \Lambda(X)$. Let t be any state such that $s \xrightarrow{a} t$. Then $t \in \overline{a}(\{s\})$, hence $\overline{a}(\{s\})$ is non-empty. By property (5) of $\sqsubseteq$, $\{t\} \sqsubseteq \overline{a}(\{s\})$. By property (2) of $\sqsubseteq$, $\overline{a}(\{s\}) \sqsubseteq \overline{a}(X)$. As $\{t\} \sqsubseteq \overline{a}(X)$, $\overline{a}(X) \not\approx \emptyset$, so n has a successor m on action a.

An invariant of the procedure is that for any edge (k, a, l) in the graph, $\overline{a}(\mathsf{L}(k)) \sqsubseteq \mathsf{L}(l)$. It follows that $\overline{a}(\{s\}) \sqsubseteq \mathsf{L}(m)$, so $\{t\} \sqsubseteq \mathsf{L}(m)$ and $t \lhd m$. This proves that $\lhd$ is a simulation relation. $\square$

Several choices for $\sqsubseteq$ have the necessary properties:

- $\sqsubseteq$ is the subset relation.
- Let $\preceq$ be a simulation relation on the LTS that is a pre-order. Then, $X \sqsubseteq Y$, defined as $(\forall s : s \in X : (\exists y : y \in Y : x \preceq y))$, is a pre-order that satisfies the conditions.

- The same preorder restricted to *directed* subsets, is appropriate for deterministic LTS's. For non-deterministic LTS's, a stronger form of directedness is needed, which is discussed in Section 6.7.

Theorem 6.3 (Model-Checking Safety Properties) *There is a reachable accepting state of the combined system iff there is a covering graph node labeled as accepting.*

Proof.

For any reachable accepting state s, by Theorem 6.2, there is a node n in the covering graph such that $\{s\} \sqsubseteq \mathsf{L}(n)$. As $\Lambda(\{s\}) = \Lambda(\mathsf{L}(n))$ by property (1) of $\sqsubseteq$, n is an accepting node.

Conversely, by Theorem 6.1, for every reachable node n of the Covering Graph, there is a non-empty reachable subset R such that $\mathsf{L}(n) \approx R$. By property (1) of $\sqsubseteq$, $\Lambda(\mathsf{L}(n)) = \Lambda(R)$, so that every state in R is accepting. □

6.4 Model-Checking Liveness Properties

Checking if a finite-state fair LTS has a computation is straightforward : the NLOGSPACE algorithm searches for a "looping" path of the form $s_0 \xrightarrow{\alpha} s \xrightarrow{\beta} s$, where s is a fair state (cf. [SVW 87]). The following theorem shows that the concept of a finite "looping" path is easily extended to the concept of a finite "self-covering" path for an ordered fair infinite-state system. Put differently, ordered fair LTS's have a *finite* witness for non-emptiness of the set of computations, even though the LTS may have an infinite number of states.

Definition 6.5 (Self-covering fair path) *A self covering fair path in an ordered fair LTS $(A, F, \preceq)$ is a finite path of the form $s \xrightarrow{*} t \xrightarrow{+} u$, where s is an initial state, $t \preceq u$, and $t \in F$.*

For the rest of this section, let $(A, \preceq)$ be an ordered LTS, and $\mathcal{B}$ a Büchi automaton. Let $\mathcal{C}$ represent the product of $\mathcal{B}$ and A.

Theorem 6.4 *There is an accepting run of $\mathcal{B}$ on A iff there is a self-covering fair path in $\mathcal{C}$.*

Proof.

($\Rightarrow$) By Proposition 6.2, $\mathcal{C}$ is an ordered fair LTS, with the induced preorder $\preceq'$ defined by $(b, s) \preceq' (c, t)$ iff $b = c$ and $s \preceq t$.

Let σ be an accepting run of $\mathcal{B}$ over A. From Proposition 6.3, σ is a computation of $\mathcal{C}$. As $\mathcal{B}$ is finite-state, some accepting state b from $\mathcal{B}$ appears infinitely often along σ. Let δ be the infinite subsequence of σ obtained by retaining those states with automaton component b.

As $\mathcal{C}$ is ordered, by Proposition 6.1, δ has an infinite subsequence γ that is a chain w.r.t. $\preceq'$. Thus, there exist distinct states t, u on δ such that $t \preceq' u$ and t is a fair state. Since t is reachable from the initial state s of σ, $s \xrightarrow{*} t \xrightarrow{+} u$ forms a self-covering fair path in $\mathcal{C}$.

($\Leftarrow$) Let $s \xrightarrow{x} t \xrightarrow{y} u$ be a self covering fair path in $\mathcal{C}$. As $t \preceq' u$ and $\preceq'$ is a simulation, for some v, $u \xrightarrow{y} v$ with $u \preceq' v$. Continuing in this manner, define an infinite path labeled by y^ω from $w_0 = t$, where for every i, $w_i \xrightarrow{y} w_{i+1}$ and $w_i \preceq' w_{i+1}$. As w_0 is a fair state, and $w_0 \preceq' w_i$ for every i, it follows from the definition of $\mathcal{C}$ that each w_i is a fair state. Hence the sequence of transitions $x; y^\omega$ induces an infinite path from s that is infinitely often fair. By Proposition 6.3 this computation is an accepting run of $\mathcal{B}$ on A. □

Theorem 6.4, when combined with the automata theoretic approach to Model Checking [VW 86], transforms the Model Checking problem for an ordered LTS to determining if a self-covering fair path exists in the ordered fair LTS formed by the product of the LTS with the Büchi automaton for the negation of the correctness property.

Definition 6.6 (Positive sequence) *A finite sequence of transitions σ of A is called* positive *for s iff $(\exists t : s \xrightarrow{\sigma} t : s \preceq t)$. An ordered LTS has the* positive path *property iff whenever σ is positive for s, for any u such that $s \preceq u$, there exist v and j such that $j > 0$, $u \xrightarrow{*} v$, σ^j is positive for v, and v is fair if u is fair.*

Note that every VASS has the positive path property. Any sequence σ that is positive for a state s has non-negative vector sum. Hence, for every u such that $s \preceq u$, σ is positive for u; i.e., $j = 1$ and $u = v$ in the definition above.

Definition 6.7 (Good SCC) *A strongly connected component (SCC) of the covering graph is* good *iff it contains a fair node n such that there is a finite path in the component from n which is positive for a state s such that $s \lhd n$.*

Theorem 6.5 *For any finite covering graph of C, any self-covering fair path in C induces a good SCC in the covering graph.*

Proof.

Let $s \xrightarrow{*} t \xrightarrow{\sigma} u$ be a self covering path in C. As $t \preceq u$, σ is a positive path for t.

Consider an infinite path ρ labeled with σ^ω starting at t (such a path exists; cf. the proof of Theorem 6.4). Let m be a node in the covering graph such that $t \lhd m$ (m exists from Theorem 6.2). By Theorem 6.2, σ^ω induces an infinite path η through the covering graph from m. Consider the set of nodes of the covering graph occurring on η after each prefix σ^i for $i \in \mathbf{N}$. Since the covering graph is finite, there is a repeated node n in this set. Let $m \xrightarrow{\sigma^l} n \xrightarrow{\sigma^k} n$ be the prefix of η up to the second occurrence of n. Let u be the state on ρ after the prefix σ^l. From these definitions $u \lhd n$, hence n is a fair node of the covering graph. From the construction of ρ, σ^k is positive for u. The cycle induced by σ^k in the covering graph from n defines a good SCC of the covering graph. □

Theorem 6.6 *If C has the positive path property, then a good SCC in the covering graph induces a self-covering fair path in C.*

Proof. Let n be the state in a good SCC from which there is a finite path σ that is positive for a state s such that $s \lhd n$. From Theorem 6.1, there is a non-empty reachable set of states R such that $\mathsf{L}(n) \approx R$. By the definition of $\lhd$ and the transitivity of $\sqsubseteq$, $\{s\} \sqsubseteq R$.

By Property (5) of $\sqsubseteq$, there is a state t in R such that $s \preceq t$. By the positive path property, there exist u, j such that $t \xrightarrow{*} u$ and σ^j $(j > 0)$ is positive for u, which implies that there is a state v such that $u \xrightarrow{\sigma^j} v$ with $u \preceq v$. Since t is fair, so is u. Since t is reachable from some initial state w, $w \xrightarrow{*} t \xrightarrow{*} u \xrightarrow{+} v$ forms a self-covering fair path in C. □

Specific choices for the simulation relation, and the representation of subsets for the construction are discussed in the following section. To check if a property specified by a Büchi automaton $\mathcal{B}$ for its negation holds for an ordered LTS A, one must

1. Define the product $\mathcal{C} = \mathcal{B} \times A$.
2. Pick an appropriate relation $\sqsubseteq$, and construct a finite covering graph.
3. As the covering graph simulates $\mathcal{C}$, one may determine if the property holds by checking it on the covering graph. If this fails, an algorithmic test to determine if an SCC of the covering graph is good is required, provided that $\mathcal{C}$ has the positive path property.

Theorem 6.7 (Model-Checking Liveness Properties) *Let A be an ordered LTS and $\mathcal{B}$ be a Büchi automaton for the negation of the correctness property such that $\mathcal{B} \times A$ has a finite covering graph. If $\mathcal{B} \times A$ has the positive path property, A is correct iff the covering graph does not contain a good SCC.*

6.5 Applications

6.5.1 Parameterized Systems

Many distributed protocols are specified as a system parameterized by the number of instances of identical processes. The processes are usually finite-state so that each instance is finite, but there is an infinite number of instances whose disjoint union forms an infinite-state system. Model-checking a parameterized system is undecidable in general [AK 86].

A commonly studied type is a control-user system, where each instance contains a single copy of the control process and a specified number of user process copies. A state of an instance is represented by a vector, with the first component being the control state and the other components indicating the number of user processes in each user state. Vectors are ordered by the usual component-wise partial ordering. For the parameterized systems studied in [GS 92] and [EN 96], this ordering is a simulation relation. In [GS 92] the parameterized system is modeled by a VASS and the Model Checking algorithm is based on Rackoff's [Rackoff 78] near-optimal algorithm for detecting self-covering paths. As the covering graph construction is effective for VASS's, it provides an alternative algorithm, although of higher worst-case complexity. In [EN 96] a synchronous composition operation is defined, which makes it impossible to model the system as a VASS. The analysis is performed with a finite "abstract graph" which is, in fact, a specialization of the covering graph construction presented in this paper.

In both cases, it is possible to recognize good SCC's algorithmically. For the reduction to a VASS, Rackoff's procedure [Rackoff 78] may be used. The algorithm for detecting good SCC's in [EN 96] uses a threading construction, which resolves a cycle in the covering graph into "threads" that indicate how processes move from one local state to another. Analysis of this threaded cycle can determine whether the cycle represents a positive path for a state covered by the initial node.

It is usually the case that correctness properties for parameterized systems are of the forms : "every process i satisfies $f(i)$" or "every distinct pair of processes (i, j) satisfies $f(i, j)$". Such properties may be reduced to checking properties of the control process of a modified system using symmetry results from [ES 93, CFJ 93].

Broadcast Protocols

The broadcast model is appropriate for analyzing bus-based hardware protocols such as those for cache coherency. For simplicity, we consider protocols where the state change in response to a broadcast is deterministic.

The system is defined as a control-user system with an interleaving composition rule. As in [GS 92, EN 96], the global state is represented by a vector. Local transitions of a process and synchronizations between pairs of processes can be represented as vector additions [GS 92], while broadcast moves are represented as matrix transforms. For instance, consider a broadcast specified by

- The broadcast send $(a!)$: $s \xrightarrow{a!} t$, and
- The corresponding deterministic broadcast receptions $(a?)$: $s \xrightarrow{a?} u$, $t \xrightarrow{a?} u$, and $u \xrightarrow{a?} s$.

This synchronized broadcast action may be represented by a set of simultaneous equations defining the number of processes in each local state after the broadcast. For a global state G, let $G.s$ represent the number of processes in local state s in G. For a transition from G to H with the broadcast action specified above, the equations are : $H.s = G.u; H.t = 1; H.u = (G.s - 1) + G.t$. Informally, the single process that broadcasts moves from s to t; the processes receiving in state s move to state u. Such transformations may in general be represented by $H = T(G)$, where $T(X) = M(X) + C$ for a 0-1 matrix M with unit vectors as columns. M has this special structure as each state occurs on the r.h.s. in exactly one equation. For

local transitions and pairwise synchronizations M is the identity matrix, so that T reduces to a vector addition. The guard of a transform is given by the conjunction of terms $x > 0$ for each variable x that is decremented by the transform; e.g., the guard for the transform above is $s > 0$. The usual component-wise ordering on vectors is easily shown to be a simulation relation for such transforms.

Lemma 6.3 *For any matrix M of the form above, there exist $m, n \in \mathbf{N}$ such that $m < n$ and $M^m = M^n$.*

Proof. For matrices M, N of this type, every column of MN is a column of M. Thus every column of M^i, for any $i > 0$, is a column of M. Since there are only finitely many distinct arrangements of columns of M into matrices of the same size, there must exist m, n such that $m < n$ and $M^m = M^n$. □

With this Lemma, we can devise an effective procedure for computing the *lub*'s of the chains that arise in the covering graph construction. Let $T(X) = M(X) + C$ be a transform and v a vector. For any i, $T^i(X)$ equals (using distributivity of matrix application over vector sum) $M^i(X) + \Sigma_{j \in [0,i)} M^j(C)$. Let m, n be as in the lemma above, and let $\Delta = n - m$. For any k, $M^{m+k*\Delta} = M^m$. Hence, for $i = m + k * \Delta$, $T^i(X)$ equals $M^m(X) + \Sigma_{j \in [0,m)} M^j(C) + k * \Sigma_{j \in [m,n)} M^j(C)$.

Now suppose v is a vector such that $v \leq T(v)$. The set $\{T^i(v) | i \in \mathbf{N}\}$ forms a chain (with $T^0(v) = v$). The set $\{T^i(v) | i \in \mathbf{N} \wedge (i \ mod \ \Delta \equiv m)\}$ is an infinite subchain of this chain, so it has the same *lub* . By the argument above, this set equals $\{u + k * w | k \in \mathbf{N}\}$, where $u = M^m(v) + \Sigma_{j \in [0,m)} M^j(C)$, and $w = \Sigma_{j \in [m,n)} M^j(C)$. For the non-$\omega$ components of v the values in w must be non-negative, as the set is a chain. The representation of the *lub* is given by changing u_i to ω, for every i such that w_i is finite and non-zero. This procedure generalizes the standard limit construction for VASS's (where M is the identity, so $m = 0, n = 1$, which implies that $u = v$ and $w = C$.).

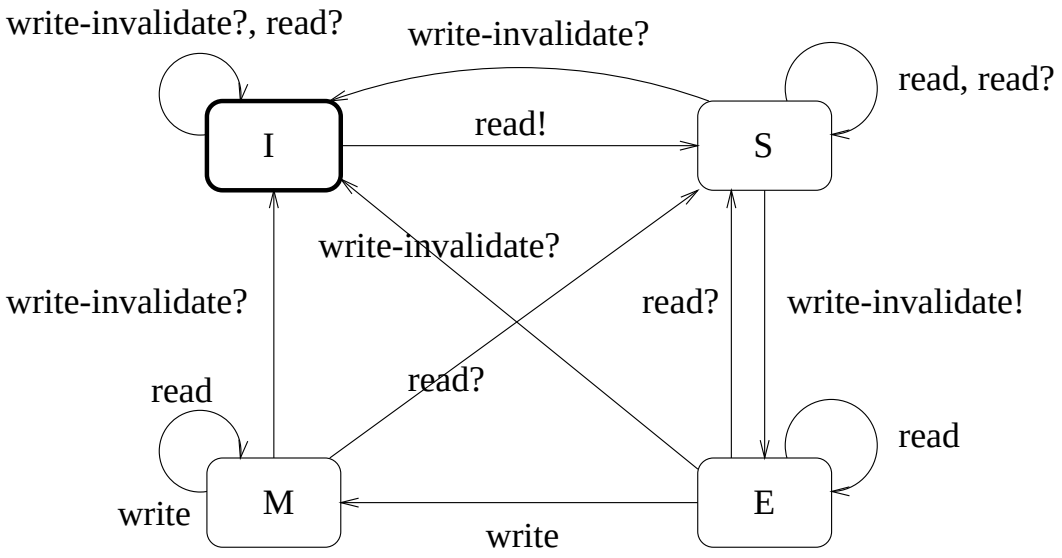


Figure 6.1: The MESI protocol

The protocol in Fig. 6.1 is a variation on the MESI protocol for cache coherency. We have modeled the synchronization mechanisms of a single address (cache line), ignoring the data stored at the address. The covering graph of Fig. 6.2 has initial state $(0, 0, 0, \omega)$ representing the set of initial states of the parameterized system, which has an arbitrary number of processes in state I. The covering graph may be used to prove several invariants of the protocol for every instance. For instance the readers-writers exclusion of S (shared) and M (modified) states, which may be written as $\mathsf{AG}(\#M * \#S = 0)$. Similarly mutual exclusion holds between the M and E states, in that $\mathsf{AG}(\#M + \#E \leq 1)$.

Pong and Dubois [PD 95] have analyzed several cache coherency protocols with an abstraction that keeps track of whether there is zero or at least one process in a given local state. The abstraction loses information in the sense that a violation of a safety property in the abstract graph is not necessarily a violation in the concrete system. The covering graph construction, however, is exact by Theorem 6.3.

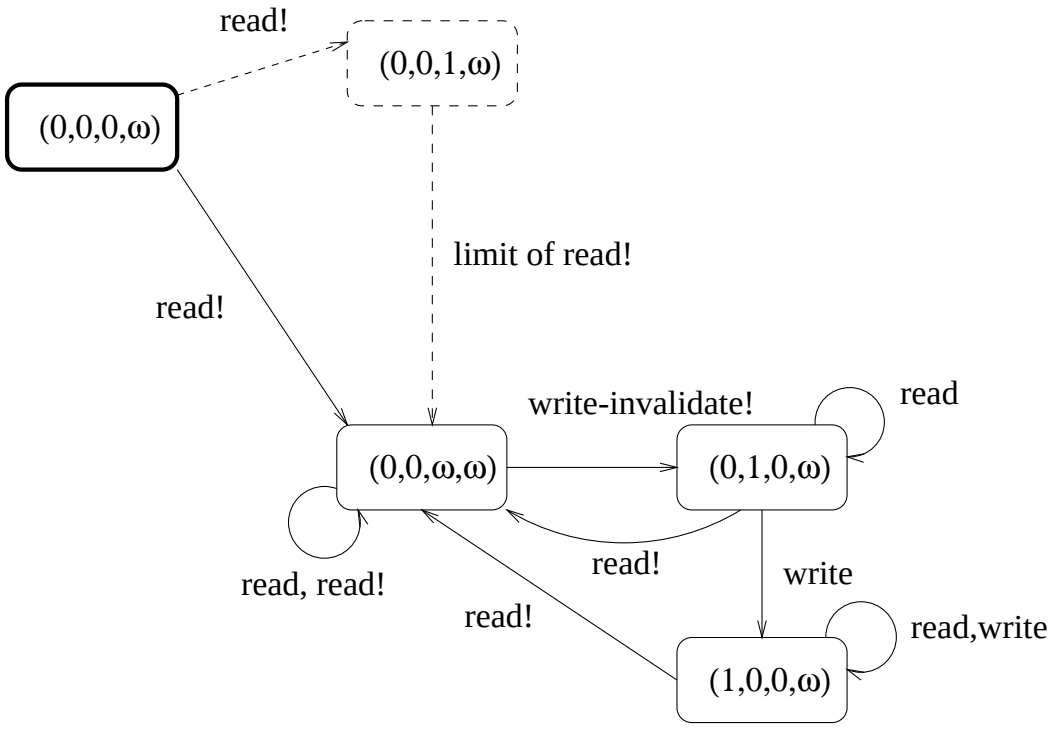


Figure 6.2: The covering graph. State vector has the form (M, E, S, I).

6.5.2 Real-Time Systems

In the parameterized systems discussed above, the equivalence relation induced by the simulation partial order does not have finite index. For some classes of systems, such as real-time and hybrid systems [AD 91, Henzinger 95], there is a bisimulation or simulation equivalence of finite index. In this case, the function *rep* of the covering graph procedure may be chosen so that $rep(X)$, for a subset X of equivalent states, is the equivalence class that the states in X belong to. With this definition, since there is a finite number of equivalence classes, the covering graph construction terminates.

In [BCG 89, EN 95], a method for verification of parameterized systems is proposed, which is to set up a family of bisimulations $\{B_n | n \geq m\}$ between instances of size $n \geq m$ and the instance of size m. If this is possible, then the correctness property holds of all instances iff it holds on instances of sizes at most m. Clearly, $B = \bigcup_{n \geq m} B_n$ is a bisimulation over the family of instances. In these papers, B is also an equivalence relation, and by the properties of B_n above, has finite index.

6.5.3 Communication protocols

[FR 88] consider systems of processes communicating with FIFO channels. They show that if the set of channel contents considered as words over an alphabet are prefixes of $u; v^*$ for some words u, v, a finite covering graph can be constructed for the protocol so that boundedness of channel contents and deadlock-freedom are decidable. Using the results in this paper, general safety properties of the finite state control for these protocols are also decidable. A related approach for dealing with communication protocols is proposed in [BG 96], where sets of reachable states are represented by deterministic automata over finite words.

6.6 Related Work and Conclusions

Among related work, Finkel [Finkel 90] generalizes the construction of Karp and Miller to *wqo* deterministic systems to solve boundedness and covering questions. As noted in the introduction, to ensure the relevant properties of every covering graph neither the restriction to deterministic systems nor the *wqo* property is essential.

Bradfield and Stirling [BS 90, Bradfield 92] use a tableau-based procedure for determining whether a μ-calculus property holds of an infinite-state system. They consider the use of the Karp-Miller graph for checking Petri Nets. As noted in [Bradfield 92], a property which is true of the Petri Net may not hold over a Karp-Miller graph for the Petri Net. The method presented here avoids this problem by adopting the automata theoretic approach and constructing the covering graph of the *product* of the system with the property automaton.

[ACJT 96] propose an interesting approach to the decidability of safety properties and the liveness property $\mathsf{AF}p$ for ordered LTS's. This is based on an abstract interpretation of the infinite state space in terms of upward closed subsets. Using their procedure, they can derive uniformly algorithms for deciding safety properties of many types of systems. It is not, however, possible to derive methods for deciding general liveness properties from their procedure.

Solutions to the problem of model-checking systems with infinite state spaces are the key to extending the applicability of model-checking procedures to parameterized systems, real-time systems, and communication protocols. General approaches to the problem have not been very well-studied, although many decidability results for specific classes are known. This paper provides such a unifying framework by demonstrating that a covering graph construction generates a graph which if finite, allows the checking of safety properties. In addition, we show that the decidability of general liveness properties in *wqo* systems, is linked to an algorithm for deciding the existence of finite self-covering fair paths. The covering

graph may be used to search for such paths. We also show that many of the known decidability results for infinite-state systems can be cast in these terms. This is a strong indication that the covering graph construction is appropriate for the analysis of infinite-state systems. We also consider a new application domain, that of parameterized broadcast protocols, and indicate how to apply the construction in this domain. This application is demonstrated on an invalidation based cache coherency protocol. These results, we hope, will motivate further applications of this procedure to a wide class of systems.

6.7 Technical Details

Lemma 6.1 *$X \approx Y$ implies $\hat{a}(X) \approx \hat{a}(Y)$.*

Proof.

$\overline{a}(X) \approx \overline{a}(Y)$

iff (definition of $\approx$)

$\overline{a}(X) \sqsubseteq \overline{a}(Y) \wedge \overline{a}(Y) \sqsubseteq \overline{a}(X)$

$\Leftarrow$ (property (2) of $\sqsubseteq$)

$X \sqsubseteq Y \wedge Y \sqsubseteq X$

iff (definition of $\approx$)

$X \approx Y$

$\square$

Lemma 6.2 *Let C and D be chains such that for every i in $\mathbf{N}$, $C(i) \approx D(i)$. Then lub $\hat{C} \approx$ lub $\hat{D}$.*

Proof.

For any Z,

$$lub\ \hat{C} \sqsubseteq Z$$

$\Rightarrow$ (definition of lub ; transitivity of $\sqsubseteq$)

$$(\forall i : i \in \mathbf{N} : C(i) \sqsubseteq Z)$$

$\Rightarrow$ ($D(i) \sqsubseteq C(i)$ for every i in $\mathbf{N}$; transitivity of $\sqsubseteq$)

$$(\forall i : i \in \mathbf{N} : D(i) \sqsubseteq Z)$$

$\Rightarrow$ (definition of lub)

$$lub\ \hat{D} \sqsubseteq Z$$

From this proof, and the reflexivity of $\sqsubseteq$, we can conclude that $lub\ \hat{C} \sqsubseteq lub\ \hat{D}$. A symmetric proof establishes the other direction. Hence, $lub\ \hat{C} \approx lub\ \hat{D}$. $\square$

6.7.1 Strongly Directed Sets

In Section 6.4, several choices for the preorder $\sqsubseteq$ are discussed. While $X \sqsubseteq Y$, defined as $(\forall s : s \in X : (\exists y : y \in Y : x \preceq y))$, is a preorder on directed subsets, it does not satisfy all the conditions on $\sqsubseteq$. In particular, for an action a, and directed subset X, $\overline{a}(X)$ may not be directed, if the LTS is nondeterministic. To satisfy this condition, we need to strengthen the directedness property.

Definition 6.8 (Dominance relation) *A relation Θ on $S^2 \times S$ (S is the set of states of the LTS A) is a dominance relation iff for any $((s,t),u) \in \Theta$,*

1. *$s \preceq u$ and $t \preceq u$, (u is an upper bound for s and t)*
2. *$(\forall s', t', a : s \xrightarrow{a} s' \wedge t \xrightarrow{a} t' : (\exists u' : u \xrightarrow{a} u' : ((s',t'),u') \in \Theta))$.*

As the conditions are monotone in Θ, there is a greatest dominance relation by the Knaster-Tarski theorem. We say that u dominates (s,t) iff $((s,t),u)$ is in the greatest dominance relation.

Definition 6.9 (Strongly directed set) *A subset X of S is strongly directed iff for any pair of states s,t in X, there is a state u in X such that u dominates (s,t).*

Any strongly directed set is directed. For a system where there is at most one outgoing edge with a given edge label, any directed set is strongly directed. For a non-deterministic system, we can show the following theorem:

Lemma 6.3 *If X is strongly directed, then $\overline{a}(X)$ is strongly directed.*

Proof.

Let x, y be an arbitrary pair of states in $\overline{a}(X)$. By definition, there exist u, v in X such that $u \xrightarrow{a} x$ and $v \xrightarrow{a} y$ holds. Since X is strongly directed, there is w in X that dominates the pair (u, v). By the definition of dominance, there is z such that $w \xrightarrow{a} z$ and z dominates the pair (x, y). Hence, z is in $\overline{a}(X)$ and dominates (x, y). Since x, y is an arbitrary pair of states, it follows that $\overline{a}(X)$ is strongly directed. $\square$

Chapter 7

Abstraction under Stuttering

7.1 Introduction

Showing equivalence between two systems at different levels of abstraction may entail mapping a single step in one system to a sequence of steps in the other, which is defined with a greater amount of detail. For instance, a compiler may transform the single assignment statement "$x := x*10+2$" to several low-level instructions. When proving correctness of the compiler, the single assignment statement step is matched with a sequence of low-level steps, in which the value of x remains unchanged until the final step. If the program state is defined by the values of program variables, then the intermediate steps introduce a finite repetition of the same state, a phenomenon called "stuttering" by Lamport [Lamport 80]. Stuttering arises in various contexts, especially as a result of operations that hide information, or refine actions to a finer grain of atomicity.

In [BCG 88, dNV 90], bisimulations that take into account such "stuttering" are defined. It is shown in [BCG 88] that states related by a stuttering bisimulation satisfy the same formulas of the powerful branching temporal logic CTL* [EH 82] that do not use the next-time operator, X. Although these definitions are well

suited to showing the relationship with CTL*, they are difficult to use in proofs of bisimulation, as they often require one to exhibit a finite, but unbounded sequence of transitions to match a single transition; thus introducing a number of proof obligations.

Determining whether an equivalence relation on a system is a bisimulation of some kind is important for abstraction. If the equivalence relation has finite index (i.e., finitely many equivalence classes), then a *quotient* system may be formed, where the new macro-states are equivalence classes, and two macro-states are related iff there exist states in each macro-state that are related in the original system. A sufficient condition for the quotient to be bisimilar to the original system is that the relation is a bisimulation on the original system. This implies in turn that properties preserved by the bisimulation may be model-checked on the smaller, finite quotient structure instead of the original large structure. Examples of the application of this general idea may be found in the theory of Symmetry Reduction [ES 93, CFJ 93], Real-Time Automata [AD 91] and Data Independence [Wolper 86, HB 95]. The kinds of temporal properties that are preserved depends on the kind of bisimulation used. For strong bisimulation, general μ-calculus properties are preserved, while for weaker notions of bisimulation such as stuttering bisimulation, properties in the next-time-free sublogic of CTL* are preserved.

The main contribution here is a simple alternative formulation of bisimulation under stuttering, called *well-founded bisimulation*, because is based on the reduction of a rank function over a well-founded set. The new formulation has the pleasant property that, like strong bisimulation [Milner 90], it can be checked by considering *single* transitions only. This substantially reduces the number of proof obligations, which is highly desirable in applications to infinite state systems such as communication protocols with unbounded channels or parameterized protocols, where checks of candidate relations are often performed by hand or with the

assistance of a mechanical theorem prover. We demonstrate the use of the new formulation with some non-trivial examples that have infinite state spaces and exhibit unbounded stuttering.

The use of rank functions and well-founded sets is inspired by their use in replacing operational arguments for termination of **do-od** loops with a proof rule that is checked for a single generic iteration (cf. [AO 92]). To the best of our knowledge, this is the first use of such concepts in a bisimulation definition. It seems possible that the ideas here are applicable to other forms of bisimulation under stuttering, such as weak bisimulation [Milner 90], and branching bisimulation [GW 89]. We have chosen to focus on stuttering bisimulation because of its close connection to CTL^*.

The chapter is structured as follows: Section 7.2 contains the definition of stuttering bisimulation from [BCG 88], and the definition of well-founded bisimulation. The equivalence of the two formulations is shown in Section 7.3. Applications of the well-founded bisimulation proof rule to the alternating bit protocol and token-ring protocols are presented in Section 7.4, together with a new quotient construction for stuttering bisimulation equivalences. The chapter concludes with a discussion of related work and future directions.

7.2 Preliminaries

We define bisimulations over Kripke structures instead of LTS's, Because of our interest in preservation of $\mathsf{CTL}^*\backslash\mathsf{X}$ properties. The results are valid for LTS's as well, under the constraint that stuttering is modeled with τ actions. Kripke Structures (KS) are represented by the tuple $(S, \rightarrow, \lambda, I, AP)$, where S is a set of *states*, $\rightarrow \subseteq S \times S$ is the *transition relation*, AP is the set of *atomic propositions*, $\lambda : S \rightarrow \mathcal{P}(AP)$ is the *labelling function*, that maps each state to the subset of atomic propositions that hold at the state, and I is the set of *initial states*. We write $s \rightarrow t$ instead of

$(s,t) \in \rightarrow$. We only consider Kripke Structures with denumerable branching, i.e., where for every state s, $|\{t \mid s \rightarrow t\}|$ is at most ω.

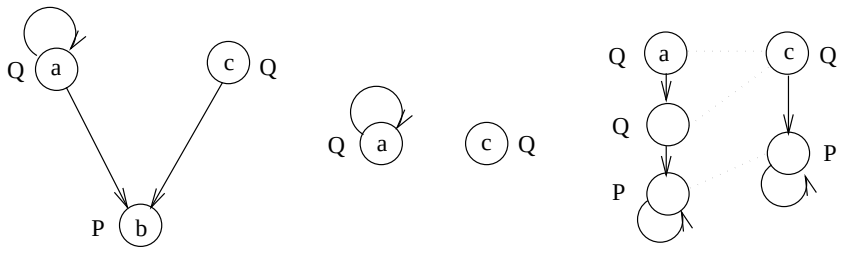
Stuttering Bisimulation is defined in Chapter 2; however, we repeat the definition here for completeness.

Definition 7.1 (Stuttering Bisimulation (cf. [BCG 88])[1]) *Let $\mathcal{A}$ be a Kripke Structure of the form $(S, \rightarrow, \lambda, I, AP)$. A relation $B \subseteq S \times S$ is a stuttering bisimulation on $\mathcal{A}$ iff B is symmetric, and for every s, t such that $(s,t) \in B$,*

1. $\lambda(s) = \lambda(t)$,
2. $(\forall \sigma : fp(s,\sigma) : (\exists \delta : fp(t,\delta) : match_B(\sigma,\delta)))$.

where $fp(s,\sigma)$ is true iff σ is a path starting at s, which is either infinite, or its last state has no successors w.r.t. $\rightarrow$. $match_B(\sigma,\delta)$ is true iff σ and δ can be divided into an equal number of non-empty, finite, segments such that any pair of states from segments with the same index is in the relation B. The formal definition of *match* is given in Chapter 2. States s and t are *stuttering bisimilar* iff there is a stuttering bisimulation relation B for which $(s,t) \in B$.

Examples:



Structure L Structure M Structure N

States a and c are not stuttering bisimilar in structures L and M, but they are in structure N. Indeed, $L, c \models \mathsf{AF}\ P$, but $L, a \not\models \mathsf{AF}\ P$. Structure M shows

that stuttering bisimulation distinguishes between deadlock (state c) and divergence (state a) : $M, c \not\models \mathsf{EX}\ true$, but $M, a \models \mathsf{EX}\ true$ [2]. The dotted lines show a stuttering bisimulation on structure N.

□

Our alternative formulation is based on a simple idea from program semantics: we define a mapping from states to a well-founded set, and require, roughly, that the mapping decrease with each stuttering step. Thus, each stuttering segment is forced to be of finite length, which makes it possible to construct matching fullpaths from related states.

Definition 7.2 (Well-Founded Bisimulation)) *Let $\mathcal{A} = (S, \rightarrow, \lambda, I, AP)$ be a KS. Let $rank : S \times S \times S \rightarrow W$ be a total function, where $(W, \prec)$ is well-founded*[3]*. A relation $B \subseteq S \times S$ is a well-founded bisimulation on $\mathcal{A}$ w.r.t. rank iff B is symmetric, and*

For every s, t such that $(s, t) \in B$,

1. $\lambda(s) = \lambda(t)$

2. $(\forall u : s \rightarrow u :$

$$(\exists v : t \rightarrow v :\ (u, v) \in B) \vee \qquad (a)$$

$$((u, t) \in B \wedge rank(u, u, t) \prec rank(s, s, t)) \vee \qquad (b)$$

$$((u, t) \notin B \wedge (\exists v : t \rightarrow v :\ (s, v) \in B \wedge rank(u, s, v) \prec rank(u, s, t)))) \qquad (c)$$

Notice that if W is a singleton, then clauses (b) and (c) are not applicable, so B is a strong bisimulation.

The intuition behind this definition is that when $(s, t) \in B$ and $s \rightarrow u$, either there is a matching transition from t (clause (2a)), or $(u, t) \in B$ (clause (2b)) - in

[2] The [dNV 90] formulation of stuttering bisimulation considers states a and c of N to be bisimilar. The difference between our formulations is only in the treatment of deadlock vs. divergence in non-total structures.

[3] *$(W, \prec)$ is well-founded iff there is no infinite subset $\{a.i \mid i \in \mathbf{N}\}$ of W that is a strictly decreasing chain, i.e. where for all $i \in \mathbf{N}$, $a_{i+1} \prec a_i$.*

which case the rank decreases, allowing (2b) to be applied only a finite number of times - or $(u,t) \notin B$, in which case (by clause (2c)), there must be a successor v of t such that $(s,v) \in B$. As the rank decreases at each application of (2c), clause (2c) can be applied only a finite number of times. Hence, eventually, a state related to u by B is reached. Theorem 7.1 (soundness) is proved along these lines.

7.3 Equivalence of the two formulations

The equivalence of the two formulations is laid out in the following theorems.

Theorem 7.1 (Soundness) *Any well-founded bisimulation on a KS is a stuttering bisimulation.*

Proof.

Let B be a well-founded bisimulation on a KS $\mathcal{A}$, w.r.t. a function *rank* and a well-founded structure $(W, \prec)$.

Let (s,t) be an arbitrary pair in B. Then, $\lambda(s) = \lambda(t)$, by clause (1) of the well-founded bisimulation definition. We show that if σ is a fullpath starting at s, then there is a fullpath δ starting at t such that $match_B(\sigma, \delta)$ holds. In the following, we use the symbol ';' for concatenation of finite paths, and $\circ$ for concatenation with removal of duplicate state. For example, $aa; ab = aaab$, and $aa \circ ab = aab$.

We construct δ inductively. For the base case, $\delta_0 = t$. Inductively assume that after i steps, $i \geq 0$, δ has been constructed to the point where it matches a prefix γ of σ such that the end states of γ and δ mark the beginning of the ith segments. Let u be the last state of γ and v be the last state of δ. By the inductive hypothesis, $(u,v) \in B$.

If σ ends at u, then u has no successor states. Let ξ be any fullpath starting at v. Since u has no successors, a simple induction using (2b) shows that for every state x in ξ, (x,u) is in B. Each application of (2b) strictly decreases *rank* along

ξ, hence ξ must be finite. The fullpath $\delta \circ \xi$ is a finite fullpath matching the finite fullpath σ.

If σ does not end at u, let w be the successor of u in σ. As $(u, v) \in B$,

(i) If (2a) holds, there is a successor x of v such that $(w, x) \in B$. Let w and x mark the beginning of a new segment. Extend δ to $\delta; x$, which matches $\gamma; w$. The induction step is proved. Otherwise,

(ii) If (2a) does not hold, but (2b) does, then $(w, v) \in B$. Let ρ be the longest prefix of the suffix of σ starting at u such that for every state a in ρ, $(a, v) \in B$, and only (2b) holds for (a, v) w.r.t. $a \rightarrow b$ for every successive pair of states $a; b$ in ρ. ρ has at least one pair, as $u; w$ is a prefix of ρ.

ρ cannot be infinite, as by (2b), for each successive pair $a; b$ in ρ, $\mathit{rank}(b, b, v) \prec \mathit{rank}(a, a, v)$, so the rank decreases strictly in the well-founded set. Let y be the last state of ρ. If σ terminates at y, the argument given earlier applies. Otherwise, y has a successor y' in σ, but as ρ is maximal, either (2a) or (2c) must apply for $(y, v) \in B$ w.r.t. $y \rightarrow y'$. (2c) cannot apply, as then there is a successor x of v such that $(y, x) \in B$, which contradicts the properties of ρ.

Hence (2a) must apply. Let x be the successor of v such that $(y', x) \in B$. Let y' and x mark the beginning of a new segment, and extend δ to $\delta; x$, which matches $(\gamma \circ \rho); y'$.

(iii) If (2c) is the only clause that holds of (u, v) w.r.t. $u \rightarrow w$, let π be a finite path maximal w.r.t. prefix ordering such that π starts at v, and for every successive pair of states $a; b$ in π, $(u, a) \in B$, only (2c) is applicable w.r.t. $u \rightarrow w$, and b is the successor of a given by the application of (2c).

Such a maximal finite path exists as, otherwise, there is an infinite path ξ satisfying the conditions above. By (2c), for successive states $a; b$ in ξ, $\mathit{rank}(w, u, b) \prec \mathit{rank}(w, u, a)$; so there is an infinite strictly decreasing chain in $(W, \prec)$, which contradicts the well-foundedness of $(W, \prec)$. Let x be the last state in π. Then $(u, x) \in B$,

and as π is maximal, either (2a) or (2b) holds of (u, x) w.r.t. $u \rightarrow w$. So $x \neq v$. (2b) cannot hold, as then (w, x) is in B; but then (2a) would hold for the predecessor of x in π.

Hence (2a) holds; so x has a successor z for which $(w, z) \in B$. Let w and z mark the beginning of a new segment, and extend δ to $(\delta \circ \pi); z$, which matches $\gamma; w$.

The induction step is shown in either case.

The inductive argument shows that successively longer prefixes of σ have successively longer matching finite paths, which are totally ordered by prefix order. Hence, if σ is infinite, the limit of these matching paths is an infinite path from t which matches σ using the partitioning into finite non-empty segments constructed in the proof. □

It is also desirable to have completeness : that for every stuttering bisimulation, there is a rank function over a well-founded set which gives rise to a well-founded bisimulation.

Theorem 7.2 (Completeness) *For any stuttering bisimulation B on a KS $\mathcal{A}$, there is a well-founded structure $(W, \prec)$ and corresponding function rank such that B is a well-founded bisimulation on $\mathcal{A}$ w.r.t. rank.*

□

Let $\mathcal{A} = (S, \rightarrow, \lambda, I, AP)$. The well-founded set W is defined as the product $W_0 \times W_1$ of two well-founded sets, with the new ordering being lexicographic order. The definitions of the well-founded sets W_0 and W_1, and associated functions $rank_0$ and $rank_1$ are given below. Informally, $rank_0(a, b)$ measures the height of a finite-depth computation tree rooted at a, whose states are related to b but not to any successor of b. $rank_1(a, b, c)$ measures the shortest finite path from c that matches b and ends in a state related to the successor a of b.

Definition of $(W_0, \prec_0)$ and $rank_0$

For a pair (s,t) of states of $\mathcal{A}$, construct a tree, $tree(s,t)$, by the following (possibly non-effective) procedure, which is based on clause (2b) of the definition of well-founded bisimulation:

1. The tree is empty if the pair (s,t) is not in B. Otherwise,

2. s is the root of the tree. The following invariant holds of the construction: For any node y of the current tree, $(y,t) \in B$, and if y is not a leaf node, then for every child z of y in the tree, z is a successor of y in $\mathcal{A}$, and there is no successor v of t in $\mathcal{A}$ such that $(z,v) \in B$.

3. For a leaf node y, and any successor z of y in $\mathcal{A}$, if $(z,t) \in B$, but there is no successor v of t in $\mathcal{A}$ such that $(z,v) \in B$, then add z as a child of y in the tree. If no such successor exists for y, then terminate the branch at y.

   Repeat step 3 for every leaf node on an unterminated branch.

Lemma 7.1 *$tree(s,t)$ is well-founded.*

Proof.

Suppose to the contrary that there is an infinite branch σ, which is therefore a fullpath, starting at s. Let u be the successor of s on σ, and let σ' be the fullpath that is the suffix of σ starting at u.

By construction of the tree, for every state x on σ', $(x,t) \in B$, and for every successor v of t, $(x,v) \notin B$. However, as $(u,t) \in B$, there must be a fullpath δ starting at t for which $match_B(\sigma', \delta)$ holds. Let w be the successor of t on δ. From the definition of $match$, for some x on σ', $(x,w) \in B$. This is a contradiction. Hence, every branch of the tree must be of finite length. □

Since $tree(s,t)$ is well-founded, it can be assigned an ordinal height using a standard bottom-up assignment technique for well-founded trees : assign the empty tree height 0, and any non-empty tree T the ordinal $sup\ \{height(S)+1 \mid S \lhd T\}$,

where $S \lhd T$ holds iff S is a strict subtree of T. Let $rank_0(s,t)$ equal the height of $tree(s,t)$. As trees with countable branching need only countable ordinals as heights, let W_0 be the set of countable ordinals, ordered by the inclusion order $\in$.

Lemma 7.2 *If $tree(s,t)$ is non-empty, and u is a child of s in the tree, then $rank_0(u,t) \prec_0 rank_0(s,t)$.*

Proof.

From the construction, $tree(u,t)$ is the subtree of $tree(s,t)$ rooted at node u; hence its height is strictly smaller. □

Definition of $(W_1, \prec_1)$ and $rank_1$

Let $W_1 = \mathbf{N}$, the set of natural numbers, and let $\prec_1$ be the usual order $<$ on $\mathbf{N}$. The definition of $rank_1$ is as follows : For a tuple (u,s,t) of states of $\mathcal{A}$,

1. If $(s,t) \in B$, $s \rightarrow u$, $(u,t) \notin B$, and for every successor v of t, $(u,v) \notin B$, then $rank_1(u,s,t)$ is the length of the shortest initial segment that matches s among all matching fullpaths $s;\sigma$ and δ, where σ starts at u, and δ starts at t. Precisely,

$$rank_1(u,s,t) = (min\ \delta, \xi, \sigma, \pi : fp(t,\delta) \wedge fp(u,\sigma) \wedge \pi, \xi \in INC \wedge corr((s;\sigma,\pi),(\delta,\xi)) : |seg_0(\delta,\xi)|)$$

   As $(s,t) \in B$, and $s \rightarrow u$, there exist matching fullpaths $s;\sigma$ and δ, with σ starting at u and δ starting at t. As $(u,t) \notin B$, and no successor of t matches u, under any partition ξ of any fullpath δ that matches a fullpath $s;\sigma$, the initial segment, $seg_0(\delta,\xi)$, matches s, and must contain at least two states: t and some successor of t. Thus, $rank_1(u,s,t)$ is defined, and is at least 2.

2. Otherwise, $rank_1(u,s,t) = 0$.

□

Theorem 7.2 (Completeness) *For any stuttering bisimulation B on KS $\mathcal{A}$, there is a well-founded set $(W, \prec)$ and corresponding function rank such that B is a well-founded bisimulation on $\mathcal{A}$ w.r.t. rank.*

Proof.

Let $W = W_0 \times W_1$. The ordering $\prec$ on W is the lexicographic ordering on $W_0 \times W_1$, i.e., $(a, b) \prec (c, d) \equiv (a \prec_0 c) \vee (a = c \wedge b \prec_1 d)$. Define $rank(u, s, t) = (rank_0(u, t), rank_1(u, s, t))$. W is well-founded, and *rank* is a total function. We have to show that B is a well-founded bisimulation w.r.t. *rank*. Let $(s, t) \in B$.

1. $\lambda(s) = \lambda(t)$, from the definition of stuttering bisimulation.

2. Let u be any successor of s. If there is no successor v of t such that $(u, v) \in B$, consider the following cases:

   - $(u, t) \in B$: As no successor of t is related to u by B, u is a child of s in $tree(s, t)$, and by Lemma 7.2, $rank_0(u, t) \prec_0 rank_0(s, t)$. Hence, $rank(u, u, t) \prec rank(s, s, t)$.

   - $(u, t) \notin B$: As no successor of t is related to u by B, $rank_1(u, s, t)$ is non-zero. Let fullpath δ starting at t and partition ξ "witness" the value of $rank_1(u, s, t)$. Let v be the successor of t in the initial segment $seg.0(\delta, \xi)$. This successor exists, as the length of the segment is at least 2. $rank_1(u, s, v)$ is at most $rank_1(u, s, t) - 1$, so $rank_1(u, s, v) \prec_1 rank_1(u, s, t)$.

     As no successor of t is related by B to u, $(u, v) \notin B$, so $rank_0(u, v) = 0$. As $(u, t) \notin B$, $rank_0(u, t) = 0$. Since *rank* is defined by lexicographic ordering, $rank(u, s, v) \prec rank(u, s, t)$.

   Hence, one of (2a),(2b) or (2c) holds for $(s, t) \in B$ w.r.t. $s \rightarrow u$.

□

For a Kripke Structure that is *finite-branching* (every state has finitely many successor states), $tree(s,t)$ for any s,t is a finite, finitely-branching tree; so its height is a natural number. Hence, $W_0 = \mathbf{N}$.

Proposition 7.1 *For a finite-branching Kripke Structure,* $W = \mathbf{N} \times \mathbf{N}$.

□

Theorem 7.3 (Main) *Let* $\mathcal{A}$ *be a Kripke Structure. A relation* B *on* $\mathcal{A}$ *is a stuttering bisimulation iff* B *is a well-founded bisimulation w.r.t. some rank function.*

Proof.

The claim follows immediately from Theorems 7.1 and 7.2. □

For simplicity, the definitions are structured so that a bisimulation is a symmetric relation. The main theorem holds for bisimulations that are not symmetric, but the definition of rank has to be modified slightly, to take the direction of matching (by B or by B^{-1}) into account.

7.4 Applications

The definition of a well-founded bisimulation is, by Theorem 7.3, in itself a simple proof rule for determining if a relation is indeed a bisimulation up to stuttering. In this section, we look at several applications of this proof rule. We outline the proofs of well-founded bisimulation for the alternating bit protocol from [Milner 90], and a class of token-ring protocols studied in [EN 95]. We also present a new quotient construction for a well-founded bisimulation that is an equivalence. In all of these applications, the construction of the appropriate well-founded set and ranking function is quite straightforward. We believe that this is the case in other applications of stuttering bisimulation as well.

7.4.1 The Alternating Bit Protocol

A version of the alternating bit protocol is given in [Milner 90], which we follow closely. The protocol has four entities : *Sender* and *Replier* processes, and message (*Trans*) and acknowledgement (*Ack*) channels. Messages and acknowledgements are tagged with bits 0 and 1 alternately. For simplicity, message contents are ignored; both channels are sequences of bits. For a sequence of values σ, let $order(\sigma)$ represent the sequence resulting from removing duplicates from c, and let $count(\sigma)$ be a vector of the numbers of duplicate bits. Vectors are compared component-wise if they have the same length. For example, $order(0^3; 1^2) = 0; 1$, $count(0^3; 1^2) = (3, 2)$, and $count(1^5) = (5)$.

The proposed WF bisimulation B relates states s and t iff

1. The local states of the *sender* and *replier* processes are identical in s and t, and

2. For the channel *Trans* from sender to replier, $order(Trans(s); rmsg(s)) = order(Trans(t); rmsg(t))$, where $rmsg(u)$ is the message stored at the replier process in state u, and

3. For the channel *Ack* from replier to sender, $order(Ack(s); \neg sflag(s)) = order(Ack(t); \neg sflag(t))$, where $sflag(u)$ is the flag used by the sender to tag the next outgoing message.

Note that the number of duplicate messages is abstracted away.

Let $\alpha(s) = (count(Trans(s); rmsg(s)), count(Ack(s); \neg sflag(s)))$, and define $rank(u, s, t)$ as $(\alpha(s), \alpha(t))$. The operations of the protocol are sending a bit or receiving a bit on either channel, and duplicating or deleting a bit on either channel, along with a *skip* action. For the sending, receiving and deleting actions, it is straightforward to verify that B is a WF bisimulation w.r.t. *rank*. The rank function is used, for instance, at a receive action in s with $msg(s) = b$ and $Trans(s) = a^1$,

while the same channel in the corresponding state t has contents $a^m; b^n$ $(n > 1)$. The receive action at s results in a state u with channel content $\langle\rangle$ and $rmsg(u) = a$, while the same action at t results in a state v with channel content $a^m; b^{n-1}$ and $rmsg(v) = b$. So u and v are unrelated but v is related to s, and $rank(u, s, v) < rank(u, s, t)$ (cf. clause (2c)).

The duplication action at a state s may not have a corresponding duplication action at a related state t if the message being duplicated is not present in the channel at t (although it must then have been received, from the definition of B). For example, s has $rmsg(s) = b$ and $Trans(s) = b^n$ $(n \geq 1)$, while t has $rmsg(s) = b$ and $Trans(s) = \langle\rangle$. However, the *skip* action can be executed from t, which matches the state after the duplication.

The example exhibits unbounded stuttering. With the original formulations of stuttering bisimulation, one would have to construct a computation of length n from state t to match the receive action from state s. This is typically done by a recursive definition of the matching computation; so the proof of matching is done by an induction on n which introduces a number of proof obligations, and complicates the proof. In contrast, with the new formulation, one need consider only a single transition from t.

Although the bisimulation B is an equivalence, it has an infinite number of equivalence classes. For the protocol, however, the initial state has empty channels and it is possible to show that, although the reachable state space is still infinite, the reachable states of the protocol have channels with *order* values of length at most 2. Thus, B induces a *finite* partition of the reachable state space. This fact can be exploited to model-check the properties of the protocol, as described subsequently in Section 7.4.3.

7.4.2 Simple Token-Ring Protocols

In [EN 95] (cf. [BCG 89]), stuttering bisimulation is used to show that for token-rings of similar processes, a small *cutoff* size ring is equivalent to one of any larger size. [EN 95] shows that the computation trees of process 0 in rings of size 2 and of size n, $n \geq 2$, are stuttering bisimilar. It follows that a property over process 0 is true of *all* sizes of rings iff it is true of the ring of size 2. From symmetry arguments (cf. [ES 93, CFJ 93]), a property holds of all processes iff it holds for process 0. This result and its extensions are presented in Chapter 3.

The proof given in that paper uses the [BCG 88] definition and is quite lengthy; we indicate here how to use well-founded bisimulation. Although the proof can be simplified in the manner indicated in Chapter 3, that requires introducing a specialized form of bisimulation under stuttering. Here, we use well-founded bisimulation, which, as shown earlier, is equivalent to the earlier definition of stuttering bisimulation.

Each process in the system alternates between blocking receive and send token transfer actions, with a finite number of local steps in between. For an n-process system with state space S_n, define $\alpha_n : S_n \rightarrow \mathbf{N}^2$ as the function given by $\alpha_n(s) = (i, j)$ where, in state s, if process m has the token, then $i = (n - m)\ mod\ n$ is the distance of of the token from process 0, and j is the sum over processes of the maximum number of steps of each process from its local state to the first token transfer action. The tuples are ordered lexicographically. Let the rank function be $rank(u, s, t) = (\alpha_m(s), \alpha_n(t))$, where s and t are states in instances with m and n processes respectively. Let the relation B be defined by $(s, t) \in B$ iff the local state of process 0 is identical in s and t.

It is straightforward to verify that B is a well-founded bisimulation w.r.t. $rank$. The rank function is used in the situation where the token is received by process 0 by a move from state s to state u; however, the reception action is not

enabled for process 0 in a state t related to s by B. In this case, some move of a process other than 0 is enabled at t, and results in a state v that reduces α_n, and hence the rank, either by a transfer of the token to the next process, or by reducing the number of steps to the first token transfer action. The next state v is related to s by B (cf. clause (2c) of the definition).

7.4.3 Quotient Structures

For a bisimulation B on KS $\mathcal{A}$ that is an equivalence relation, a *quotient structure* $\mathcal{A}/B$ (read as $\mathcal{A}$ "mod" B) can be defined, where the states are equivalence classes (w.r.t. B) of states of $\mathcal{A}$, and the new transition relation is derived from the transition relation of $\mathcal{A}$. Quotient structures are usually much smaller than the original; a bisimulation with finitely many classes induces a finite quotient, as is the case in the examples given in the previous sections.

Let $\mathcal{A} = (S, \rightarrow, \lambda, I, AP)$ be a KS, and B be a well-founded bisimulation on $\mathcal{A}$ w.r.t. a rank function α, that is an equivalence relation on S. The equivalence class of a state s is denoted by $[s]$. Define $\mathcal{A}/B$ as the KS $(\mathcal{S}, \rightsquigarrow, \Lambda, \mathcal{I}, AP)$ where:

- $\mathcal{S} = \{[s] \mid s \in S\}$
- The transition relation is given by : For $C, D \in \mathcal{S}$, $C \rightsquigarrow D$ iff either
  1. $C \neq D$, and $(\exists s, t : s \in C \wedge t \in D : s \rightarrow t)$, or
  2. $C = D$, and $(\forall s : s \in C : (\exists t : t \in C : s \rightarrow t))$.

  The distinction between the two cases is made in order to prevent spurious self-loops in the quotient, arising from stuttering steps in the original.
- The labelling function is given by $\Lambda(C) = \lambda(s)$, for some s in C. (states in an equivalence class have the same label)
- The set of initial states, $\mathcal{I}$, equals $\{[s] \mid s \in I\}$.

Theorem 7.4 *$\mathcal{A}$ is stuttering bisimilar to $\mathcal{A}/B$.*

Proof.

Form the disjoint union of the KS's $\mathcal{A}$ and $\mathcal{A}/B$. The bisimulation on this structure relates states of $\mathcal{A}$ and $\mathcal{A}/B$ as follows : $(a, b) \in R$ iff $[a] = b \vee [b] = a$.

Let $sw : \mathcal{S} \to S$ (read "state witness") be a partial function, defined at C only when $C \rightsquigarrow C$ does not hold. When defined, $v = sw(C)$ is such that $v \in C$, but no successor of v w.r.t. $\to$ is in C. Such a v exists by the definition of $\rightsquigarrow$. Let $ew : \mathcal{S}^2 \to S^2$ (read "edge witness") be a partial function, defined at (D, C) iff $C \rightsquigarrow D$. When defined, $(v, u) = ew(D, C)$ is such that $u \in C, v \in D$, and $u \to v$.

Let *rank* be a function defined on $W \cup \{\perp\}$ ($\perp$ is a new element unrelated to any elements of W) by : If $u, s \in S$, and $sw(C)$ is defined, then $rank(u, s, C) = \alpha(u, s, sw(C))$. If $D, C \in \mathcal{S}$ and $s \in S$, then $rank(D, C, s) = \alpha(ew(D, C), s)$, if $ew(D, C)$ is defined. Otherwise, $rank(a, b, c) = \perp$.

Let $(a, b) \in R$. From the definition of R, a and b have the same label.

- $a \in S$: For clarity, we rename (a, b) to (s, C). By the definition of R, $C = [s]$. Let $s \to u$. If $[s] \rightsquigarrow [u]$, then there is a successor $D = [u]$ of C such that $(u, D) \in R$, and clause (2a) holds.

  If the edge from $[s]$ to $[u]$ is absent, then $[s]$ must equal $[u]$, and $sw(C)$ is defined. Let $x = sw(C)$. As $(s, x) \in B$, and $(u, x) \in B$, but x has no successors to match u, clause (2b) holds for B, i.e., $\alpha(u, u, x) \prec \alpha(s, s, x)$. By definition of *rank*, $rank(u, u, C) \prec rank(s, s, C)$, so (2b) holds for R.

- $a \in \mathcal{S}$: For clarity, we rename (a, b) to (C, s). Let $C \rightsquigarrow D$. Let $(y, x) = ew(D, C)$. As $x \to y$, and $(x, s) \in B$, there are three cases to consider :

  1. There is a successor u of s such that $(y, u) \in B$. Then $[y] = [u]$, so $(D, u) \in R$, and (2a) holds.

2. $(y, s) \in B$. Then $[y] = [x]$, so $C = D$. As $C \rightsquigarrow D$, and $s \in C$, s has a successor u such that $u \in C$; hence (D, u) is in R and (2a) holds.

3. $(y, s) \notin B$ and there exists u such that $s \rightarrow u$, $(x, u) \in B$, and $\alpha(y, x, u) \prec \alpha(y, x, s)$. Hence, $(C, u) \in R$, and $rank(D, C, u) \prec rank(D, C, s)$. So clause (2c) holds.

□

7.5 Related Work and Conclusions

Other formulations of bisimulation under stuttering have been proposed; however, they too involve reasoning about finite, but unbounded sequences of transitions. Examples include branching bisimulation [GW 89], divergence sensitive stuttering [dNV 90], and weak bisimulation [Milner 90]. We believe that it is possible to characterize branching bisimulation in a manner similar to our characterization of stuttering bisimulation, given the close connection between the two that is pointed out in [dNV 90]. An interesting question is whether a similar characterization can be shown for weak bisimulation [Milner 90].

Many proof rules for temporal properties are based on well-foundedness arguments, especially those for termination of programs under fairness constraints (cf. [GFMdR 83, Francez 86, AO 92]). Vardi [Vardi 87], and Klarlund and Kozen [KK 91] develop such proof rules for very general types of linear temporal properties. Our use of well-foundedness arguments for defining a bisimulation appears to be new, and, we believe, of intrinsic mathematical interest. The motivation in each of these instances is the same : to replace reasoning about unbounded or infinite paths with reasoning about single transitions.

Earlier definitions of stuttering bisimulation are difficult to apply to large problems essentially because of the difficulty of reasoning about unbounded stutter-

ing paths. Our new characterization, which replaces such reasoning with reasoning about single steps, makes proofs of equivalence under stuttering easier to demonstrate and understand. In the example applications, it was quite straightforward to determine an appropriate well-founded set and rank function. Indeed, rank functions are implicit in proofs that use the earlier formulations. As the examples demonstrate, using rank functions explicitly leads to proofs that are shorter and which can be carried out with the assistance of a mechanical theorem prover.

Chapter 8

Conclusions and Future Work

8.1 Summary

This dissertation has as its goal the identification of methods to ameliorate the effect of state explosion in automated verification procedures. Common types of systems that exhibit state explosion include those that are parameterized by the number of component processes and those with a large data domain and a relatively small control component which is largely independent of the data. This dissertation is focussed on methods of verifying such types systems with a view to reducing state explosion. The contributions of this dissertation are summarized below.

For two types of parameterized systems, token-rings with a synchronizing token and synchronous control-user systems, we show that the model-checking problem is decidable for interesting types of indexed specification formulas. We also give tight bounds on the complexity of these algorithms, and delineate the border between decidability and undecidability of the verification task. In both cases, the decidability follows from abstractions that establishes an exact correspondence between every member of the infinite family of instances of the parameterized system and a finite abstract graph. These algorithms have been applied to the verification

of an industrial standard bus protocol, the SAE-J1850 protocol.

Reviewing this earlier work, it becomes apparent that there are many similarities between the abstractions used here and those applied elsewhere for the automated verification of other types of infinite-state systems, such as Petri Nets. A contribution of this dissertation is the development of a general framework for model-checking infinite-state systems, which exposes these similarities. This framework has been utilized in developing a semi-algorithmic procedure for model-checking parameterized broadcast protocols. This procedure terminates in the case of a simple MESI invalidation-based cache consistency protocol and produces a finite graph, over which safety properties of the protocol are determined to hold for the entire family of instances.

Systems with a large data domain, and a control component that is largely independent of the data are quite commonplace. Examples include FIFO buffers, instruction pipelines, and many cache coherency protocols. A common technique for verifying such *data-insensitive* systems is to develop abstractions that partition the data domain into large equivalence classes, such that the control component has the same behavior for equivalent data items. To correctly preserve control properties between the original and abstracted system, however, the abstraction has to be shown to be a *bisimulation*. Often, the bisimulation is insensitive to the stuttering (finite repetition) of state propositions. A contribution of this dissertation is a simple, *local* proof method to show bisimulation under stuttering, which is proved to be equivalent to known global proof methods. This method is used to prove a bisimulation that reduces an infinite-state alternating bit protocol to a finite state abstraction that preserves relevant properties of the protocol.

8.2 Future Work

8.2.1 Verification of Parameterized Systems

The work presented in this dissertation provides some preliminary answers to the difficult question of devising algorithms for the verification of parameterized systems. The systems presented here are all characterized syntactically. A topic for future research is to develop algorithms for systems that are characterized semantically. For such an algorithm to be usable, the conditions characterizing a class must be kept simple, as they will have to be be checked by a hand proof or with the assistance of a mechanical theorem prover. Invariants or other safety properties on the communication patterns between processes may be the appropriate type of condition to utilize.

8.2.2 Data Abstraction

A general technique for data abstraction is to show that a proposed abstraction is a bisimulation on the state space. Applying this in practice, however, is a difficult task. Some of the complexity comes from global definitions of bisimulation as opposed to local ones; an issue that is addressed in this work. Another source of difficulty is the question of coming up with a candidate relation. Some of this difficulty may be alleviated by identifying a class of programs that can be suitably annotated so that an automatic procedure can develop candidate abstractions. For instance, an annotation that separates data components from control, and identifies the mathematical operations performed on the data would be very useful. Automated theorem provers such as PVS [HS 96] or Nqthm [BM 79] may be used to aid the proof of correctness of proposed abstractions. This is one direction in which the differing strengths of the Model Checking and automated theorem proving approaches may be combined to good effect.

8.2.3 Compositionality

Given that the problem of model-checking using succinct representations is PSPACE-hard [GW 83], it is unlikely that these algorithms can handle the large state spaces of most programs. Thus compositional methods for verification are of prime importance. Many of the compositional methods that have been proposed use assume-guarantee rules, which are often quite complex in order to prevent circular reasoning. It is thus desirable to search for simple compositional methods; perhaps, those that use the communication structure of the composition to simplify proof obligations.

Dijkstra [Dijkstra 76] demonstrates effectively the manner in which rules for verification may be turned into heuristics for guiding the *design* of programs that are correct by construction. In a similar vein, the restrictions on systems that will be needed in order to make the above approaches work effectively may in turn aid the design of such systems.

Bibliography

[AD 91] Alur, R., Dill, D. Automata for modeling Real-Time Systems, ICALP 1991.

[ACD 90] Alur, R., Courcoubetis, C., Dill, D. Model-Checking for Real-Time Systems, LICS 1990.

[AHV 93] Alur, R., Henzinger, T., Vardi, M. Parametric Real-Time Reasoning, STOC 1993.

[AJ 93] Abdulla, P., Jonsson, B. Verifying programs with unreliable channels. LICS, 1993. (full version in *Information and Computation*, 127(2):91-101, 1996)

[ACJT 96] Abdulla, P., Cerans, K., Jonsson B., Tsay Y-K. General Decidability Theorems for Infinite-State Systems, LICS, 1996.

[AK 86] Apt, K., Kozen, D. Limits for automatic verification of finite-state concurrent systems. IPL 15.

[AO 92] Apt, K., Olderog, E. R. Verification of Sequential and Concurrent Programs, Springer Verlag, 1992.

[Browne 89] Browne, M. Automatic Verification of Finite State Machines using Temporal Logic. PhD thesis, Carnegie-Mellon Univ., CMU-CS-89-117.

[Bradfield 92] Bradfield, J. *Verifying Temporal Properties of Systems*, in Progress in Theoretical Computer Science, Birkhäuser, 1992.

[BCG 88] Browne, M. C., Clarke, E. M., Grumberg, O. Characterizing Finite Kripke Structures in Propositional Temporal Logic, Theor. Comp. Sci., vol. 59, 1988.

[BCG 89] Browne, M. C., Clarke, E. M., Grumberg, O. Reasoning about Networks with Many Identical Finite State Processes, Information and Computation, vol. 81, no. 1, April 1989.

[BCMDH 90] Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J. Symbolic model checking: 10^20 states and beyond, LICS 1990.

[BG 96] Boigelot, B., Godefroid, P., Symbolic verification of communication protocols with infinite state spaces using QDD's, in CAV 1996.

[BGWW 97] Boigelot, B., Godefroid, P., Willems, B., Wolper, P. The power of QDD's, Proceedings of the Fourth International Static Analysis Symposium, Paris, September 1997. LNCS, Springer-Verlag.

[BM 79] Boyer, R., Moore, J.S. *A Computational Logic.* Academic Press, New York, 1979.

[BS 90] Bradfield, J., Stirling, C. Local Model Checking for Infinite State Spaces, *Theoretical Computer Science*, 1990.

[Cleaveland 93] Cleaveland, R. Analyzing Concurrent Systems using the Concurrency Workbench. in *Functional Programming, Concurrency, Simulation, and Automated Reasoning*, Springer-Verlag LNCS 693.

[CE 81] Clarke, E. M., Emerson, E. A. Design and Synthesis of Synchronization

Skeletons using Branching Time Temporal Logic, in *Workshop on Logics of Programs*, Springer-Verlag LNCS 131.

[CES 86] Clarke, E. M., Emerson, E. A., and Sistla, A. P., Automatic Verification of Finite-State Concurrent Systems using Temporal Logic, *ACM Trans. Prog Lang and Sys.*, vol. 8, no. 2, April 1986.

[CFJ 93] Clarke, E. M., Filkorn, T., Jha, S. Exploiting Symmetry in Temporal Logic Model Checking, 5th CAV, Springer-Verlag LNCS 697.

[CG 87] Clarke, E. M., Grumberg, O. Avoiding the State Explosion Problem in Temporal Logic Model Checking Algorithms, PODC 1987.

[CGJ 95] Clarke, E.M., Grumberg, O., Jha, S. Verifying Parameterized Networks using Abstraction and Regular Languages. CONCUR 95.

[CH 93] Cleaveland, R., Hennessy, M. Testing Equivalence as a Bisimulation Equivalence. Formal Aspects of Computing, vol. 5, 1993.

[CPS 89] Cleaveland, R., Parrow, J., Steffen, B. The Concurrency Workbench. In J.Sifakis (ed), *Automatic Verification Methods for Finite State Systems*, Springer-Verlag, LNCS 407.

[Dijkstra 76] Dijkstra, E.W. *A Discipline of Programming*. Prentice-Hall, 1976.

[Dijkstra 85] Dijkstra, E. W. Invariance and non-determinacy, in *Mathematical Logic and Programming Languages*, Prentice-Hall International Series in Computer Science. eds. C.A.R Hoare and J.C Shepherdson.

[DS 90] Dijkstra, E. W., Scholten. C. S. *Predicate Calculus and Program Semantics*, Springer-Verlag, 1990.

[dNV 90] De Nicola, R., Vaandrager, F. Three logics for Branching Bisimulation, 5th Annual IEEE Symp. on Logic in Computer Science, 1990.

[Emerson 90] Emerson, E. A. Temporal and Modal Logic, in Handbook of Theoretical Computer Science, (J. van Leeuwen, ed.), Elsevier/North-Holland, 1991.

[Esparza 94] Esparza, J. On the decidability of model checking for several μ-calculi and Petri nets, in CAAP 94, LNCS 787.

[EH 82] Emerson, E.A., Halpern, J. Y. Decision procedures and Expressiveness in the Temporal Logic of Branching Time, *Proc. 14th ACM STOC*, San Francisco, 1982.

[EL 86] Emerson, E.A., Lei, C.L. Efficient Model Checking in Fragments of the Propositional Mu-Calculus (Extended Abstract). LICS 1986.

[EJ 88] Emerson, E.A., Jutla, C.S. The Complexity of Tree Automata and Logics of Programs (Extended Abstract), FOCS 1988.

[EN 95] Emerson, E.A., Namjoshi, K.S. Reasoning about Rings. POPL 1995.

[EN 96] Emerson, E.A., Namjoshi, K.S. Automated Verification of Parameterized Synchronous Systems, CAV 1996.

[EN 98] Emerson, E.A., Namjoshi, K.S. On Model Checking Nondeterministic Infinite State Systems, LICS 1998.

[EN 98a] Emerson, E.A., Namjoshi, K.S. Verification of a Parameterized Bus Arbitration Protocol, CAV 1998.

[ES 93] Emerson, E. A., Sistla, A. P. Symmetry and Model Checking, 5th CAV, Springer-Verlag LNCS 697.

[ES 95] Emerson, E.A., Sistla, A.P. Utilizing Symmetry when Model Checking under Fairness Assumptions: An Automata-theoretic approach. CAV 1995.

[ESr 90] Emerson, E.A., Srinivasan, J. A decidable temporal logic to reason about many processes. PODC 1990.

[Finkel 90] Finkel, A. Reduction and Covering of Infinite Reachability Trees, *Information and Computation* vol. 89, 1990.

[Fraisse 86] Fraisse, R. *Theory of Relations*, Studies in Logic and the Foundations of Mathematics, vol. 118, North-Holland, 1986.

[Francez 86] Francez, N. *Fairness*. Springer Verlag, 1986.

[FR 88] Finkel, A., Rosier, L. A survey on the decidability questions for classes of fifo nets. in Advances in Petri Nets, 1988, LNCS 340.

[GS 92] German, S. M., Sistla, A. P. Reasoning about Systems with Many Processes. J.ACM, Vol. 39, Number 3, July 1992.

[GV 90] Groote, J.F.; Vaandrager, F. An efficient algorithm for branching bisimulation and stuttering equivalence, ICALP 1990.

[GW 83] Galperin, H., Wigderson, A. Succinct representations of graphs, *Information and Control*, March 1983.

[GW 89] van Glabbeek, R. J., Weijland, W. P. Branching time and abstraction in bisimulation semantics. in *Information Processing 89*, Elsevier Science Publishers, North-Holland, 1989.

[GFMdR 83] Grumberg, O., Francez, N., Makowski, J., de Roever, W-P. A proof rule for fair termination, in *Information and Control*, 1983.

[Henzinger 95] Henzinger, T. A. Hybrid automata with finite bisimulations, in ICALP 1995.

[Hoare 78] Hoare, C. A. R. Communicating Sequential Processes, *Communications of the ACM*, 1978.

[Hoare 85] Hoare, C.A.R. **Communicating Sequential Processes**, Englewood Cliffs, NJ: Prentice Hall International, 1985.

[HB 95] Hojati, R., Brayton, R. Automatic Datapath Abstraction in Hardware Systems, CAV 1995.

[HM 85] Hennessy, M., Milner, R. Algebraic Laws for Nondeterminism and Concurrency. *Journal of the ACM*, 1985.

[HS 96] Havelund, K., Shankar, N. Experiments in Theorem Proving and Model Checking for Protocol Verification, FME, 1996.

[ID 93] Ip, C., Dill, D. Better verification through symmetry. Proc. 11th Intl. Symp. on Computer Hardware Description Languages and their Applications.

[ID 96] Ip, N., Dill, D. Verifying systems with replicated components in Murϕ, CAV 1996.

[Keller 76] Keller, R. M. Formal verification of parallel programs. *Communications of the ACM*, July 1976.

[KK 91] Klarlund, N., Kozen, D. Rabin measures and their applications to fairness and automata theory, in LICS 1991.

[KM 69] Karp, R., Miller, R. Parallel Program Schemata, *J.CSS*, vol. 3., 1969.

[KM 89] Kurshan, R. P., McMillan, K. A Structural Induction Theorem for Processes, PODC 1989.

[KMMPS 97] Kesten, Y., Maler, O., Marcus, M., Pnueli, A., Shahar, E., Symbolic model checking with rich assertional languages, in CAV 1997.

[KS 90] Kanellakis, P.C., Smolka, S.A. CCS Expressions, Finite State Processes, and Three Problems of Equivalence, in *Information and Computation*, May 1990.

[Lamport 80] Lamport, L. "Sometimes" is Sometimes "Not Never", in *POPL*, 1980.

[LSY 94] Li, J., Suzuki, I., Yamashita, M. A New Structural Induction Theorem for Rings of Temporal Petri Nets. IEEE Trans. Soft. Engg, vol. 20, No. 2, February 1994.

[LP 85] Litchtenstein, O., and Pnueli, A., Checking That Finite State Concurrent Programs Satisfy Their Linear Specifications, POPL 85.

[Long 93] Long, D. Model Checking, Abstraction, and Compositional Verification. Ph.D. Thesis, Carnegie-Mellon University, 1993.

[Lubachevsky 84] Lubachevsky, B. An Approach to Automating the Verification of Compact Parallel Coordination Programs I. *Acta Informatica* 21, 1984.

[CM 88] Chandy, K. M., Misra, J. *Parallel Program Design : A Foundation*, Addison-Wesley Publishers, 1988.

[Milner 71] Milner, R. An Algebraic Definition of Simulation Between Programs, Proceedings of the 2nd International Joint Conference on Artificial Intelligence, William Kaufmann, September 1971.

[Milner 80] Milner, R. *A Calculus for Communicating Processes*, LNCS Vol. 92, Springer Verlag, 1980.

[Milner 90] Milner, R. *Communication and Concurrency*, Prentice-Hall International Series in Computer Science. ed. C.A.R Hoare.

[McMillan 92] McMillan, K., Symbolic Model Checking: An Approach to the State Explosion Problem, Ph.D. Thesis, Carnegie-Mellon University, 1992.

[Minsky 61] Minsky, M. Recursive unsolvability of Post's problem of "tag" and other topics in the theory of Turing machines", Ann. of Math., 1961.

[MP 92] Manna, Z., Pnueli, A. Temporal Logic of Reactive and Concurrent Systems: Specification, Springer-Verlag, 1992.

[MP 94] Manna, Z., Pnueli, A. Verification of Parameterized Programs. in *Specification and Validation Methods* (E. Borger, ed.), Oxford University Press, 1994.

[Namjoshi 97] Namjoshi, K. S. A Simple Characterization of Stuttering Bisimulation, FST & TCS, 1997.

[dNV 90] de Nicola, R., Vaandrager, F. Three logics for branching bisimulation. in LICS 1990. Full version in *Journal of the ACM*, 42(2):458-487, 1995.

[Park 81] Park, D.M.R., Concurrency and Automata on Infinite Sequences, Proceedings of the 5th GI Conference on Theoretical Computer Science, LNCS 104, Springer-Verlag, Berlin, 1981.

[PT 87] Paige, R., Tarjan, R. E. Three partition refinement algorithms. *SIAM Journal of Computing*, December 1987.

[Pnueli 77] Pnueli, A. The Temporal Logic of Programs. FOCS 1977.

[Pnueli 85] Pnueli, A. Linear and Branching Structures in the Semantics and Logics of Reactive Systems. ICALP 1985.

[PD 95] Pong, F., Dubois, M. A New Approach for the Verification of Cache Coherence Protocols. *IEEE Transactions on Parallel and Distributed Systems*, August 1995.

[QS 82] Queille, J.P., J. Sifakis, Specification and Verification of Concurrent Systems in CESAR, Proc. of the 5th International Symposium on Programming, LNCS#137, Springer–Verlag, April 1982.

[Rackoff 78] Rackoff, C. The covering and boundedness problem for Petri Nets and Vector Addition Systems. *TCS*, 1978.

[Reisig 85] Reisig, W. Petri Nets. An Introduction, Volume 4 of Monographs on Theoretical Computer Science. Springer-Verlag, 1985.

[RS 85] Reif, J., Sistla, A. P. A multiprocess network logic with temporal and spatial modalities. *JCSS* 30(1), 1985.

[RS 93] Rho, J. K., Somenzi, F. Automatic Generation of Network Invariants for the Verification of Iterative Sequential Systems. CAV 1993, LNCS 697.

[Suzuki 88] Suzuki, I. Proving properties of a ring of finite state machines. IPL 28.

[SAE 92] SAE J1850 Class B data communication network interface. Society of Automotive Engineers, Inc., 1992.

[SC 85] Sistla, P., Clarke, E.M. The Complexity of Propositional Linear Temporal Logics, *Journal of the ACM*, 1985.

[SE 89] Streett, R.S., Emerson, E.A. An automata theoretic decision procedure for the propositional mu-calculus. *Information and Computation*, 81, 1989.

[SG 89] Shtadler, Z., Grumberg, O. Network Grammars, Communication Behaviours and Automatic Verification. In J.Sifakis (ed), *Automatic Verification Methods for Finite State Systems*, Springer-Verlag, LNCS 407.

[SVW 87] Sistla, P., Vardi, M., Wolper, P. The Complementation Problem for Büchi Automata with Applications to Temporal Logic. TCS 49: 217-237 (1987)

[Tarski 55] Tarski, A. A Lattice-Theoretical Fixpoint Theorem and its Applications, *Pacific J. Math.*, 1955.

[Thomas 90] Thomas, W. Automata on Infinite Objects, *Handbook of Theoretical Computer Science*, vol. B.

[Vardi 87] Vardi, M. Verification of Concurrent Programs - The Automata Theoretic Framework. in *LICS*, 1987. Full version in *Annals of Pure and Applied Logic*, 51:79-98, 1991.

[Vardi 94] Vardi, M. An Automata-theoretic Approach to Linear Temporal Logic, Proceedings of Banff Higher Order Workshop on Logics for Concurrency, F. Moller, ed., Springer-Verlag LNCS.

[VW 86] Vardi, M., Wolper, P. An Automata-theoretic Approach to Automatic Program Verification, Proc. IEEE LICS 1986.

[Vernier 93] Vernier, I. Specification and Verification of Parameterized Parallel Programs. Proc. 8th International Symposium on Computer and Information Sciences, Istanbul, Turkey, 1993.

[Wolper 86] Wolper, P. Expressing Interesting Properties of Programs in Propositional Temporal Logic, POPL 1986.

[WL 89] Wolper, P., Lovinfosse, V. Verifying Properties of Large Sets of Processes with Network Invariants. In J.Sifakis (ed), *Automatic Verification Methods for Finite State Systems*, Springer-Verlag, LNCS 407.

Vita

Kedar Sharadchandra Namjoshi was born in Calcutta, India, on the 30th of November 1968 to Sharadchandra and Vijayalaxmi Namjoshi. After graduating in 1986 from Loyola High School, Pune, he studied Computer Science and Engineering at the Indian Institute of Technology at Madras, graduating with a Bachelor of Technology degree in 1990. He joined the graduate program at the University of Texas at Austin the same year with a MCD fellowship. Chitra Phadke and he were married on the 25th of June, 1995. In his spare time, he particularly enjoys reading and playing racquetball and squash.

Permanent Address: 1011/17, Chatushringi Road, Pune, India-411016.

This dissertation was typeset with LaTeX 2_ε[1] by the author.

[1] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin.