Automatic Verification of Parameterized Systems

by

Jiazhao (Jessie) Xu

A Dissertation Submitted in Partial Fulfillment

of the Requirements for the Degree of

Doctor of Philosophy

Department of Computer Science

New York University

May, 2005

Advisor: Amir Pnueli

Dedication

To my parents and
the memory of the late Dr. Robert Paige

Acknowledgements

The road leading to the Doctor's Degree of Philosophy in Computer Science has been a long and winding journey for me. It started with the encouragement of my parents who inseminated in me the love of science and intellectual fulfillment. At times when I was in serious doubt about my academic choices and was near the point of giving up all my research it always came back to this very basic point.

The reality of being a graduate student can be harsh at times, and the experience of being a part-time Ph.D. student and a full-time working mother is a once in a lifetime adventure, and I am very glad that I had it. For that I owe abundance of thank-yous to my husband Cris, whose insistence upon completing the course made me carry on, whose sharing of child-caring and house work gave me most precious time, and whose sense of humor and good spirit melt away my frustrations yet spared me no nonsense. Another person whose life and encouragement moved me so much was the late Professor of computer science at NYU, Dr. Robert Paige. The last few months of his life was an inspiration to us all. Even at the very late stage of his lung cancer he still cared about his students' future like mine more than his health. It was he who suggested to me to continue my research in formal methods with Prof. Amir Pnueli and showed me what it meant to pursue your goals and never give up.

The research presented in this thesis probably would not have happened without the guidance of my thesis advisor, Prof. Amir Pnueli. It is such a great honor to be given the opportunity to work with the pioneer of the field. His academic insight always amazes me. Being able to take his courses and to work with him on research problems are the highlights of my academic life. As his former student now an accomplished scholar, Lenore Zuck, once put it, "Amir is the best academic advisor that a graduate student can hope for", could not be more true. I almost regret my graduating from NYU for losing the guidance from

Amir and the opportunity of learning more about doing research from him.

I owe a thank-you to all my friends and colleagues who supported me on my academic pursuit. In particular I would like to thank Lenore Zuck for her knowledge and insight in formal methods which inspired many of the results in this thesis; I would like to thank my colleagues Geert Janssen, Andreas Kuehlmann for stimulating discussions on formal verification and my IBM team Jason Baumgartner, Victor Rodriguez, Mark Williams, Viresh Paruthi, Florian Krohm, Bob Kanzelman, Hari Mony, Fadi Zaraket, Anthony Hackner, Travis Pouarz, Yee Ja, Scott Mack, Jim Swift and my former manager Linda Ryan-Doolittle for their support of my education goals and their dedication to research and development of formal methods. One positive facet of being a part-time Ph.D. student is that I am able to witness the real problems that a verification engineer faces in the industrial world, which provide motivation and guidance for academic research. I also would like to thank Prof. Alan Siegel for his kind encouragement and many interesting perspectives on life and research. I would deliver a special thank-you to the graduate student coordinator, Anina Karmen, for her friendship and words of wisdom at those harsh times. And many thanks go to my friends over the years, Yi Wang, Rong Lu, YuHua Lin, Mei-Ting Hsu, Jose Tierno and Marianne Knirsch, whose friendship brighten the days of my life. I would like to thank Prof. Benjamin Goldberg, Prof. Clark Barrett for being on my thesis committee and giving me wonderful suggestions and support on my thesis writing. Also I would like to credit my school work to IBM Corporation's work-study program and to applaud this company's farsighted vision to encourage and support its employees' career development. Now it comes to the last but not the least, I would like to thank other members of my family for always standing by me: my children Brian and Kenneth who can not yet read this thank-you note for being such a joy in my life and making everything meaningful; my mother-in-law Marlene who sacrificed some of her weekend hours to help with kids to free me some extra time for my school work.

To whom I might have forgotten to mention here, I owe you a sincere apology and thank you!

Abstract

Verification plays an indispensable role in designing reliable computer hardware and software systems. With the fast growth in design complexity and the quick turnaround in design time, formal verification has become an increasingly important technology for establishing correctness as well as for finding difficult bugs. Since there is no "silver-bullet" to solve all verification problems, a spectrum of powerful techniques in formal verification have been developed to tackle different verification problems and complexity issues. Depending on the nature of the problem whose most salient components are the system implementation and the property specification, a proper methodology or a combination of different techniques is applied to solve the problem.

In this thesis, we focus on the research and development of formal methods to *uniformly* verify parameterized systems. A parameterized system is a class of systems obtained by instantiating the system parameters. Parameterized verification seeks a single correctness proof of a property for the entire class. Although the general parameterized verification problem is undecidable [AK86], it is possible to solve special classes by applying a repertoire of techniques and heuristics. Many methods in parameterized verification require a great deal of human interaction. This makes the application of these methods to real world problems infeasible. Thus, the main focus of this research is to develop techniques that can be automated to deliver proofs of safety and liveness properties.

Our research combines various formal techniques such as deductive methods, abstraction and model checking. One main result in this thesis is an automatic deductive method for parameterized verification. We apply small model properties of Bounded Data Systems (a special type of parameterized system) to help prove deductive inference rules for the safety properties of BDS systems. Another methodology we developed enables us to prove liveness properties of parameterized systems via an automatic abstraction method

called *counter abstraction*. There are several useful by-products from our research: A set of heuristics is established for the automatic generation of program invariants which can benefit deductive verification in general; also we proposed methodologies for the automatic abstraction of fairness conditions that are crucial for proving liveness properties.

Table of Contents

List of Figures

List of Appendices

Chapter 1
Introduction

1.1 Overview of Verification of Parameterized Systems

Verifying that a piece of hardware or software design functions as *intended* has always been an important task for computer engineers. Methods and techniques to perform verification tasks evolve over time. With the increasing presence and involvement of computer hardware and software in daily life, checking the safety and reliability of these systems has become essential and sometimes even critical. A bug in a system can cause not only inconveniences such as data loss and service delays, but in some unfortunate cases disastrous consequences such as aviation accidents and nuclear plant malfunctions. The damage cost of a bug and the expense for its fix can both be very expensive; Intel's FPU bug, the Y2K glitch, and network crippling virus attacks consumed millions of dollars as well as production time. Therefore it is an imperative task for computer scientists and engineers to develop advanced verification technologies that will support the development of reliable systems despite the growing complexity of designs.

Formal verification has emerged as a promising verification method because of its strength in areas where other verification methods are inadequate:

- It can provide a correctness proof by using methods that are relatively inexpensive in comparison to exhaustive simulation.
- It can find tough bugs in cases where simulation fails due to complexity.
- It can be applied to software programs and hardware systems.
- It can verify infinite-state systems.

The key distinction between formal verification (or *formal methods*) and other verification techniques lies in the utilization of mathematically-based formal languages, techniques and tools to specify and verify hardware and software systems. It provides a mathematical model for the system under study, specifies the properties (or rules) that the system should comply with, and then applies mathematical techniques to check that the model satisfies the properties.

According to their expressive power and approach towards formalization, formal methods fall into the following major categories: *model checking* ([CE81], [QS81]) and *deductive methods* (we discount *equivalence checking* here due to its limited application to hardware verification). To choose an appropriate proof method for a verification task, one should consider the following aspects:

- **Proving power.** One must choose a method with sufficient proving power. For example, model checking can only be applied to finite state systems, thus can not be applied to an infinite-state system without first abstracting or reducing the system to a finite-state one.

- **Soundness and completeness of the proof method.** A viable proof method must be sound, i.e. every statement that it proves to be $true/false$ must actually be true/false. The completeness of the method indicates whether every statement is provable. An incomplete method can cause the proof process to be nonterminating or simply inapplicable.

- **The degree of automation.** With the rapid advance of technology, most systems are too complicated to be proven by hand. Software tools are deployed to fully or partially automate the proof process. The degree of automation directly affects the applicability of the method.

- **The complexity of the method (assuming that the method can be automated).** A lower computational complexity greatly increases the scalability of the method.

- **The diversity of the methodology.** Due to the complexity of design and verification, a successful verification job usually requires that we reduce the model size by applying abstraction and other model reduction techniques to extract reasonably sized subcomponents. Thus a viable proof methodology should consider the support for a few major techniques of this nature, such as:
  - **Abstraction.** Proofs for an abstracted system that can imply the validity of the proofs on its concrete counterpart.
  - **Compositional proofs.** Proofs for a large system that can be constructed easily from the proofs of its components ([CLM98]).
  - **Inductive proofs.** It supports uniform proofs for parameterized systems that are described inductively.

Background on Uniform Verification of Parameterized Systems

The problem of *uniform verification of parameterized systems* is one of the most challenging problems in verification today. Given a parameterized system $S(N) : P[1] \| \cdots \| P[N]$ and a property p, uniform verification attempts to verify $S(N) \models p$ for every $N > 1$. Apt and Kozen proved that, in general, the parameterized verification problem is undecidable, even with finite state components [AK86]. However, for restricted families of parameterized systems the problem is decidable. This is well illustrated in the work of German and Sistla [GS92] and Emerson and Namjoshi [EN95], [EN96], where the systems under consideration are composed of synchronously communicating processes. However their methods to deal with asynchronous systems where processes communicate by shared variables turned out to be much more ad hoc; for instance [EK00] presents different algorithms to deal with systems in which the guards are all disjunctive and systems in which the guards are all conjunctive. Other factors that affect the decidability of the problem include system topologies and property types. Most of the methods target safety properties and only a few address liveness properties.

In contrast to putting restrictions on systems and properties to make the problem decidable, another approach to the problem is to devise sound but incomplete methods in the hope that they can be effective on certain types of systems. One of the promising approaches to the verification of infinite-state systems (and a parameterized system is essentially such a system, due to the unbounded value of the parameter N), is the method of *finitary abstraction* in which we abstract an infinite-state system into a finite-state one, which can then be model-checked. A general theory of finitary abstraction which jointly abstracts a system together with the property to be proven is presented in [KP00c, KP00a]. This method can be applied to the verification of arbitrary LTL properties, including liveness properties. A few notable works proceeding in this direction include using explicit induction [EN95] network invariants and implicit induction [LHR97], [WL89] network invariants, abstraction and approximation of network invariants [CGJ95], and structural induction that handles networks with parameterized topologies [CR00]. Most of these methods require the user to provide auxiliary constructs or abstract mappings for constructing network invariants. In both the verification of Burn's mutual exclusion protocol [JL98] and Szymanski's algorithm [GZ98, MAB+94] the user has to provide abstraction functions or lemmas. Another very useful technique used in parameterized verification is symmetry reduction, which is studied extensively in [CEFJ96, CFJ93],[ES96, ES97]. The work in [ID96] shows how to detect symmetries by inspection of the system description.

For the automatic incomplete approaches, a class of systems which includes bounded-data systems may be analyzed by representing linear configurations of processes as a word in a regular language [KMM+97, ABJN99, JN00]. In many cases, the analysis procedure diverges and special user-suggested acceleration procedures have to be applied, and this in turn breaks the automatic nature of the process. Other works utilize symmetry reduction [GS97] and compositional methods [McM98] that combine automatic abstraction with finite-instantiation.

Most approaches only deal with safety properties. Notable work in verifying liveness properties of parameterized systems includes the PAX system [BLS01] which is based on

predicate abstraction [BBM95]. In [BLS01] parameterized systems are modeled as WS1S systems [BBLS00]. However, the abstract space is determined by the assertions appearing in the transition system and in the temporal property to be verified. As a result, the structure of the abstraction varies from case to case and the computation of the added fairness requirements depends on a marking algorithm which has to be re-applied for each case separately, searching for an appropriate well-founded ranking.

Even though the theory of linear abstraction, as presented in [KP00c], is aimed at verifying liveness (and general LTL) properties, it admits that the general recipe of abstraction is often too weak to provide a working finitary abstraction. In order to obtain a complete method, [KP00c] suggests that we first augment the system under consideration by an auxiliary monitoring module and then abstract the composed system. While this may be theoretically satisfactory, no method is provided for designing the augmenting monitor.

1.2 Contribution and Outline of the Thesis

The main goal of this thesis research is to develop *automatic* formal methods for verification of parameterized systems. Uniform verification of parameterized systems is a theoretically challenging problem with a broad application base such as protocol verification and array verification in hardware systems. Automatic formal methods can further the cause by making it applicable to real world problems. Like most formal methods in this area, our study is limited to special classes of parameterized systems due to the undecidability of the general problem. However it distinguishes itself from previously proposed methods in the following aspects:

- The main focus is on automatic formal methods. As noted before, the degree of automation is an important characteristic of a formal method. Algorithmic verification makes the application of a formal method to real world problems more attractive. Formal methods that utilize abstraction usually require the user to come up with abstract mapping or auxiliary constructs, this is the major road block to automation. The acceleration procedures ([PS00]) that apply to systems by representing linear

configuration of processes as a word in a regular language are automatable but often diverge, and in consequence require special acceleration schemes from the user, which again breaks the automatic process. Our methods seek to avoid such shortcomings.

- Our studies of invisible invariant methods produced encouraging results towards automatic deductive verification of parameterized systems. The achievements and lessons learned shed light on the task of automating deductive verification.

- We tackled the problem of automatic verification of liveness properties of parameterized systems using counter abstraction. Verification of liveness properties is a much harder problem than verifying safety properties in the context of parameterized systems. Our results demonstrated a viable approach and various methods for abstracting fairness conditions in a parameterized system.

Contribution

Here is a list of the main contributions of the thesis:

- Extended the small model theorem to multi-indexed stratified and unstratified BDS systems.

- Applied the method of invisible invariants to test cases such as Bakery (stratified BDS), Peterson (unstratified BDS), Fischer (real time programs).

- Generalized the method of invisible invariants to waiting-for formulas, and to general safety properties.

- Developed the method of counter abstraction.

- Developed methods for abstracting justice requirements using justice-suppressing assertions, and methods for abstracting compassion fairness requirements.

- Implemented and applied counter abstraction to test cases such as Bakery and Szymansky.

Outline

The organization of the thesis is as follows:

- Chapter 1 is a general introduction to parameterized verification and related work.
- Chapter 2 provides basic concepts and background materials necessary for the reader to understand the main results in this thesis.
- Chapter 3 describes the method of *invisible invariants*. We present the small model theorems for various *Bounded Data Systems*. Then we discuss in detail different heuristics in automatic generation of inductive assertions.
- Chapter 4 presents the method of *counter abstraction.* The focus is on the safe and automatic abstraction of fairness conditions. We demonstrate the implementation and the application of this method on various examples.
- Chapter 5 summarizes the results and discusses possible future research areas.

1.3 A Word on Notation

Most notations are introduced throughout our presentation as needed. Here is a list of frequently used logical notations and expressions to start with:

– Quantified expressions are written in the format $(\mathbf{Q}x \in r : p)$, where $\mathbf{Q}$ is the quantifier (e.g., $\exists, \forall$), x the *bound* variable, r the range, and p the expression being quantified. When the range r of x is clear from the context, it may be dropped, resulting in the abbreviated form $(\mathbf{Q}x : p)$. To make the expression succinct sometimes we abbreviate $(\mathbf{Q}x : x \neq y \wedge p)$ simply as $(\mathbf{Q}x \neq y : p)$

– For the *pre-condition* and *post-condition* of a transition, we use the logic expression, $\varphi \wedge \rho \rightarrow \varphi'$, where ρ is the logic representation of the entire transition relation and ψ' represents the evaluation of the assertion ψ using the next state values of its variables. And we refer to ψ' as the *primed version* of ψ.

– $P[1]\|P[2]$ is used to denote the asynchronous composition of processes $P[1]$ and $P[2]$ which allows the processors to interleave their transitions. For the synchronous composition of the processes we use the notation $P[1] \;|||\; P[2]$ which requires all processors to execute their transition steps synchronously.

– For an assertion $p(V)$ and transition relation $\rho(V, V')$, we define the ρ-**successor** of p, denoted by $p \diamond \rho$, by the formula

$$p \diamond \rho = unprime(\exists V : p(V) \wedge \rho(V, V'))$$

The operation $unprime$ is the syntactic replacement of each primed occurrence v' by its unprimed version v. We can also define the ρ-**successors** of p through iterated ρ-successor computation,

$$p \diamond \rho^* = p \vee p \diamond \rho \vee (p \diamond \rho) \diamond \rho \vee ((p \diamond \rho) \diamond \rho) \diamond \rho \vee \ldots,$$

which for finite-state systems is guaranteed to terminate.

Acronyms are used in this thesis as a succinct way of referring to commonly used terms, such as temporal logics, software languages, tool sets, or mathematical representations. Typically acronyms use the small-caps font, such as LTL, CTL. Here are a few commonly used acronyms:

– BDD stands for Binary Decision Diagram ([Bry86]). It is a canonical binary tree representation of a boolean function. BDD was initially used in the SMV model checker [McM92] by Ken McMillan and later widely adopted as an efficient technique to symbolically represent assertions.

– TLV stands for Temporal Logic Verifier. It is a verification tool set [PS96] based on the SMV system and with programmable user interface.

– SPL stands for Simple Programming Language. It is a simple concurrent programming language introduced in [MP95].

– SCS stands for Strongly Connected Subgraph, which refers to a directed subgraph where any two nodes in the subgraph are connected by *some* path. In other literature it might also be named *Strongly Connected Component* (SCC). In general we only consider non-singular (or non-trivial) SCSs, i.e. excluding the case that the subgraph consists of a singleton node with no edges. One needs to take extra care to distinguish a singular SCS from a special SCS which consists of a single node with a self-looping edge.

In the thesis there are quotations from TLV program scripts and the SPL programs. For TLV programs we use a distinct font, and for SPL programs we usually typeset using framed figure. In the SPL programs we put all the SPL keywords in bold-face, such as **local**, **while**.

Chapter 2
Background

2.1 Temporal Logic

Temporal Logics are special modal logics that use modalities **A**, **E**, **F**, **G**, **X** and **U** to specify time-dependent properties such as that a designated state will *eventually* be reached or some bad state will *never* occur in a valid computation. It was first introduced by Pnueli in 1977 ([Pnu77]) to reason about concurrent programs. Corresponding to the different syntactic rules for combining temporal modalities with boolean connectives there are different temporal logics, each with its unique expressive power and possibly distinctive semantics. Commonly used ones are LTL (Linear Temporal Logic) and CTL (Branching Time Logic or Computation Tree Logic), both sublogics of CTL*.

Definition 1 CTL* *is a temporal logic which includes two classes of formulas: state formulas and path formulas.* CTL* *formulas are composed of path quantifiers* **A**, **E** *and temporal operators* **F**, **G**, **U**, **X** *as well as boolean connectives. The rules of forming state formulas (S1 – S3) and path formulas (P1 – P2) are as follows:*

S1. An 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 $\mathbf{A}p$ is a state formula.

P1. A state formula p is also a path formula.

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

P3. If p, q are path formulas then so are $\mathbf{X}p$ and $p\mathbf{U}q$.

Notice that we don't mention the modalities **E**, **F**, **G** in the above definition because they can be obtained from the three basic modalities **A**, **X** and **U** using logical operations:

- **E**f = $\neg$ **A**$\neg f$
- **F**f = *true* **U**f
- **G**f = $\neg$ **F**$\neg f$

The semantics of a temporal formula are defined with respect to a Kripke structure M. A Kripke structure is a nondeterministic finite state machine whose states are labeled with boolean variables, which represent an evaluation of the various predicates in that state. It may be extended by including fairness constraints. The semantics of most temporal logics are defined over a Kripke structure.

Definition 2 (Kripke Structure) *Let AP be a set of atomic propositions. A **Kripke Structure** M over AP is defined by a four tuple (Q, δ, λ, I), where*

- *Q is a nonempty set of states,*
- *$\delta \subseteq Q \times Q$ is a transition relation that is total, i.e. for every state $s \in Q$ there is a state $s' \in Q$ such that $\delta(s, s')$,*
- *$\lambda : Q \rightarrow 2^{AP}$ is a function that labels each state with the set of atomic propositions true in that state,*
- *$I \subseteq Q$ is the set of initial states.*

An infinite path in the graph of a Kripke structure is called a *fullpath*, e.g. $x = (s_0, s_1, \ldots)$, denotes a fullpath starting at an initial state s_0 followed by s_1 and so on, where any two consecutive states in the fullpath s_i, s_{i+1} must satisfy the transition relation. 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 (fullpath x). When M is understood, we write simply $s \models p$ $(x \models p)$. We leave the detailed definition of the semantics of path and state formulas for a given temporal logic to the later sections as needed.

Sublogics of CTL* are defined either by restrictions on the operators allowed, or by restrictions on the ways in which the operators can be combined. Here are a few commonly used sublogics of CTL*.

- LTL (Linear Temporal Logic) is obtained by leaving out (S2) and replacing (P1) by:

  P1′. If $p \in AP$, then p is a path formula.

  This effectively allows only state formulas of the form $\mathbf{A}f$ where f is a path formula in which the only state subformulas permitted are atomic propositions. In consequence the path quantifier **A** is often omitted in LTL notations.

- CTL (Computation Tree Logic) is obtained by restricting the kinds of path formulas allowed, that is by leaving out (P1), (P2) and replace (P3) with:

  P3′. If p, q are state formulas then $\mathbf{X}p$ and $p\mathbf{U}q$ are path formulas.

  This ensures that path quantifiers **A**, **E** are always attached to a single temporal operator. For example, $\mathbf{AG}(p \Rightarrow \mathbf{AF}q)$ is a CTL formula, whereas $\mathbf{AG}(p \Rightarrow \mathbf{F}q)$ is not.

- ACTL* is obtained by considering only positive formulas, built up from the literals (propositions or negated propositions), without negation, and using only the **A** path quantification operator. The dual logic is known as ECTL*.

- CTL$^*\backslash$X is obtained by leaving out the next operator. This makes the logic insensitive to *stuttering* (repetitions of the same state).

2.1.1 Linear Temporal Logic

For the specification of reactive systems we use temporal logic, in particular we choose LTL, linear temporal logic, for our studies. To comply with the prevalent notations used in LTL, we use the temporal operators $\Box, \Diamond$ instead of the CTL* operators **G**, **F**. In LTL, a temporal

formula is constructed from state formulas to which we apply the boolean operators $\neg$ and $\vee$, and the following basic temporal operators:

$$\bigcirc \text{ – Next} \quad \ominus \text{ – Previous}$$
$$\mathcal{U} \text{ – Until} \quad \mathcal{S} \text{ – Since}$$

Additional temporal operators can be defined as follows:

$$\begin{aligned}
\Diamond p &= true\, \mathcal{U}\, p \\
\Box p &= \neg \Diamond \neg p \\
p\, \mathcal{W}\, q &= \Box p \vee (p\, \mathcal{U}\, q) \\
\Diamond\!\!\!- p &= true\, \mathcal{S}\, p \\
\Box\!\!\!- p &= \neg \Diamond\!\!\!- \neg p \\
p\, \mathcal{B}\, q &= \Box\!\!\!- p \vee (p\, \mathcal{S}\, q)
\end{aligned}$$

Another useful derived operator is the *entailment* operator, defined by:

$$(p \Rightarrow q) \iff \Box(p \rightarrow q)$$

We refer to $\bigcirc$, $\mathcal{U}$, $\Diamond$, $\Box$, and $\mathcal{W}$ as *future operators* and to $\ominus$, $\mathcal{S}$, $\Diamond\!\!\!-$, $\Box\!\!\!-$, and $\mathcal{B}$ as *past operators*. A temporal formula that contains no future operators is called a *past* formula. A temporal formula that contains no past operators is called a *future* formula. A state formula (assertion) is both a past and a future formula.

To define the semantic meaning of LTL formulas, we use linear computation sequences instead of a computation tree which is more suited for defining CTL semantics. A *model* for a temporal formula p is an infinite sequence of states $\sigma : s_0, s_1, \ldots$, where each state s_j provides an interpretation for the variables mentioned in p. Given a model σ, we present an inductive definition for the notion of a temporal formula p holding at a position $j \geq 0$ in

σ, denoted by $(\sigma, j) \models p$.

$$
\begin{array}{lcl}
(\sigma, j) \models p & \iff & s_j \models p, \quad (p \text{ is a state formula}) \\
(\sigma, j) \models \neg p & \iff & (\sigma, j) \not\models p \\
(\sigma, j) \models p \vee q & \iff & (\sigma, j) \models p \text{ or } (\sigma, j) \models q \\
(\sigma, j) \models \bigcirc p & \iff & (\sigma, j+1) \models p \\
(\sigma, j) \models p \,\mathcal{U}\, q & \iff & \exists k : k \geq j, (\sigma, k) \models q \wedge (\forall i : j \leq i < k, (\sigma, i) \models p) \\
(\sigma, j) \models \ominus p & \iff & j > 0 \text{ and } (\sigma, j-1) \models p \\
(\sigma, j) \models p \,\mathcal{S}\, q & \iff & \exists k : k \leq j, (\sigma, k) \models q \wedge (\forall i : j \geq i > k, (\sigma, i) \models p)
\end{array}
$$

If $(\sigma, 0) \models p$, we say that p *holds* on σ. A formula is called *satisfiable* if it holds on some model, and it is called *temporally valid* if it holds on all models.

Temporal formulas for specifying program properties can be arranged in a hierarchy that identifies several classes of formulas with distinctive expressive power. The following are canonical forms for these classes of LTL formulas (where p, p_i and q_i are past formulas):

- $\Box\, p$ is a canonical **safety** formula.
- $\Diamond\, p$ is a canonical **guarantee** formula.
- $\bigwedge_{i=1}^{n} [\Box\, p_i \vee \Diamond\, q_i]$ is a canonical **obligation** formula.
- $\Box \Diamond\, p$ is a canonical **response** formula.
- $\Diamond \Box\, p$ is a canonical **persistence** formula.
- $\bigwedge_{i=1}^{n} [\Box \Diamond\, p_i \vee \Diamond \Box\, q_i]$ is a canonical **reactivity** formula.

Formulas that don't belong to the safety class are called **progress** formulas.

Among all classes of temporal formulas we are particularly interested in verifying properties that can be specified by safety and response formulas. In general a *safety property* can be expressed in the form $\Box\, p$ for past formula p. An *invariance* formula is a special form of

a safety formula where p is a state formula. Another interesting subclass of safety properties is the class of *nested waiting-for* formulas with canonical form $p \Rightarrow q_m \, \mathcal{W} \, q_{m-1} \cdots q_1 \, \mathcal{W} \, q_0$ ($p, q_0, q_1, \ldots, q_m$ are state formulas). It can be shown that a nested waiting-for formula can be transformed to a safety formula $\Box \, \psi$ for some past formula ψ. For example one can show that the simple waiting-formula $p \Rightarrow q \, \mathcal{W} \, r$ is equivalent to the canonical safety formula $\Box(\neg q \rightarrow \neg p \, \mathcal{B} \, r)$. Waiting-for formulas are very useful for expressing a rich class of *precedence* properties such as order preservation of messages and bounded overtaking in communication protocols.

A *response* property is a special class of progress properties which can be expressed by a formula $p \Rightarrow \diamondsuit \, q$ for past formulas p and q (it is equivalent to the canonical form $\Box \diamondsuit((\neg p) \, \mathcal{B} \, q)$). In particular we are interested in cases where both p and q are state formulas.

2.2 Fair Discrete Systems

In the previous section we discussed temporal logic LTL as our chosen formalism for specifying system properties. In this section we address the formalization of reactive systems. The computation model we use for reactive systems is the model of *fair discrete system* (FDS), a slight variation on the *fair transition systems* (FTS) model of [MP95].
An FDS $\mathcal{D} : \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C} \rangle$ consists of the following components:

- V: A finite set of typed system variables, containing data and control variables. The set of states (interpretations) over V is denoted by Σ.
- Θ: The initial condition, an assertion (state formula) characterizing the set of initial states.
- ρ: A transition relation, an assertion $\rho(V, V')$, relating the values of present states to the values of next states (which we denote as the *primed version* of the states).
- $\mathcal{J} : \{J_1, \ldots, J_k\}$: A set of *justice* (*weak fairness*) requirements. A justice require-

ment $J \in \mathcal{J}$ is an assertion, intended to guarantee that every computation contains infinitely many J-states (states satisfying J).

– $\mathcal{C} : \{\langle p_1, q_1\rangle, \ldots, \langle p_n, q_n\rangle\}$: A set of *compassion* (*strong fairness*) requirements. A compassion requirement $\langle p, q\rangle \in \mathcal{C}$ is a pair of assertions, intended to guarantee that every computation containing infinitely many p-states also contains infinitely many q-states.

A state t is called a $\mathcal{D}$-successor of a state s if $\rho(s, t')$ holds (t' is the primed version of state t). We require that every state $s \in \Sigma$ has at least one $\mathcal{D}$-successor. This is often ensured by including in ρ the *idling* disjunct $V = V'$ (also called the *stuttering* step). In such cases, every state s is its own $\mathcal{D}$-successor.

A *computation* of an FDS $\mathcal{D} : \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C}\rangle$ is an infinite sequence of states $\sigma : s_0, s_1, s_2, ...$, satisfying the following requirements:

Initiality: s_0 is initial, i.e., $s_0 \models \Theta$.

Consecution: For each $j = 0, 1, ...$, the state s_{j+1} is a $\mathcal{D}$-successor of s_j.

Justice: For each $J \in \mathcal{J}$, σ contains infinitely many J-states.

Compassion: For each $\langle p, q\rangle \in \mathcal{C}$, if σ contains infinitely many p-states, then it also contain infinitely many q-states.

For an FDS $\mathcal{D}$, we denote by $\mathcal{C}omp(\mathcal{D})$ the set of all computations of $\mathcal{D}$. For a given program P (or its computational model $\mathcal{D}$), if the formula p holds on all possible computations in P (or $\mathcal{D}$), then we say that the formula is *P-valid* (or *$\mathcal{D}$-valid*).

2.2.1 Parameterized Systems

The parameterized systems we consider normally consist of a parallel composition of N symmetric processes, all executing the same program which may refer to the *id* of the executing process. In addition, the process programs may access a set of global shared variables and each other's local variables. Access to the local variables of the other processes

can be done under existential or universal quantification. For example we may express the fact that process i holds the biggest ticket $y[i]$ by using a universally-quantified assertion: $\forall j \neq i : y[j] < y[i]$; similarly we may express the condition that only when there exists a process at location d can process i have access to location c through the existential guard: $\exists j \neq i : \pi[j] = d$.

A simple example of such a system is a mutual exclusion program MUX-SEM as shown in Fig. 2.1. Each of the processes is executing exactly the same program and there are

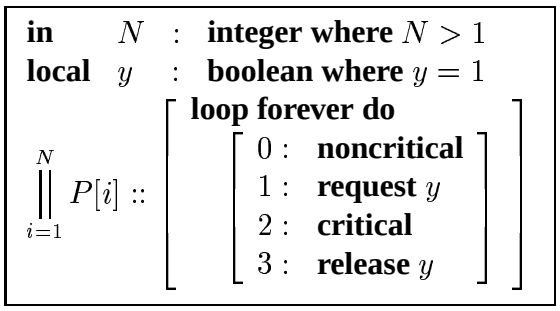


Figure 2.1: Program MUX-SEM

only references to the single shared variable y. The program is written in the simple programming language SPL [MP95, MAB+94]. The semaphore instructions "**release** y" and "**request** y" stand, respectively, for $y := 1$ and $\langle$**when** $y = 1$ **do** $y := 0\rangle$ which represents an atomic operation. The keyword **local** means that variable y is local to the block.

There are several different ways to view a parameterized system $S(N)$. One way views $S(N)$ as a single FDS, where the parameter N is considered an input variable which retains its value throughout the computation. We prefer to consider $S(N)$ as a representation of an infinite family of systems, one for each value assumed by N . In Fig. 2.2, we present the family of FDS's MUX-SEM(N) corresponding to program MUX-SEM.

$$V: \quad \left\{ \begin{array}{ll} N: & \textbf{integer} \\ y: & \textbf{boolean} \\ \pi: & \textbf{array}\ [1..N]\ \textbf{of}\ \{0,1,2,3\} \end{array} \right.$$

$$\Theta: \quad y \ \wedge\ N > 1 \ \wedge\ \forall i : [1..N] : \pi[i] = 0$$

$$\rho: \quad \exists i : [1..N] \left(\begin{array}{ll} & \pi'[i] = \pi[i] \ \wedge\ y' = y \\ \vee & \pi[i] = 0 \ \wedge\ \pi'[i] = 1 \ \wedge\ y' = y \\ \vee & \pi[i] = 1 \ \wedge\ y = 1 \ \wedge\ \pi'[i] = 2 \ \wedge\ y' = 0 \\ \vee & \pi[i] = 2 \ \wedge\ \pi'[i] = 3 \ \wedge\ y' = y \\ \vee & \pi[i] = 3 \ \wedge\ \pi'[i] = 0 \ \wedge\ y' = 1 \end{array} \right)$$
$$\wedge\ \forall j \neq i : \pi'[j] = \pi[j] \ \wedge\ N = N'$$

$$\mathcal{J}: \quad \Big\{ \pi[i] \neq 2 \quad | \quad i \in [1..N] \Big\} \ \bigcup \ \Big\{ \pi[i] \neq 3 \quad | \quad i \in [1..N] \Big\}$$
$$\mathcal{C}: \quad \Big\{ \langle \pi[i] = 1 \ \wedge\ y\ ,\ \pi[i] = 2 \rangle \quad | \quad i \in [1..N] \Big\}$$

Figure 2.2: The FDS Corresponding to Program MUX-SEM(N)

2.3 Deductive Verification Framework

Given a reactive system (or a program) P and a temporal formula φ, program properties are verified using formal deduction based on a set of inference rules ([MP95]). The set of inference rules are provided for each class of temporal formulas. Furthermore, it is shown that this deductive verification framework is *complete* [MP95] for proving temporal properties, that is, every correct property of a system can be formally proven. Another powerful feature of deductive verification is that it can be applied to any reactive system, be it finite or not. This sometimes makes it the only viable choice for verifying infinite systems because other popular methods such as model checking are no longer applicable.

There are several obstacles to the application of deductive verification in practice. The main drawback of the method is that it often needs user guidance or ingenuity to find auxiliary assertions in order to complete the proofs. Secondly, proving inference rules in most cases is not fully automatic; the computational cost can be high as well, especially in the verification of infinite systems. In this thesis we make some efforts to overcome these difficulties and propose effective methodologies to perform automatic deductive verification

on restricted classes of verification problems. Though our main focus is on proving safety properties, the methods have been shown to be extendable to liveness properties [FPPZ04] as well by the recent research work performed by the ACSys group at NYU.

Now we look at a few important inference rules which we will encounter in our later discussion on proving safety properties.

Invariance Rule

An invariance formula is a special safety formula of the form $\Box\, p$ where p is an assertion (a state formula with no temporal operators). To apply deductive verification on invariance formulas, we need to first find one "auxiliary" assertion φ and then establish the validity of the set of premises listed in rule INV (Fig. 2.3).

For assertions φ, p,

$$\begin{array}{ll} \text{I1.} & \Theta \rightarrow \varphi \\ \text{I2.} & \varphi \wedge \rho \rightarrow \varphi' \\ \text{I3.} & \varphi \rightarrow p \end{array}$$

$$\hline$$

$$\Box\, p$$

Figure 2.3: Rule INV for Proving General Invariance

Rule INV claims that if the implications listed in premises I1–I3 are P-state valid (state valid over program P), then $\Box\, p$ is P-valid, i.e., p is a P-invariant. An assertion φ satisfying premises I1, I2 of rule INV is called an *inductive assertion.* It can be shown that if p is a P-invariant, then there always exists an inductive assertion φ stronger than p. For a finite state program P, a trivial (and strongest) inductive assertion is $reach$ which represents all the reachable states of the program.

Simple Waiting-for Rule

A simple waiting-for formula is a special safety formula of the general form $p \Rightarrow q\, \mathcal{W}\, r$ where p, q, r are assertions. The deductive verification of general waiting-for formulas

needs one auxiliary assertion φ in rule WAIT (Fig. 2.4).

For assertions p, q, r, φ,

W1. $p \rightarrow \varphi \vee r$
W2. $\varphi \rightarrow q$
W3. $\varphi \wedge \rho \rightarrow \varphi' \vee r'$

$$p \Rightarrow q \,\mathcal{W}\, r$$

Figure 2.4: Rule WAIT for Proving General Waiting-for

Rule WAIT claims that if the implications listed in premises W1–W3 are P-state valid, then $p \Rightarrow q \,\mathcal{W}\, r$ is P-valid. Note that the justice requirements are not needed to prove the general waiting-for properties because the weak-until operator $\mathcal{W}$ doesn't require the eventual fulfillment of r.

Nested Waiting-for Rule

A nested waiting-for formula is a safety formula of the form $p \Rightarrow q_m \,\mathcal{W}\, q_{m-1} \cdots q_1 \,\mathcal{W}\, q_0$ where $p, q_0, \ldots, q_m$ are assertions. The deductive verification of nested waiting-for formulas calls for $m+1$ auxiliary assertions $\varphi_0, \ldots, \varphi_m$ in rule NWAIT (Fig. 2.5).

For assertions $p, q_0, q_1, \ldots, q_m,$ and $\varphi_0, \varphi_1, \ldots, \varphi_m,$

N1. $p \rightarrow \bigvee_{j=0}^{m} \varphi_j$
N2. $\varphi_i \rightarrow q_i$ for $i = 0, 1, \ldots, m$
N3. $\varphi_i \wedge \rho \rightarrow \bigvee_{j \leq i} \varphi'_j$ for $i = 1, \ldots, m$

$$p \Rightarrow q_m \,\mathcal{W}\, q_{m-1} \cdots q_1 \,\mathcal{W}\, q_0$$

Figure 2.5: Rule NWAIT for Proving Nested Waiting-for

Rule NWAIT claims that if the implications listed in premises N1–N3 are P-state valid, then $p \Rightarrow q_m \,\mathcal{W}\, q_{m-1} \cdots q_1 \,\mathcal{W}\, q_0$ is P-valid. Typically one can choose $\varphi_0 = q_0$, however we still need to find the other m auxiliary assertions $\varphi_1, \ldots, \varphi_m$.

Verifying General Safety Properties

A general safety property can be specified by a formula of the form $\Box\, p$ for some past formula p. In [MP95] an invariance rule INV-P is presented for verifying past formulas. It can be viewed as the (past) extension of the rule INV (Fig. 2.6).

For past formulas φ, p,

P1. $\Theta \rightarrow (\varphi)_0$
P2. $\varphi \wedge \rho \rightarrow \varphi'$
P3. $\varphi \Rightarrow p$

$\Box\, p$

Figure 2.6: Rule INV-P for Proving General Safety Property

The differences between rule INV-P and rule INV are:

- p, φ are no longer state formulas (assertions), instead they are past formulas.
- the use of the entailment operator ($\Rightarrow$) in premise P3 instead of the implication operator ($\rightarrow$) as in premise I3.
- the use of the formula $(\varphi)_0$, the initial version of formula φ ([MP95]), in premise P1.
- the extended definition of φ' as the *past primed version of* φ in premise P2.

In rule INV's I3 we use implication instead of the entailment operator due to the fact that $\Box(\varphi \rightarrow p)$ is P-valid if and only if $\varphi \rightarrow p$ is P-state valid for assertions φ and p.
The initial version $(\varphi)_0$ is a state formula that expresses the truth-value of φ at position 0, which can be defined inductively for past formulas. Similarly the primed version of a past formula φ can be defined inductively as well.

The application of rule INV-P requires generation of the auxiliary past formula φ which is out of the scope of this thesis. Here we explore other methods to perform the deductive verification of general safety properties. Our method utilizes temporal testers to build a combined model of the system and the property, and turns the problem of verifying the

safety problem $\Box\, p$ where p is a past formula into the problem of verifying an invariance formula in the combined system. We will leave further elaboration of the method to Chapter 3.

In summary, deductive verification of safety properties is performed by checking the P-state validity of the premises in the corresponding inference rules for the target properties. This method theoretically is complete, which means that any correct safety property can be verified using the deductive verification framework. However in practice deductive verification can be very expensive in computational costs and often requires human guidance. The two main challenges faced by *automatic* deductive verification are:

- automatic generation of auxiliary assertions, and
- automatic and efficient discharge of proof obligations, i.e. the premises of inference rules.

In Chapter 3 we will address these challenges and lay out a sound (but incomplete) methodology for automatic deductive verification of a restricted class of parameterized systems.

2.4 Data Abstraction

One useful observation that has been made by researchers in formal methods is that, often, to verify a property of a complex system, one can omit a lot of details of the system behavior and instead verify the property on a smaller abstract version of the original system. This observation leads to the concept and effective technique of abstraction ([CGL92]), without which the application realm of formal methods would be much more limited. Formally speaking, abstraction is a general and effective method in formal verification to contain the complexity of verification problems. It has been widely applied in verification of parameterized systems. Only with the essential help of abstraction does it become possible to apply formal techniques such as model checking to verify infinite state systems.

There are many interesting and powerful abstraction techniques such as network invariants, predicate abstraction etc. Most of the abstraction techniques require user assistance in providing key elements and mappings. Since our goal is to pursue automatic verification methods we want to avoid techniques with this undesirable feature, and this leads us to consider one particular abstraction technique called *finitary abstraction*. A general theory of finitary abstraction which jointly abstracts a system together with the property to be proven is presented in [KP00c, KP00a]. The idea behind finitary abstraction (inspired by [CC77]) is to reduce the problem of verifying that a given *concrete* system satisfies its (concrete) specifications, into a problem of verifying that some *finite-state abstract* system satisfies its finite-state (abstract) specifications. In cases where an infinite data domain of a system is abstracted into a finite data domain we refer to the method as *data abstraction*.

Given an FDS $\mathcal{D}$ and a property ψ, finitary abstraction requires the user to define a finitary abstraction mapping α from the variable domain V in FDS $\mathcal{D}$ to a finite abstract variable domain V_α in FDS $\mathcal{D}^\alpha$. A safe abstraction "recipe" will automatically abstract the concrete FDS $\mathcal{D}$ into a finite-state abstract FDS $\mathcal{D}^\alpha$ and the property ψ into a finitary abstract property ψ^α. It should be guaranteed that $\mathcal{D}^\alpha \models \psi^\alpha$ implies that $\mathcal{D} \models \psi$ (but not necessarily vice versa). The following is a brief description of this general abstraction method (please refer to [KP00b] for additional details):
Given an FDS $\mathcal{D} = \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C} \rangle$, and an abstraction scheme α given by a set of abstract variables V_A and a set of expressions $\mathcal{E}^\alpha$ such that $\alpha : V_A = \mathcal{E}^\alpha(V)$. Let $p(V)$ be a (concrete) assertion. Define

$$\begin{array}{lcl} \alpha^+(p) & : & \exists V \Big(V_A = \mathcal{E}^\alpha(V) \ \wedge \ p(V) \Big) \\ \alpha^-(p) & : & \exists V (V_A = \mathcal{E}^\alpha(V)) \ \wedge \ \forall V \Big(V_A = \mathcal{E}^\alpha(V) \ \rightarrow \ p(V) \Big). \end{array}$$

For a given abstract state S, $\alpha^+(p)$ holds over S if p holds over *some* concrete state s that is α-abstracted into S. The abstraction $\alpha^-(p)$ holds over S if S is an abstraction of some concrete state and p holds over *all* concrete states s that are α-abstracted into S. The abstractions α^- and α^+ can be generalized to (positive form) temporal formulas ([KP00b]).

To abstract the temporal formula ψ we apply the abstraction α^- to all of its contained assertions and denote the resulting temporal formula by ψ^α.

The abstract system that corresponds to $\mathcal{D}$ under α is $\mathcal{D}^\alpha = \langle V^\alpha, \Theta^\alpha, \rho^\alpha, \mathcal{J}^\alpha, \mathcal{C}^\alpha \rangle$ where:

$$\begin{aligned}
\Theta^\alpha &= \alpha^+(\Theta) \\
\rho^\alpha &= \alpha^{++}(\rho) \\
\mathcal{J}^\alpha &= \{\alpha^+(J) \mid J \in \mathcal{J}\} \\
\mathcal{C}^\alpha &= \{\langle \alpha^-(p), \alpha^+(q)\rangle \mid \langle p, q\rangle \in \mathcal{C}\}
\end{aligned}$$

where $\alpha^{++}(\rho) = \exists V, V' : \Big(V_{_A} = \mathcal{E}^\alpha(V) \ \wedge \ V'_{_A} = \mathcal{E}^\alpha(V') \ \wedge \ \rho(V, V')\Big)$.

It is proven in [KP00b] that this abstraction method is sound. That is, if $\mathcal{D}^\alpha \models \psi^\alpha$ then $\mathcal{D} \models \psi$. Since formula ψ is an arbitrary temporal formula, this provides a sound abstraction method for verifying liveness as well as safety properties. In fact this method is also relatively complete. It has been shown that for every system $\mathcal{D}$ and temporal property φ, such that $\mathcal{D} \models \varphi$ there exists a (progress) monitor M_δ whose (synchronous) composition with $\mathcal{D}$ does not constrain the computations of $\mathcal{D}$ and a finitary state abstraction mapping α, such that $(D \ ||| \ M_\delta)^\alpha \models \varphi^\alpha$ (the formal definition of the synchronous composition operator $|||$ can be found in Subsection 3.4).

In order to automate the data abstraction we need to find an abstract mapping α, whereafter the data abstraction method proposed in [KP00b] can automatically abstract the system and property. One main goal of ours is to use data abstraction to automatically prove liveness properties of parameterized systems. Two steps are necessary to achieve this goal: devising an automatic abstract mapping method and ensuring that our data abstraction method can automatically abstract strong enough fairness conditions that are crucial in proving liveness properties for the abstract system. Our approach and results will be reported in Chapter 4 of this thesis.

Chapter 3
Parameterized Verification using Invisible Invariants

3.1 Small Model Theorem

In order to prove, using deductive methods, that $\forall N : S(N) \models \psi$, where ψ is a temporal formula, we need to discharge a set of premises in the form of assertions. Here we focus on the special case where ψ is an *invariance* formula of the form $\Box\, \varphi$, where φ is a state formula or a waiting-for formula such as $p \rightarrow (q\, \mathcal{W}\, r)$ for state formulas p, q, r.

Let us start by considering the problem of verifying invariance properties of a parameterized system. The general deductive rule for proving invariance formulas is rule INV as illustrated in the previous chapter (see Fig. 2.3). For a parameterized system we need to establish the truth of premises I1, I2, I3 for every $N > 1$. Since all the premises of rule INV refer to the parameter N we make a slight distinction in notation for the premises as shown the diagram below. In summary, given a parameterized system $S(N)$ with transition relation ρ and initial condition Θ and the property $\psi = \Box\, \varphi$, we wish to prove the premises for the set of inference rules for all $N > 1$. The following is the inference rule INV for an instance of the parameterized system, $S(N)$:

With parameter N, and given assertions φ_N, η_N

I1. $\Theta_N \rightarrow \eta_N$

I2. $\eta_N \wedge \rho_N \rightarrow \eta'_N$

I3. $\eta_N \rightarrow \varphi_N$

$\Box\, \varphi_N$

The auxiliary assertion η_N is *inductive* if it satisfies both I1 and I2.

Since our goal is to establish the truth of premises I1, I2 and I3 for all $N > 1$, one approach is to treat the premises as theorems and apply mathematical reasoning to prove each of the premises separately. In this chapter we consider an alternative approach based on the notion of a small model property.

Definition 3 (Small Model Property) *For a given formula ψ, a parameterized system has the small model property if we can find a small integer K such that if $\forall N \leq K, S(N) \models \psi$ then $\forall N > 1, S(N) \models \psi$. The integer K is sometimes referred to as the **cutoff** value.*

The small model property provides a way to establish the correctness of a logical formula by checking its validity on a finite number of small instances. The derivation of the bound K is highly dependent on the verification method being used. For example in Namjoshi and Emerson's work [EN95] of verifying parameterized token ring networks, they establish the cutoff value for a token ring network based on their abstraction of the symmetric ring structure. This cutoff value is then used to model check the small instances of networks to establish correctness for the entire family. In their work they apply the small model property to a temporal formula of the form $\Box\, p$. In the deductive verification framework our main verification obligation is to discharge the premises (typically state formulas) of the inference rules. Therefore, we apply the small model theorem only to the premises of rule INV. Based on the special form of the state formula we want to discharge, the cutoff value can be derived accordingly from its data signature and logical composition. So when verifying the same token ring network the cutoff value K derived by our method may be different from the cutoff value derived in Namjoshi and Emerson's method.

In order to establish the main small model theorem for the class of parameterized systems which we named *Bounded Data Systems* (BDS), we first need to establish small model properties for a few special logical forms which represent what we encounter in the premises of inference rules for deductive verification of BDS systems. The foremost is the small model property for a special form of assertion called an AER-assertion on a single

index. The methodology and theoretical consequences can be extended to derive small model properties for other types of logic formulas.

3.1.1 Small Model Theorem for AER-Assertions on a Single Index

Definition 4 (R-assertion and AER-assertion) *An **R-assertion on a single index** is constructed according to the following syntactical definition (for simplicity we will refer to it as an R-assertion):*

- *An index term is an index variable i_j. The set of index variables usually includes the special variables 1 and N.*
- *An atomic formula is a boolean variable x_k, a boolean term $a[i]$, where a is a boolean array and i is an index term, or a comparison $t_1 < t_2$ where t_1 and t_2 are index terms.*
- *A boolean combination of atomic formulas is a **boolean R-assertion**.*
- *A boolean R-assertion is an R-assertion.*
- *Quantifying an R-assertion over index variables results in an R-assertion.*

*An R-assertion of the form $\forall\vec{i}\,\exists\vec{j} : \psi$ where ψ is a boolean R-assertion is called an AER-**assertion** (where AE is used to symbolize the quantifiers $\forall\exists$ and also indicates the order of quantification). An AER-assertion is said to be **index-closed** if it does not contain any free index variables, except possibly 1 and N.*

Corollary 5 *R-assertions are closed under boolean operations and quantifications over index variables. That is, the resulting assertion can be transformed into the standard form of an R-assertion with universal and existential quantification of a boolean R-assertion over index variables.*

Due to this corollary the R-assertions through this chapter do not necessary assume their standard syntactic forms, that is, we admit R-assertions in the form of $\varphi \wedge \psi$ and $\neg\psi$ where φ and ψ are general R-assertions.

Definition 6 *Let φ be an R-assertion. A **model** $M(n)$ over the vocabulary of φ is an interpretation of the assertion where:*

- *Each boolean variable x_j is assigned a boolean value.*
- *Each index variable i_j is assigned a value from $[1..n]$. The special variables 1 and N are assigned the values 1 and n, respectively.*
- *Each array variable a_j is assigned a boolean array of size n.*

In general, we don't assign values to index variables which appear quantified in φ. For example, a model $M(3)$ for $\varphi = \forall i : \exists j \geq i : y[j] \neq y[i]$ is:

$$N = 3,\ y = (0, 1, 1)$$

Given an R-assertion φ and a model $M(n)$, we can evaluate φ over $M(n)$ and find out whether $M(n)$ satisfies φ. An R-assertion φ is said to be *valid* if it is satisfied by every model.

Definition 7 *Let $M(n)$ be a model over the range $[1..n]$, and $a \in \{1, \ldots, n\}$. We denote by $M(n)[i \mapsto a]$ a **variant model** which interprets the index variable i as a, and interprets all other constructs in the same way as $M(n)$.*

The relation $M(n) \models \varphi$ denotes that model $M(n)$ satisfies assertion φ. And it has the following properties:

1. $M(n) \models f \wedge g \iff M(n) \models f$ and $M(n) \models g$.
2. $M(n) \models \neg f \iff$ It is not the case that $M(n) \models f$.
3. $M(n) \models \forall i : \psi(i) \iff M(n)[i \mapsto a] \models \psi(i)$ for all $a \in [1..n]$.
4. $M(n) \models \exists i : \psi(i) \iff M(n)[i \mapsto a] \models \psi(i)$ for some $a \in [1..n]$.

Theorem 8 (Small Model Theorem for AER-assertion)
Let $\varphi = \forall i_1, \ldots, i_k \exists j_1, \ldots, j_m : \psi$ be an index-closed AER-assertion. φ is valid if and only if it is satisfied by all models $M(n)$ for $n \leq k + 2$.

Proof: It is sufficient to show that if the assertion φ is satisfiable by all models $M(n)$ for $n \leq k+2$ then it is valid. We can establish that by showing that if the negation $\chi\ (= \neg\varphi) = \exists i_1, \ldots, i_k \forall j_1, \ldots, j_m : \neg\psi$ has a satisfying model $M(n)$ of size $n > k+2$ then it also has a satisfying model of size $k+2$. By induction we only need to prove that every satisfying model of size $n > k+2$ can be reduced to a satisfying model of size $n-1$.

Let $M(n)$ be a satisfying model (for χ) of size $n > k+2$. According to property (4) of the model relation we can derive a variant model $M(n)[i_1 \mapsto a_1, \ldots, i_k \mapsto a_k]$ which assigns to $1, i_1, \ldots, i_k, N$ values ranging in the set $1..n$ and satisfies $\forall j_1, \ldots, j_m : \neg\psi$ where $i_1 = a_1, \ldots, i_k = a_k$. This set of values can contain at most $k+2 (< n)$ distinct values. Thus there exists a value $r \in [1..n]$ which doesn't belong to this set. Now let us construct a new model $M'(n-1)$ for χ as follows:

(We write $M[t]$ to denote the interpretation of term t in model M.)

- $M'[x] = M[x]$ for each boolean variable x.
- $M'[N] = n-1$
- For each index term t, $M'[t] =$ if $(M[t] < r)$ then $M[t]$ else $M[t]-1$.
- For each boolean array a, $M'[a] = \lambda u :$ if $u < r$ then $M[a][u]$ else $M[a][u+1]$.

Now we show that if the original model satisfies a boolean R-assertion then the reduced model satisfies it too. It is an important step towards establishing the small model theorem.

Lemma 9 (model reduction for a boolean R-assertion)
Given a boolean R-assertion $\psi(i_1, \ldots, i_k, j_1, \ldots, j_m)$ with index terms $i_1, \ldots, i_k$ and $j_1, \ldots, j_m$, and a satisfying model $M(n)$ for ψ such that $M[i_1], \ldots, M[i_k]$, $M[j_1], \ldots, M[j_m] \in [1..n] \backslash \{r\}$, we claim that the constructed reduced model $M'(n-1)$ satisfies ψ as well.

Proof: To prove the lemma it is sufficient to show that

$$M(n)[i_1 \mapsto M[i_1], \ldots, i_k \mapsto M[i_k], j_1 \mapsto M[j_1], \ldots, j_m \mapsto M[j_m]] \models \psi(i_1, \ldots, i_k, j_1, \ldots, j_m)$$

$$\Longleftrightarrow$$

$$M'(n-1)[i_1 \mapsto M'[i_1], \ldots, i_k \mapsto M'[i_k], j_1 \mapsto M'[j_1], \ldots, j_m \mapsto M'[j_m]] \models \psi(i_1, \ldots, i_k, j_1, \ldots, j_m)$$

Since ψ is a boolean R-assertion it is made of boolean combinations of atomic formulas. The proof is based on structural induction. We start by examining the effect of model reduction on all possible cases of atomic formulas.

- For a boolean variable x, $M'[x] = M[x]$.

- For an atomic formula $t_i < t_j$ where t_i, t_j are index terms and $M[t_i], M[t_j] \neq r$,

$$\begin{aligned} M'[t_i] < M'[t_j] = \; & (M[t_i] < r \wedge M[t_j] < r) \wedge (M[t_i] < M[t_j]) \vee \\ & (M[t_i] < r < M[t_j]) \wedge (M[t_i] < M[t_j] - 1) \vee \\ & (M[t_i] > r \wedge M[t_j] > r) \wedge (M[t_i] - 1 < M[t_j] - 1) \\ = \; & M[t_i] < M[t_j] \end{aligned}$$

- For a boolean array variable $a[t_i]$,

$$\begin{aligned} M'[a][M'[t_i]] = \; & M[a][M'[t_i]] \wedge (M'[t_i] < r) \vee \\ & M[a][M'[t_i] + 1] \wedge (M'[t_i] \geq r) \\ = \; & M[a][M[t_i]] \wedge (M[t_i] < r) \vee \\ & M[a][M[t_i] - 1 + 1] \wedge (M[t_i] > r) \quad (M[t_i] \neq r) \\ = \; & M[a][M[t_i]] \end{aligned}$$

This shows that the model reduction preserves the truth of all atomic formulas. Using the properties of model relations, in particular, $M(n) \models f \wedge g \Leftrightarrow (M(n) \models f) \wedge (M(n) \models g)$ and $M(n) \models \neg f \Leftrightarrow \neg(M(n) \models f)$, it is trivial to show that the model reduction preserves the truth of the boolean R-assertion ψ.

That is $M(n) \models \psi(i_1, \ldots, i_m, j_1, \ldots, j_m) \Longleftrightarrow M'(n-1) \models \psi(i_1, \ldots, i_m, j_1, \ldots, j_m)$ where $M[i_1], \ldots, M[i_k], M[j_1], \ldots, M[j_m] \in [1..n] \backslash \{r\}$. ⌟

Now we can prove that $M(n) \models \chi \Longrightarrow M'(n-1) \models \chi$ by applying the properties of the model relation and Lemma 9,

$$
\begin{array}{ll}
M(n) & \models \chi \\
\Longrightarrow & M(n) \models \exists i_1, \ldots, i_k \in [1..n]\ \forall j_1, \ldots, j_m \in [1..n] : \neg\psi(i_1, \ldots, i_k, j_1, \ldots, j_m) \\
\overset{\text{Def. 7}}{\Longrightarrow} & M(n)[i_1 \mapsto a_1, \ldots, i_k \mapsto a_k] \models \forall j_1, \ldots, j_m : \neg\psi(i_1, \ldots, i_k, j_1, \ldots, j_m) \\
 & \qquad \text{for some } a_1, \ldots, a_k \in [1..n] \backslash \{r\} \\
\Longrightarrow & \forall b_1, \ldots, b_m \in [1..n] : \\
 & \quad M(n)[i_1 \mapsto a_1, \ldots, i_k \mapsto a_k, j_1 \mapsto b_1, \ldots, j_m \mapsto b_m] \models \\
 & \qquad \neg\psi(i_1, \ldots, i_k, j_1, \ldots, j_m) \\
\Longrightarrow & \forall b_1, \ldots, b_m \in [1..n] \backslash \{r\} : \\
 & \quad M(n)[i_1 \mapsto a_1, \ldots, i_k \mapsto a_k, j_1 \mapsto b_1, \ldots, j_m \mapsto b_m] \models \\
 & \qquad \neg\psi(i_1, \ldots, i_k, j_1, \ldots, j_m) \\
\overset{\text{Lemma 9}}{\Longleftrightarrow} & \forall b'_1, \ldots, b'_m \in [1..n-1] : \\
 & \quad M'(n-1)[i_1 \mapsto a'_1, \ldots, i_k \mapsto a'_k, j_1 \mapsto b'_1, \ldots, j_m \mapsto b'_m] \models \\
 & \qquad \neg\psi(i_1, \ldots, i_k, j_1, \ldots, j_m) \\
 & \qquad \text{where } a'_1 = M'[i_1], \ldots, a'_k = M'[i_k] \\
 & \qquad \text{and } b'_1 = M'[j_1], \ldots, b'_m = M'[j_m] \\
\Longrightarrow & M'(n-1)[i_1 \mapsto a'_1, \ldots, i_k \mapsto a'_k] \models \\
 & \quad \forall j_1, \ldots, j_m \in [1..n-1] : \neg\psi(i_1, \ldots, i_k, j_1, \ldots, j_m) \\
 & \qquad \text{where } a'_1, \ldots, a'_m \in [1..n-1] \\
\overset{\text{Def. 7}}{\Longrightarrow} & M'(n-1) \models \exists i_1, \ldots, i_k \forall j_1, \ldots, j_m : \neg\psi(i_1, \ldots, i_k, j_1, \ldots, j_m) \\
\Longrightarrow & M'(n-1) \models \chi
\end{array}
$$

⌟

Corollary 10 *Let $\varphi = (\exists i_1, \ldots, i_k \forall j_1, \ldots, j_m : \psi) \longrightarrow (\forall h_1, \ldots, h_q : \zeta)$ be an index-closed R-assertion with boolean R-assertions ψ and ζ. We have that φ is valid if and only if it is satisfied by all models $M(n)$ for $n \leq k + q + 2$.*

The proof can be obtained by applying the same method of model reduction for a model

$M(n)$ where $n > k + q + 2$ for the negated formula $\neg\varphi$.

3.1.2 Small Model Theorem for AER-Assertions on Multiple Types of Indices

Now we would like to allow multiple finite domain data types for indices. The main complication comes from introducing array types u^{ij} : **type**$_i \mapsto$ **type**$_j$.

Definition 11 *Suppose that we have m types of indices, where* **type**$_i$: $[1..N_i], (1 \leq i \leq m)$*, for $N_i \in \mathcal{N}^+$. An R-**assertion on** m **types of indices** satisfies the following semantics:*

- *An index term of type* **type**$_j$ *is an index variable i_k^j :* **type**$_j$ *(usually with the special terms 1 and N_j), or an index term $u^{ij}[k]$ where u is an array that maps from* **type**$_i$ *to* **type**$_j$ *and k is an index term of* **type**$_i$*.*
- *An atomic formula is a boolean variable x_k, or a boolean term $a[i]$, where a is a boolean array and i is an index term, or a comparison $t_1 < t_2$ where t_1 and t_2 are index terms of same type, or one of them is the constant 1.*
- *A boolean combination of atomic formulas is a **boolean** R-**assertion**.*
- *A boolean R-assertion is an R-assertion.*
- *Quantifying an R-assertion over index variables forms an R-assertion.*

*An R-assertion of the form $\forall\vec{i_1}\ldots\vec{i_k}\exists\vec{j_1},\ldots,\vec{j_m}$: ψ where ψ is a boolean R-assertion is called an AER-**assertion**. An AER-assertion is said to be **index-closed** if it does not contain any free index variables.*

Definition 12 *Let φ be an R-assertion on m types of indices. A **model** $M(n_1,\ldots,n_m)$ over the vocabulary of φ is an interpretation of the assertion where:*

- *Each boolean variable x_j is assigned a boolean value.*
- *Each index variable i_j of type* **type**$_k$: $[1..N_k]$ *is assigned a value from $[1..n_k]$. The constants 1 and N_k are assigned the values 1 and n_k.*

– *Each boolean array variable* a_j *which maps from* $\mathbf{type}_k : [1..N_k]$ *to the boolean domain is assigned a boolean array of size* n_k*.*

– *Each index array variable* $b_j : \mathbf{type}_i \mapsto \mathbf{type}_j$ *is assigned an array of type* $[1..n_i] \mapsto [1..n_j]$*.*

In order to develop the small model theorem for AER-assertions on multiple types of indices we need to introduce the notion of a *stratified R-assertion.*

Definition 13 *An R-assertion on multiple types of indices* $\mathbf{type}_1, \ldots, \mathbf{type}_m$ *is called a* ***stratified*** R***-assertion*** *if every non-boolean array of the type* $\mathbf{type}_i \mapsto \mathbf{type}_j$ *satisfies* $i < j$*.*

Theorem 14 (small model theorem on stratified AER-assertions)
Let $\varphi = \forall \vec{i^1}, \ldots \vec{i^m} \exists \vec{j^1}, \ldots \vec{j^t} : \psi$ *be an index-closed stratified AER-assertion. φ is valid if and only if it is satisfied by all models $M(n_1, \ldots, n_m)$ for $n_1 \leq n_1^0, \ldots, n_m \leq n_m^0$ where $n_1^0 = |\vec{i^1}| + 2$, and for $k = 2, \ldots, m$, $n_k^0 = (|\vec{i^k}| + 2) + \sum_{j=1}^{k-1}(e_{jk} \cdot n_j^0)$ where e_{jk} represents the number of arrays of type* $\mathbf{type}_j \mapsto \mathbf{type}_k$*.*

Proof: We start by performing model reduction on $\mathbf{type}_1$ indices. We show that we can reduce a satisfying model $M(n_1, n_2, \ldots, n_m)$ of size $n_1 > |\vec{i^1}| + 2$ for the negated AER-assertion $\neg\varphi$ to a satisfying model $M(n_1 - 1, n_2, \ldots, n_m)$.

In addition to the model reduction that is used for proving the small model theorem on a single index (Theorem 8) we also need to define the reduction for an index array of $\mathbf{type}_1 \mapsto \mathbf{type}_k$ for $k > 1$:

– For an index array $u^{1k} : \mathbf{type}_1 \mapsto \mathbf{type}_k$,
$M'[u^{1k}] = \lambda z :$ if $z < r$ then $M[u^{1k}][z]$ else $M[u^{1k}][z+1]$

where $r \in [1..n_1]$ is an uninterpreted value for the $\mathbf{type}_1$ vector $\vec{i^1}$. We can proceed to show that this model reduction preserves the truth of a boolean R-assertion using a proof similar to that of Lemma 9. The additional atomic formulas we need to consider here are the comparison of $\mathbf{type}_k$ ($k > 1$) index terms where a $\mathbf{type}_1 \mapsto \mathbf{type}_k$ array term is involved.

– For an atomic formula $u^{1k}[z] < t$ where t is a **type**$_k$ index variable and z is a **type**$_1$ index term.

$$
\begin{aligned}
& M'[u^{1k}][M'[z]] < M'[t] \\
= \;& (M[u^{1k}][M'[z]] < M[t]) \wedge (M'[z] < r) \vee \\
& (M[u^{1k}][M'[z]+1] < M[t]) \wedge (M'[z] \geq r) \\
= \;& (M[u^{1k}][M[z]] < M[t]) \wedge (M[z] < r) \vee \\
& (M[u^{1k}][M[z]-1+1] < M[t]) \wedge (M[z] > r) \\
= \;& M[u^{1k}][M[z]] < M[t]
\end{aligned}
$$

– For an atomic formula $u^{1k}[z_1] < v^{1k}[z_2]$ where z_1, z_2 are **type**$_1$ index terms.

$$
\begin{aligned}
& M'[u^{1k}][M'[z_1]] < M'[v^{1k}][M'[z_2]] \\
= \;& (M[u^{1k}][M'[z_1]] < M[v^{1k}][M'[z_2]]) \wedge (M'[z_1] < r) \wedge (M'[z_2] < r) \vee \\
& (M[u^{1k}][M'[z_1]+1] < M[v^{1k}][M'[z_2]]) \wedge (M'[z_1] \geq r) \wedge (M'[z_2] < r) \vee \\
& (M[u^{1k}][M'[z_1]] < M[v^{1k}][M'[z_2]+1]) \wedge (M'[z_1] < r) \wedge (M'[z_2] \geq r) \vee \\
& (M[u^{1k}][M'[z_1]+1] < M[v^{1k}][M'[z_2]+1]) \wedge (M'[z_1] \geq r) \wedge (M'[z_2] \geq r) \\
= \;& (M[u^{1k}][M[z_1]] < M[v^{1k}][M[z_2]]) \wedge (M[z_1] < r) \wedge (M[z_2] < r) \vee \\
& (M[u^{1k}][M[z_1]-1+1] < M[v^{1k}][M[z_2]]) \wedge (M[z_1] > r) \wedge (M[z_2] < r) \vee \\
& (M[u^{1k}][M[z_1]] < M[v^{1k}][M[z_2]-1+1]) \wedge (M[z_1] < r) \wedge (M[z_2] > r) \vee \\
& (M[u^{1k}][M[z_1]-1+1] < M[v^{1k}][M[z_2]-1+1]) \wedge (M[z_1] > r) \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge (M[z_2] > r) \\
= \;& M[u^{1k}][M[z_1]] < M[v^{1k}][M[z_2]]
\end{aligned}
$$

After having proven that the reduced model satisfies a boolean R-assertion, and by using the properties of the model relation, we can deliver the same proof (as in Theorem 8) that the reduced model $M'(n_1 - 1, n_2, \ldots, n_m)$ satisfies the R-assertion $\neg\varphi$.

Therefore we can keep reducing the model on **type**$_1$ indices until we reach the cutoff value $n_1^0 = |\vec{i^1}|$, where, we obtain the model $M(n_1^0, n_2, \ldots, n_m)$ that satisfies the negated AER-assertion $\neg\varphi$.

Next we perform the model reduction on **type**$_2$ indices. We can treat **type**$_2$ the same way as we treat **type**$_1$. The only difference is that the possible interpretations we need to

provide for **type**$_2$ index terms include not only the vector $\vec{i^2}$ but all the **type**$_1 \mapsto$ **type**$_2$ index terms as well. The total number can reach $|\vec{i^2}| + e_{12} \cdot n_1^0$, which is the cutoff value for **type**$_2$ indices. Subsequently we can derive the cutoff values for **type**$_3, \ldots,$ **type**$_m$, which turn out to be $n_k^0 = (|\vec{i^k}| + 2) + \sum_{j=1}^{k-1}(e_{jk} \cdot n_j^0)$ where e_{jk} represents the number of arrays of type **type**$_j \mapsto$ **type**$_k$ for $k = 3, \ldots, m$. ▟

3.1.3 Bounded Data Systems

Previously when introducing the definition (Def. 3) of the small model property we pointed out that the calculation of the cutoff value for a parameterized system depends on a few critical factors, i.e., the system characteristics, the type of property we want to verify and the verification methodology. Here our goal is to verify invariance properties of parameterized systems using deductive methods, therefore we want to apply small model properties to help discharge the premises of the deductive inference rules, in particular rule INV in this case. We will define a special yet important class of parameterized systems called *Bounded Data Systems* (BDS) for which we can derive small model properties based on the data type signatures of BDS systems. The definition of a BDS system lays out the necessary conditions to ensure that the premises of the inference rules are in the categories of R-assertions with well-established small model theorems.

Definition 15 *A **bounded-data system (**BDS**)** $S = \langle V, \Theta, \rho \rangle$ consists of*

- *scalar data types* **type**$_0, \ldots,$ **type**$_m$ *– where* **type**$_0$ *(often denoted as* **bool***) is the set of boolean or finite-range scalars, and* **type**$_1, \ldots,$ **type**$_m$ *are a set of scalar data types where each* **type**$_i$ *includes integers in the range* $[1..N_i]$ *for some* $N_i > 1$*. We refer to* $N_1, \ldots, N_m$ *as system parameters.*

- V*: a set of system variables of the following types,*

  $x_1, \ldots, x_a$: **bool**
  $y_1^i, \ldots, y_{b_i}^i$: **type**$_i$
  $u_1^i, \ldots, u_{c_i}^i$: **array type**$_i$ **of bool**
  $w_1^{ij}, \ldots, w_{e_{ij}}^{ij}$: **array type**$_i$ **of type**$_j$ $(1 \leq i, j \leq m)$

A state of the system S provides a type-consistent interpretation of the system variables V. For a state s and a system variable $v \in V$, we denote by $s[v]$ the value assigned to v by the state s. The set of states over V is denoted by Σ.

– $\Theta(V)$*: the initial condition, whose characteristic function can be expressed as a negated AER-assertion on* $\mathbf{type}_1, \ldots, \mathbf{type}_m$*,*

$$\exists \underbrace{i_1^1, \ldots, i_{K_1}^1}_{\mathbf{type}_1} , \ldots, \underbrace{i_1^m, \ldots, i_{K_m}^m}_{\mathbf{type}_m} \forall \vec{j^1}, \ldots, \vec{j^m} : \theta(\vec{i^1}, \ldots, \vec{i^m}, \vec{j^1}, \ldots, \vec{j^m})$$

– $\rho(V, V')$*: the transition relation relates the values V of the variables in a state $s \in \Sigma$ to the values V' in an S-successor state $s' \in \Sigma$. In particular it can be written in the form of a negated AER-assertion:*

$$\exists \underbrace{h_1^1, \ldots, h_{H_1}^1}_{\mathbf{type}_1} , \ldots, \underbrace{h_1^m, \ldots, h_{H_m}^m}_{\mathbf{type}_m} \forall \vec{t^1}, \ldots, \vec{t^m} : R(\vec{h^1}, \ldots, \vec{h^m}, \vec{t^1}, \ldots, \vec{t^m})$$

Without loss of generality we assume that $K_1 \leq H_1, \ldots, K_m \leq H_m$. And we refer to $H_1, \ldots, H_m$ as the ***type measures*** *of the* BDS *system.*

In summary the class of BDS systems is a special class of FDS systems with restricted data types and special logical forms for initial conditions and translation relations, and no fairness requirements. We don't consider the fairness requirements here because our primary focus is to prove invariance properties. When extending this methodology to prove liveness properties we need to add fairness components to construct a fair BDS system [FPPZ04].

BDS systems are distinguished by their *signatures*, which determine the types of variables allowed, as well as the assertions allowed in the transition relation and initial condition. Whenever the signature of a system includes the type $\mathbf{type}_i \mapsto \mathbf{type}_j$, we assume by default that it also includes the types $\mathbf{type}_i$ and $\mathbf{type}_j$.

3.1.4 Small Model Theorem for BDS systems with Signature $\langle \mathbf{type}_1 \mapsto \mathbf{bool} \rangle$

This class of BDS systems allows the following data types:

- Boolean and other finite domain variables (note the finite domain variables can be easily encoded as a set of boolean variables).
- The $\mathbf{type}_1$ variables whose domains are $[1..N]$.
- Parameterized arrays which map from $[1..N]$ to a boolean or to a finite domain.

Recall the rule INV for proving invariance property $\Box\,\varphi$ of BDS systems:

With parameter N, and given assertions φ_N, η_N

I1. $\Theta_N \rightarrow \eta_N$

I2. $\eta_N \wedge \rho_N \rightarrow \eta'_N$

I3. $\eta_N \rightarrow \varphi_N$

$\Box\,\varphi_N$

For BDS systems with signature $\langle \mathbf{type}_1 \mapsto \mathbf{bool} \rangle$ their initial conditions and transition relations assume the form of negated AER-assertions, therefore we can calculate the cutoff values using Corollary 10 to help discharge premises I1 and I2. Without loss of generality we assume the common case where I2 is the most complicated formula to discharge and has the highest cutoff value.

Theorem 16 *Let $S(N)$ be a parameterized* BDS *system of signature $\langle \mathbf{type}_1 \mapsto \mathbf{bool} \rangle$ with type measure H and $\mathbf{type}_1$ variables $y_1, \ldots, y_b$. Let $\psi = \Box\,\varphi$ where $\varphi = \forall \vec{i} : q(\vec{i})$ with $|\vec{i}| = I$ be an invariance property to be verified and let $\eta = \forall \vec{i} : p(\vec{i})$ with $|\vec{i}| = I$ be an auxiliary assertion in rule* INV. *Then the premises of rule* INV *are valid over $S(N)$ for all $N > 1$ if and only if they are valid over $S(N)$ for all $N, 1 < N \leq N_0$, where $N_0 = 2b + I + H + 2$ is the cutoff value.*

Proof: We only need to prove that if rule INV is proven valid for $S(N)$ (for $1 < N \leq N_0$) then it is valid for all $N > 1$. From now on we adopt the notation $i_H^{\vec{k}}$ to mean a **type**$_k$ vector of size H. Given $\eta = \forall \vec{i_I} : p(\vec{i})$ by rule INV we have the following inference rule:

$$\begin{array}{ll} \text{I1.} & (\exists \vec{i}_K \forall \vec{j} : \theta(\vec{i}, \vec{j})) \rightarrow \forall \vec{i_I} : p(\vec{i}) \ \text{ (where } K \leq H) \\ \text{I2.} & (\forall \vec{i_I} : \ p(\vec{i})) \wedge (\exists \vec{h}_H \forall \vec{t}_T : R(\vec{h}, \vec{t})) \rightarrow \forall \vec{i_I} : p'(\vec{i}) \\ \text{I3.} & (\forall \vec{i_I} : p(\vec{i})) \rightarrow \forall \vec{i_I} : q(\vec{i}) \\ \hline & \Box \, \varphi \end{array}$$

Since I2 is not a index-closed formula we may need to interpret all the **type**$_1$ index variables $y_1, \ldots, y_b$ as well as their primed version. By Corollary 10 we have that I2 is valid if and only if it is satisfied by all models $M(n)$ for $n \leq 2b + 2 + (I + H)$, that is the cutoff value to discharge I2 is $2b + I + H + 2$. Similarly we can obtain the cutoff values to discharge I1 and I3 to be $2b + I + K + 2$. Since we assume that $K < H$, therefore we choose the maximum value $n \leq 2b + I + H + 2$ to be the cutoff value to discharge all the premises I1- I3. ∎

Definition 17 *A bounded-data system is said to be* ***par-deterministic*** *if for any* **type**$_i$ *index variable* y*, all atomic formulas involving its primed version* y' *is of the form* $y' = u$ *for some unprimed* **type**$_i$ *index term* u*.*

For a par-deterministic BDS system the cutoff value can be tightened to $N_0 = b+I+H+2$, which turns out to be the case for most common parameterized systems. Furthermore if we know the constant index terms $1, N$ are not explicitly referred to in the assertions, then we can further tighten the cutoff value to $N_0 = b + I + H$.

3.1.5 Small Model Theorem for General Stratified BDS systems

General BDS systems allow array types **type**$_i \mapsto$ **type**$_j$ and as a consequence have R-assertions on multiple types of indices in the deductive inference rules. To apply small model theorems on such systems we need to define the category of stratified BDS systems.

Definition 18 *Consider a* BDS *system with k parameterized types* $\mathbf{type}_1, \ldots, \mathbf{type}_k$ *(as always* $\mathbf{type}_0$ *is reserved for* **bool***). This system is called* ***stratified*** *if for each array type in the system* $\mathbf{type}_i \mapsto \mathbf{type}_j$ *$(i, j \neq 0)$ we have $i < j$.*

The class of stratified BDS systems allows the following data types:

- Boolean and other finite domain variables (note the finite domain variables can be easily encoded as a set of boolean variables).
- Variables of $\mathbf{type}_i$. Let b_i be the number of $\mathbf{type}_i$ variables in the system.
- Parameterized arrays which map from $\mathbf{type}_i$ to boolean or finite domain.
- Parameterized arrays which map from $\mathbf{type}_i$ to $\mathbf{type}_j (0 < i < j)$. Let e_{ij} be the number of $\mathbf{type}_i \mapsto \mathbf{type}_j$ arrays in the system.

Theorem 19 (small model theorem for stratified BDS **systems)**
Let $S(N_1, \ldots, N_k)$ be a stratified BDS *system with k types and type measures $H_1, \ldots, H_k$ $(k \geq 1)$, to which we wish to apply the proof rule* INV *with the assertions φ and η each having the form $\forall i^{\vec{1}}_{I_1}, \ldots, i^{\vec{k}}_{I_k} : p(\vec{i_1}, \ldots, \vec{i_k})$. Also assume the number of* $\mathbf{type}_i$ *index variables in $S(N_1, \ldots, N_k)$ is b_i. Then the premises of rule* INV *are valid over $S(N_1, \ldots, N_k)$ for all $N_1, \ldots, N_k > 1$ iff they are valid over $S(N_1, \ldots, N_k)$ for $N \leq N_1^0, \ldots, N \leq N_k^0$ where $N_1^0 = 2b_1 + H_1 + I_1 + 2$, and for every $i = 2, \ldots, k$, $N_i^0 = (2b_i + H_i + I_i + 2) + \Sigma_{j=1}^{i-1}(2e_{ji} \cdot N_j^0)$ where e_{ji} represents the number of arrays of type* $\mathbf{type}_j \mapsto \mathbf{type}_i$.

Proof: Similar to the arguments used to prove the small model theorem for a BDS system with a single index type, we can apply the small model theorem for AER-assertions on multiple types of indices on I2 (and I1, I3) to obtain the maximum of the cutoff values. ∎

For a par-deterministic BDS system the cutoff value can be tightened to $N_1^0 = b_1 + H_1 + I_1 + 2$, and for every $i = 2, \ldots, k$, $N_i^0 = (b_i + H_i + I_i + 2) + \Sigma_{j=1}^{i-1}(e_{ji} \cdot N_j^0)$.

In particular the small model theorem for BDS systems with signature $\langle \mathbf{type}_1 \mapsto \mathbf{bool} \rangle$ is a special case of the small model theorem for general stratified BDS systems. All such

theorems are sound but incomplete. The incompleteness comes from the fact that we do not always succeed in finding an inductive auxiliary assertion of a desired special logical form. In the later part of this chapter we will devote our discussion to the issue of automatic generation of auxiliary assertions used in deductive verification.

3.1.6 Small Model Approach for Unstratified BDS Systems

After the success of the small model theorem with stratified BDS systems, one may wonder why the theorem failed to work for unstratified BDS systems. The answer to the question might provide insights to the core of the problem and lead us to a possible partial remedy.

In the derivation of small model theorem for stratified BDS systems (as well as stratified AER-assertions) our methodology relies on the partial order indicated by the stratified array structure. We first derive the cutoff value for $\mathbf{type}_1$ variables, then we use that to derive the cutoff value for $\mathbf{type}_2$ variables and so on till we reach the cutoff value for $\mathbf{type}_m$ variables. Furthermore, if there exists an array type $\mathbf{type}_i \mapsto \mathbf{type}_j$ then we know that the cutoff value for $\mathbf{type}_i$ is less than the cutoff value for $\mathbf{type}_j$, i.e. $N_i < N_j$ except for the special case where there is no $\mathbf{type}_j$ variables then we might have $N_i \leq N_j$. The ascending order of array type indices used in the definition of stratified systems is only used to simplify our reasoning about stratified structures. In general the necessary requirement for stratified array structure calls for a topological order among types such that there exists no cycle in the array structure. One can always rename the types to satisfy the additional ascending order requirement.

However for an unstratified BDS system one can always find a cycle in the array structure such as $\mathbf{type}_{k_1} \mapsto \mathbf{type}_{k_2}, \mathbf{type}_{k_2} \mapsto \mathbf{type}_{k_3}, \ldots, \mathbf{type}_{k_m} \mapsto \mathbf{type}_{k_1}$. If we apply the same method to derive the cutoff values then often we have the cutoff values $N_{k_1} < N_{k_2} < \ldots < N_{k_m} < N_{k_1}$ which leads to a contradiction.

Therefore in theory we do not have a straight-forward generalization of the small model

property from stratified BDS systems to general BDS systems with unstratified array structure. However this does not imply that we cannot find a sound but restricted small model approach for unstratified BDS systems. We studied the verification of the safety property of Peterson's mutual exclusion algorithm with simple unstratified array structures, and defined a small model approach for deductive verification of unstratified BDS system. In the following sections we first present this small model approach and then illustrate its application to Peterson's mutual exclusion algorithm.

Small Model Theorem via Instantiation

Our main focus here is to search for a small model theorem to discharge the premises I1– I3 in rule INV for general BDS systems, in particular, unstratified BDS system. As stated previously the cutoff value usually comes from discharging the most complicated verification condition I2 of the following general form:

$$\text{I2}: \qquad (\forall \vec{j} : \psi(\vec{j})) \wedge (\exists \vec{h}\, \forall \vec{t} : R(\vec{h}, \vec{t})) \longrightarrow \forall \vec{i} : \psi'(\vec{i})$$

First we apply skolemization and remove all the existential quantifications on the left-hand side of the implication and all the universal quantification on the right-hand side of the implication leading to an equivalent statement:

$$(\forall \vec{j} : \psi(\vec{j})) \wedge (\forall \vec{t} : R(\vec{h_0}, \vec{t})) \longrightarrow \psi'(\vec{i_0})$$

for some constant vector $\vec{h_0}, \vec{i_0}$.

A deductive proof normally instantiates the remaining universal quantifications for $\vec{j}$ and $\vec{t}$ by concrete terms. Most often the concrete terms are taken from the (now) free variables, namely, $\vec{h_0}$ and $\vec{i_0}$. So we suggest replacing the universal quantification over $\vec{j}$ and $\vec{t}$ by a conjunction in which each conjunct is obtained by instantiating the relevant variables ($\vec{j}$ or $\vec{t}$) by a subset of the free variables $\vec{h_0}$ and $\vec{i_0}$. The conjunction should be taken over all such possible instantiations. The resulting quantifier-free formula is not equivalent

to the original formula but the validity of the new one implies the validity of the original one.

$$\text{I2}^*: \qquad (\bigwedge_{\vec{j}\in\{\vec{h_0},\vec{i_0}\}} \psi(\vec{j})) \wedge (\bigwedge_{\vec{t}\in\{\vec{h_0},\vec{i_0}\}} R(\vec{h_0},\vec{t})) \longrightarrow \psi'(\vec{i_0})$$

We have transformed our original proof obligation I2 into a new quantifier-free formula I2* via partial instantiation. For a quantifier-free formula, we have again the property of model reduction, which can be used to formulate the appropriate decision procedure for an unstratified BDS system. If the quantifier-free formula is true for all the possible instantiations of $\vec{h_0}$ and $\vec{i_0}$ in the small models then the verification condition I2* is true and in turn implies that I2 is true.

We can apply similar partial instantiation to transform I1 and I3 into quantifier-free formulas. And we denote the newly obtained inference rule as rule INV*, where we obtain a stronger set of (quantifier-free) premises than the premises in rule INV. Next we are going to establish a small model theorem for quantifier-free R-assertions.

Definition 20 *An R-assertion is a **simple R-assertion** if we restrict our definition of the index term to allow only $u^{ij}[k]$ where u is an array which maps from $[1..N_i]$ to $[1..N_j]$ and k is an index variable (that is we don't allow k to be an arbitrary index term thus excluding forms like $u^{ij}[u^{hi}[k]]$).*

This definition does not change the expressive power of R-assertions because we can convert the term $u^{ij}[u^{li}[k]]$ to $u^{ij}[d] \wedge d = u^{li}[k]$ which abides by the syntactic forms required of a simple R-assertion.

Although general BDS systems with unstratified array structures allow self-indexing array types, i.e. $\textbf{type}_k \mapsto \textbf{type}_k$, however for the simplicity of our presentation we disallow such array types, and replace it with two array types $\textbf{type}_k \mapsto \textbf{type}_m$ and $\textbf{type}_m \mapsto \textbf{type}_k$ by introducing a "cloned" type $\textbf{type}_m$.

Lemma 21 *Let $\psi_1, \ldots, \psi_m$ be boolean simple R-assertions on m types of indices. Let $\varphi = ß_{t^1_{H_1},\ldots,t^m_{H_m}}(\psi_1, \ldots, \psi_m)$ be a quantifier-free simple R-assertion obtained from $\psi_1, \ldots, \psi_m$*

using boolean connectives with all **type**$_i$ *variables given values from* $\vec{t}^{\,i}_{H_i}$*, a vector of size* H_i *consisted of free* **type**$_i$ *variables. Let* e_{ji} *be the number of arrays of type* **type**$_j \mapsto$ **type**$_i$*. Then* φ *is valid iff it is satisfied by all models* $M(n_1, \ldots, n_m)$ *where* $n_i \leq (H_i + 2) + \Sigma_{1 \leq j \leq m \wedge j \neq i}(e_{ji} \cdot (H_j + 2))$ *for* $1 \leq i \leq m$*.*

Proof: It is sufficient show that any model $M(n_1, \ldots, n_k, \ldots, n_m)$ that satisfying the negated assertion $\neg\varphi$ with size n_k ($k \in [1..m]$) greater than the cutoff value can be reduced to a model $M'(n_1, \ldots, n_k - 1, \ldots, n_m)$.

A model M for a quantifier-free simple R-assertion only needs to interpret the free index variables and the array elements indexed by these index variables. In this case such terms of **type**$_k$ include $t^{\vec{k}}_{H_k}$ and $u^{jk}[i]$ where $i \in t^{\vec{k}}_{H_k}$ and possibly the constant terms $1, N_k$. Thus the maximum number of these **type**$_k$ index terms is $H_k + 2 + \Sigma_{1 \leq j \leq m \wedge j \neq k} e_{jk}(H_j + 2)$, where e_{jk} is the number of **type**$_j \mapsto$ **type**$_k$ arrays.

Let $M(n_1, \ldots, n_k, \ldots, n_m)$ be a model of $\neg\varphi$ with $n_k > H_k + 2 + \Sigma_{1 \leq j \leq m \wedge j \neq k} e_{jk} \cdot (H_j + 2)$, then we can find an integer r such that $1 < r < n_k$ and there is no **type**$_k$ index term assigned the value r in model M. Now let us construct a reduced model $M'(n_1, \ldots, n_k - 1, \ldots, n_m)$ which also satisfies the quantifier-free simple R-assertion $\neg\varphi$:

- $M'[x] = M[x]$ for a boolean variable x.
- $M'[t] = M[t]$ for any index variable t whose type is not **type**$_k$.
- $M'[c] = M[c]$ for an array c : **type**$_i \mapsto$ **type**$_j$ where $i, j \neq k$.
- $M'[N_k] = n_k - 1$.
- For a **type**$_k$ index term t, $M'[t] =$ if $(M[t] < r)$ then $M[t]$ else $M[t] - 1$.
- For an array d : **type**$_i \mapsto$ **type**$_k$ where $i \neq k$,
  $M'[d] = \lambda u :$ if $(M[d][u] < r)$ then $M[u][d]$ else $M[u][d] - 1$.
- For a boolean array a : **type**$_k \mapsto$ **bool**,
  $M'[a] = \lambda u :$ if $u < r$ then $M[a][u]$ else $M[a][u + 1]$.

– For an index array b : **type**$_k \mapsto$ **type**$_j$ ($j \neq k$),
$M'[b] = \lambda u :$ if $u < r$ then $M[b][u]$ else $M[b][u+1]$.

Note that unlike stratified R-assertions this model reduction doesn't have to follow the order of an array structure since φ is a simple R-assertion. Now we show that all the atomic formulas are preserved by this model reduction.

– For a boolean variable x, $M'[x] = M[x]$.

– For a boolean array a : **type**$_i \mapsto$ **bool** ($i \neq k$) and an index variable t of **type**$_i$,
$M'[a][M'[t]] = M[a][M[t]]$.

– For a boolean array a : **type**$_k \mapsto$ **bool** and an index variable t : **type**$_k$,

$$\begin{aligned} M'[a][M'[t]] = \; & M[a][M'[t]] \wedge (M'[t] < r) \vee \\ & M[a][M'[t]+1] \wedge (M'[t] \geq r) \\ = \; & M[a][M[t]] \wedge (M[t] < r) \vee \\ & M[a][M[t]-1+1] \wedge (M[t] > r) \\ = \; & M[a][M[t]] \qquad\qquad (M[t] \neq r) \end{aligned}$$

– For an atomic formula $s < t$ where index terms s, t are of **type**$_i$ and $i \neq k$, s, t can be: index variables, or array terms of type **type**$_j \mapsto$ **type**$_i (j \neq k)$, or array terms of type **type**$_k \mapsto$ **type**$_i$.
For the first two cases it is trivial that $M'[t] = M[t]$ and $M'[s] = M[s]$. For the third case suppose $t = c[u]$ where c : **type**$_k \mapsto$ **type**$_i$ and u : **type**$_k$, then we have:

$$\begin{aligned} M'[t] = \; & M'[c][M'[u]] \\ = \; & M[c][M'[u]] \wedge (M'[u] < r) \vee \\ & M[c][M'[u]+1] \wedge (M'[u] \geq r) \\ = \; & M[c][M[u]] \wedge (M[u] < r) \vee \\ & M[c][M[u]-1+1] \wedge (M[u] > r) \\ = \; & M[c][M[u]] \qquad\qquad (M[u] \neq r) \\ = \; & M[t] \end{aligned}$$

In all cases the interpretation of such index terms is unchanged in both models. Therefore $M'[s] < M'[t] = M[s] < M[t]$.

– For an atomic formula $s < t$ where index terms s, t are of **type**$_k$, s, t can be **type**$_k$ index variables, or array terms which maps from **type**$_i$ to **type**$_k$ $(i \neq k)$. In all possible cases we have:

$$\begin{aligned} M'[s] < M'[t] = \; & (M[s] < r \wedge M[t] < r) \wedge (M[s] < M[t]) \vee \\ & (M[s] < r < M[t]) \wedge (M[s] < M[t] - 1) \vee \\ & (M[s] > r \wedge M[t] > r) \wedge (M[s] - 1 < M[t] - 1) \\ = \; & M[s] < M[t] \end{aligned}$$

This concludes the proof of preservations of atomic formulas in a simple R-assertion $\neg\varphi$. By structural induction and the properties of model relation, it implies that $M'(n_1, \ldots, n_k - 1, \ldots, n_m)$ satisfies $\neg\varphi$. Similar reductions can be carried out on other types of indices. ∎

For simplicity and without loss of generality, we assume that for an unstratified BDS system, the initial condition Θ and the transition relation ρ are written as simple R-assertions; each has K_i and $H_i (\geq K_i)$ existentially quantified **type**$_i$ variables respectively.

Theorem 22 (small model theorem for unstratified BDS systems)
Let $S(N_1, \ldots, N_k)$ be an unstratified BDS *system with k index types and type measures $H_1, \ldots, H_k$ ($k \geq 1$), to which we wish to apply the modified proof rule* INV* *with the simple R-assertions φ and η each having the form $\forall i^{\vec{1}}_{I_1}, \ldots, i^{\vec{k}}_{I_k} : \psi(\vec{i_1}, \ldots, \vec{i_k})$. Also assume the number of* **type**$_i$ *index variables in $S(N_1, \ldots, N_k)$ is b_i. Then the premises of rule* INV* *are valid over $S(N_1, \ldots, N_k)$ for all $N_1, \ldots, N_k > 1$ iff they are valid over $S(N_1, \ldots, N_k)$ for $N \leq N^0_1, \ldots, N \leq N^0_k$ where for every $i = 1, \ldots, k$, $N^0_i = (2b_i + H_i + I_i + 2) + \Sigma_{1 \leq j \leq k \wedge j \neq i}(2e_{ji} \cdot (2b_j + H_j + I_j + 2))$ where e_{ji} represents the number of arrays of type* **type**$_j \mapsto$ **type**$_i$.

Proof: Without loss of generality, we assume that the upper bound comes from discharging I2*, the most complicated verification condition in rule INV*. To simplify the notation we use **type**$_j$ variable vectors $\vec{v^j} = \{1, N_j, \vec{y^j}, \vec{h^j_0}, \vec{i^j_0}\}$ with $|\vec{y^j}| = b_j, |\vec{h^j_0}| = H_j, |\vec{i^j_0}| = I_j$ for $j \in [1..k]$. Here is the simple quantifier-free R-assertion for I2*:

$$\bigwedge_{\vec{j_1} \in \vec{v^1}, \ldots, \vec{j_k} \in \vec{v^k}} \psi(\vec{j_1}, \ldots, \vec{j_k}) \wedge \bigwedge_{\vec{t_1} \in \vec{v^1}, \ldots, \vec{t_k} \in \vec{v^k}} R(\vec{h^1_0}, \ldots, \vec{h^k_0}, \vec{t_1}, \ldots, \vec{t_k}) \longrightarrow \psi'(\vec{i^1_0}, \ldots, \vec{i^k_0})$$

Here we might need to interpret the primed version of $\vec{y^j}$ and other array terms. Applying Lemma 21 on I2* we obtain the bound $N_i^0 = (2b_i + H_i + I_i + 2) + \Sigma_{1 \le j \le m \wedge j \neq i}(2e_{ji} \cdot (2b_j + H_j + I_j + 2))$ for $i = 1, \ldots, k$. ∎

For a par-deterministic BDS system we can tighten the bound to $N_i^0 = (b_i + H_i + I_i + 2) + \Sigma_{1 \le j \le m \wedge j \neq i}(e_{ji} \cdot (b_j + H_j + I_j + 2))$ for $i = 1, \ldots, k$. Thus we have obtained a sound (but incomplete) small model property for deductive verification of unstratified BDS systems using a partially instantiated inference rule INV*. Next we apply this new verification methodology on the well-known Peterson's mutual exclusion algorithm.

3.1.7 Proving the Safety Property of Peterson's Mutual Exclusion Algorithm

Gary Peterson's famous algorithm for two-process mutual exclusion was first published in 1981, he also gave an algorithm to deal with mutual exclusion for more than two processes. In Fig. 3.1 we present a version of Peterson's mutual exclusion algorithm for N processes where $N > 1$ (written in the SPL language).

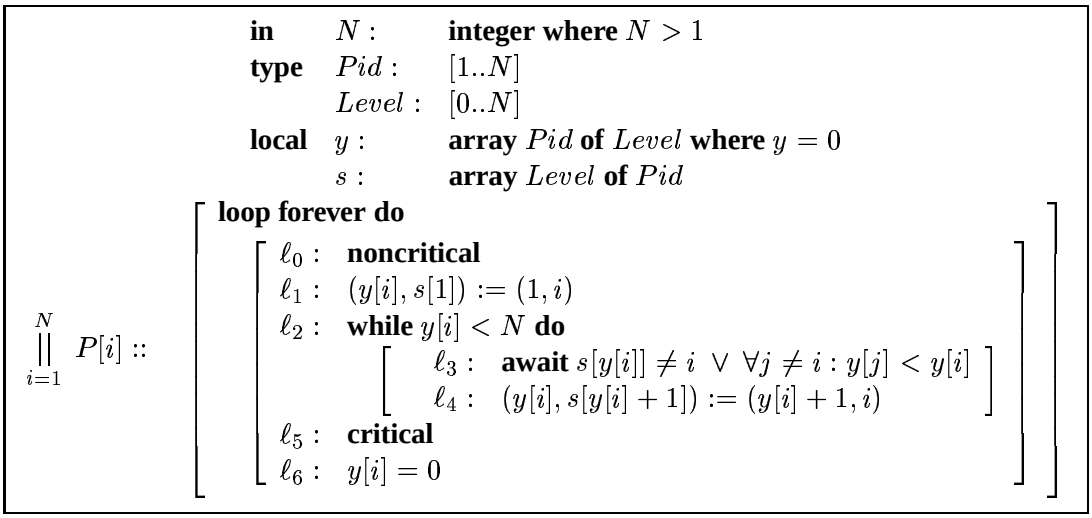


Figure 3.1: Parameterized Peterson's Mutual Exclusion Algorithm

Each process is in the noncritical section at location ℓ_0. At ℓ_1 process $P[i]$ occupies level 1

by assigning $y[i] = 1$. Since $P[i]$ is the most recent process gaining access to level 1, $s[1]$ is assigned to i at the same time. The loop at ℓ_2 controls when process $P[i]$ can advance to a higher level, and the end of the loop condition is satisfied when process $P[i]$ reaches level N and subsequently gains access to the critical section. Inside the loop, ℓ_3 dictates that $P[i]$ can advance to the next level only when either there is another process which gains access to the current level after $P[i]$, or all other processes are at lower levels than $P[i]$. Finally when process $P[i]$ gets out of the critical section, it assigns $y[i] = 0$ at ℓ_6. In location ℓ_4 we have the atomic assignment $(y[i], s[y[i] + 1]) = (y[i] + 1, i)$ that advances process $P[i]$ to the next level. If one wants to break this assignment into two assignments, then extra care must be taken to make sure that the assignment $y[i] := y[i] + 1$ precedes the assignment $s[y[i] + 1] := i$. Otherwise the program will no longer be correct and mutual exclusion can be violated.

We are particularly interested in the above Peterson's algorithm because it contains a simple unstratified array structure $y : Pid \to Level$ and $s : Level \to Pid$ and provides an interesting case of an unstratified BDS system. In order to use deductive methods to prove the safety property that *no two processes are in the critical section at the same time,* our main obligations are to discharge the premises I2 and I3 in the rule INV. In the next two sections we demonstrate how to apply our methodologies to accomplish these two tasks.

Discharging the Verification Condition I2

In Peterson's Algorithm (Fig. 3.1) we have two types of indices, Pid with range $[1..N]$ and $Level$ with range $[0..N]$, and two arrays $y : Pid \mapsto Level$ and $s : Level \mapsto Pid$. The transition relation ρ can also be written as a simple R-assertion of the form $\exists i_3, l_3, l_4\ \forall i, l : R(i, i_3, i_4, l, l_4)$ for $i, i_3 : Pid$ and $l, l_3, l_4 : Level$. We would like to prove the verification condition I2: $\varphi \wedge \rho \longrightarrow \varphi'$. where φ is a simple R-assertion of the form: $\forall i_1, i_2, l_1, l_2 : \psi$ where i_1, i_2 are of type Pid and l_1, l_2 are of type $Level$. The intuition behind why we need two Pid indices and two $Level$ indices for φ comes from the observation of transitions at location ℓ_3 and ℓ_4. Those statements affect the levels $l_1 = y[i]$ and $l_2 = y[i] + 1$ and the processes $s[l_1]$ and $s[l_2]$. In a later section on heuristics for the automatic generation of

invariants we will discuss in detail how to generate this auxiliary invariant φ, which turned out to be like the following inductive invariant:

$$\forall i_1, i_2, l_1, l_2 > 0 : y[i_1] = l_2 \wedge l_1 \leq l_2 \wedge s[l_1] = i_2 \longrightarrow s[l_2] = i_1 \vee y[i_2] = l_1$$

In order to explain this invariant clearly, we need to introduce the concept of *a level being contested*. We say level l_1 is *contested* if $\exists i_1 \neq i_2 : y[i_1] \geq l_1 \wedge y[i_2] \geq l_1$. Alternatively one may use an equivalent definition, only seemingly stronger, which says level l_1 is contested if $\exists i_1 \neq i_2, l_2 : (y[i_1] = l_2 \wedge l_1 \leq l_2 \wedge s[l_1] = i_2)$. By using the following program invariant:

$$\exists i_1 \neq i_2, l_2 : (y[i_1] = l_2 \wedge l_1 \leq l_2 \wedge s[l_1] = i_2 \longrightarrow \exists i_3 \neq i_1 : y[i_3] = l_1)$$

we see that for $i_1 \neq i_3$ we have $y[i_1](= l_2) \geq l_1$ and $y[i_3] = l_1$ which implies that level l_1 is contested.

Now we can interpret the invariant φ as follows: if level $l_1(\leq l_2)$ is contested (when $i_1 \neq i_2$) then it is occupied by process $i_2(= s[l_1])$, i.e. $y[i_2] = l_1$; if level $l_1(\leq l_2)$ is not contested ($i_1 = i_2$) then the levels l_2 and l_1 must point to the same process, that is $s[l_2](= s[l_1]) = i_1$.

In both the transition relation ρ and the assertion φ (also observable from the program), atomic formulas assume the following forms (for Pid variables i_r, i_s and $Level$ variables l_r, l_s):

$$\begin{aligned} i_r &= i_s, \\ s[l_r] &= i_s, \\ l_r &= l_s + 1, \\ l_r &= l_s, \\ l_r &< l_s, \\ y[i_r] &= l_s \end{aligned}$$

which meet the requirements of simple R-assertions with the exception of the atomic formula $l_r = l_s + 1$. We argue that adding this atomic formula to R-assertions will not change

the small model theorems that we established because the crucial model reduction step we use in our proofs preserves the truth of such atomic formulas. To be more convincing, there is a translation of a formula with the form $\exists l_r, l_s : l_r = l_s + 1 \wedge \forall \vec{t} : R(l_s, l_r, \vec{t})$ to the following equivalent R-assertion:

$$\exists l_r, l_s : l_s < l_r \wedge (\forall l : l \leq l_s \vee l_r \leq l) \wedge \forall \vec{t} : R(l_r, l_s, \vec{t})$$

So for simplicity we admit the notation $l_r = l_s + 1$ as an atomic formula.

We first apply skolemization and partial instantiation on the premise I2 and arrive at the following stronger *quantifier-free* simple R-assertion, I2*:
(to simplify the notations we have $V^0 = \{i_1^0, i_2^0, i_3^0\}, W^0 = \{l_1^0, l_2^0, l_3^0, l_4^0\}$)

$$\text{I2}^* : \bigwedge_{\substack{i_1, i_2 \in V^0 \\ l_1, l_2 \in W^0}} \psi(i_1, i_2, l_1, l_2) \wedge \bigwedge_{\substack{i \in V^0 \setminus \{i_3^0\} \\ l \in W^0 \setminus \{l_3^0, l_4^0\}}} R(i_3^0, l_3^0, l_4^0, i, l) \longrightarrow \psi'(i_1^0, i_2^0, l_1^0, l_2^0)$$

Applying the small model property for the above *quantifier-free* simple R-assertion, we get $H_1 = 1$ (for i_3^0), $H_2 = 2$ (for l_3^0, l_4^0) and $I_1 = 2$ (for i_1^0, i_2^0), $I_2 = 2$ (for l_1^0, l_2^0) and therefore we only need to prove the truth of the above quantifier-free simple R-assertion for small models $M(n_1, n_2)$ where $n_1 \leq (H_1 + I_1 + 2 + (H_2 + I_2 + 2)) = 11, n_2 \leq (H_2 + I_2 + 2 + (H_1 + I_1 + 2)) = 11$. We can further tighten the bounds in this particular test case by exploiting the following facts obtained from simple observations:

- The constant Pid terms $1, N$ are not used as array indices therefore we can reduce the bounds for both Pid and $Level$ by 2.
- Due to the relations between $Level$ terms $y[i_r] = l_s$ and between Pid terms $s[l_r] = i_s$ we can reduce the small model bounds for both types of indices by at least 1.

Thus it is sufficient to prove the formula I2* for small models $M(n_1, n_2)$ where $n_1 \leq 8, n_2 \leq 8$, which we successfully achieved using TLV, the Weizmann Institute programmable model checker [PS96].

Discharging the Verification Condition I3

The verification condition I3 states:

$$(\forall i_1, i_2, l_1, l_2 : \psi(i_1, i_2, l_1, l_2)) \longrightarrow \forall j_1 \neq j_2 : \neg(y[j_1] = N \wedge y[j_2] = N)$$

That is, the assertion $\varphi = \forall i_1, i_2, l_1, l_2 : \psi$ implies that no two (distinct) processes can occupy level N at the same time, which in turn implies the mutual exclusion safety property. Since I3 is an unstratified simple R-assertion, we first applied the small model method that we used on I2. However the method failed to prove the truth of I3. Recall that we say level l is *contested* if $\exists i \neq j : y[i] \geq l \wedge y[j] \geq l$. The invariant φ guarantees that if level l_1 is contested then it is occupied by process $i_2 (= s[l_1])$, i.e. $y[i_2] = l_1$. In this case, the only valid counter-example to I3 requires the scenario that all levels are *contested* and occupied by *distinct* processes and there are *two* processes on the highest level. This explains why partial instantiation of the universal quantification failed to deliver such a scenario.

However, intuitively we know that I3 has a small model property, which might require a different model reduction technique. Mathematically I3 can be proved by using induction: *if there is a model* $M(N)$ (*where* $N > N_0$ *for a small integer* N_0) *which violates* I3 *then we can construct a smaller model* $M(N-1)$ *which also violates* I3. In consequence, if we can show that I3 holds for all N where $N \leq N_0$ then by induction I3 holds for all $N > N_0$. Our next focus is to define a model reduction method that reduces a counter-example of size N into a counter-example of size $N - 1$ for I3.

A counter-example to I3 is a model of the following assertion (negated I3):

$$(\forall i_1, i_2, l_1, l_2 : \psi(i_1, i_2, l_1, l_2)) \wedge \exists j_1 \neq j_2 : (y[j_1] = N \wedge y[j_2] = N)$$

As usual we skolemize j_1, j_2 and get:

$$(\forall i_1, i_2, l_1, l_2 : \psi(i_1, i_2, l_1, l_2)) \wedge (y[j_1^0] = N \wedge y[j_2^0] = N)$$

for some $j_1^0, j_2^0 \in [1..N]$.

The mathematical induction step is formalized as follows (M is a counter-example model):

$$(\forall i_1, i_2, l_1, l_2 : M[\psi(i_1, i_2, l_1, l_2)] \wedge j_1^0 \neq j_2^0 \wedge M[y][j_1^0] = N \wedge M[y][j_2^0] = N) \wedge \rho_r$$
$$\longrightarrow (\forall i_1, i_2, l_1, l_2 : \hat{M}[\psi(i_1, i_2, l_1, l_2)]) \wedge \exists j_1, j_2 : \hat{M}[y][j_1] = N - 1 \wedge \hat{M}[y][j_2] = N - 1$$

for some reduction ρ_r yet to be defined. By applying skolemization to the right-hand side, we get:

$$\forall i_1, i_2, l_1, l_2 : M[\psi(i_1, i_2, l_1, l_2)] \wedge M[y][j_1^0] = N \wedge M[y][j_2^0] = N \wedge \rho_r \longrightarrow$$
$$\hat{M}[\psi(v_1, v_2, w_1, w_2)] \wedge \hat{M}[y][k_1^0] = N - 1 \wedge \hat{M}[y][k_2^0] = N - 1$$

for all $v_1, v_2, w_1, w_2 \in [1..N-1]$ and for some $k_1^0, k_2^0 \in [1..N-1]$ and $j_1^0, j_2^0 \in [1..N]$.

In order to define the reduction relation ρ_r we need to use the following proposition:

$$\forall i_1, i_2, l_1, l_2 : \psi(i_1, i_2, l_1, l_2) \wedge y[j_1^0] = N \wedge y[j_2^0] = N$$
$$\longrightarrow \forall\, 0 < l < N\, \exists\, i : s[l] = i \wedge y[i] = l \wedge i \notin \{j_1^0, j_2^0\}$$

(Note that $y[i]$ is initially assigned to 0, once it is assigned a non-zero value then it will remain non-zero.)

Applying instantiation to the right-hand side and we have the special case:

$$\forall i_1, i_2, l_1, l_2 : \psi(i_1, i_2, l_1, l_2) \wedge y[j_1^0] = N \wedge y[j_2^0] = N$$
$$\longrightarrow s[1] = i^0 \wedge y[i^0] = 1 \wedge i^0 \notin \{j_1^0, j_2^0\}$$

for some $i^0 \in [1..N]$. That means that $s[1]$ always points to a process index in the range

$[1..N]$ which is different from j_1^0, j_2^0. In other words:

$$\forall i_1, i_2, l_1, l_2 : \psi(i_1, i_2, l_1, l_2) \wedge y[j_1^0] = N \wedge y[j_2^0] = N \wedge s[1] = i^0 \\ \longrightarrow y[i^0] = 1 \wedge i^0 \notin \{j_1^0, j_2^0\}$$

We replace the universal quantification on the left-hand side by partial instantiation and obtain a much weaker assertion:

$$\psi(j_1^0, i^0, 1, N) \wedge \psi(j_2^0, i^0, 1, N) \wedge y[j_1^0] = N \wedge y[j_2^0] = N \wedge s[1] = i^0 \\ \longrightarrow y[i^0] = 1 \wedge i^0 \notin \{j_1^0, j_2^0\}$$

The validity of the above proposition is proved using TLV. This in turn allows us to perform the following reduction:

- Remove the process i^0 to which $s[1](= i^0)$ points (it is guaranteed that i^0 is not a process at the highest level, in particular $i^0 \neq j_1^0, j_2^0$).
- Reduce all the levels bigger than 1 by 1, and all the process indices higher than i^0 by 1.

Formally the reduction ρ_r is defined as follows:

- $\hat{M}[x] = M[x]$ for each boolean variable x.
- For each $Level$ variable l, $\hat{M}[l] = M[l] - 1$ for $M[l] > 1$ else l.
- For each Pid variable i, $\hat{M}[i] = M[i] - 1$ for $M[i] > i^0$ else $M[i]$.
- For each array y: $Pid \mapsto Level$,

$$\begin{array}{lrlll} \hat{M}[y] = \lambda u : & \text{if} & u < i^0 \wedge M[y][u] > 1 & \text{then} & M[y][u] - 1 \\ & \text{elsif} & u < i^0 \wedge M[y][u] \leq 1 & \text{then} & M[y][u] \\ & \text{elsif} & u \geq i^0 \wedge M[y][u+1] > 1 & \text{then} & M[y][u+1] - 1 \\ & \text{else} & M[y][u+1]. & & \end{array}$$

– For each array $s : Level \mapsto Pid$,

$$\hat{M}[s] = \lambda v : \quad \text{if} \quad M[s][v+1] > i^0 \quad \text{then} \quad M[s][v+1] - 1$$
$$\text{else} \quad M[s][v+1]$$

Applying the partial instantiation to the deduction formula we get

$$\begin{aligned}
&M[\psi(v_1, v_2, w_1, w_2)] \wedge M[\psi(v_1, v_2, w_1, w_2 + 1)] \wedge \\
&M[\psi(v_1, v_2, w_1 + 1, w_2)] \wedge M[\psi(v_1, v_2, w_1 + 1, w_2 + 1)] \wedge \\
&M[\psi(v_1, v_2 + 1, w_1, w_2)] \wedge M[\psi(v_1, v_2 + 1, w_1, w_2 + 1)] \wedge \\
&M[\psi(v_1, v_2 + 1, w_1 + 1, w_2)] \wedge M[\psi(v_1, v_2 + 1, w_1 + 1, w_2 + 1)] \wedge \\
&M[\psi(v_1 + 1, v_2, w_1, w_2)] \wedge M[\psi(v_1 + 1, v_2, w_1, w_2 + 1)] \wedge \\
&M[\psi(v_1 + 1, v_2, w_1 + 1, w_2)] \wedge M[\psi(v_1 + 1, v_2, w_1 + 1, w_2 + 1)] \wedge \\
&M[\psi(v_1 + 1, v_2 + 1, w_1, w_2)] \wedge M[\psi(v_1 + 1, v_2 + 1, w_1, w_2 + 1)] \wedge \\
&M[\psi(v_1 + 1, v_2 + 1, w_1 + 1, w_2)] \wedge M[\psi(v_1 + 1, v_2 + 1, w_1 + 1, w_2 + 1)] \wedge \\
&M[y][j_1^0] = N \wedge M[y][j_2^0] = N \wedge \rho_r^* \quad \longrightarrow \\
&\hat{M}[\psi(v_1, v_2, w_1, w_2)] \wedge \hat{M}[y][\hat{j_1^0}] = N - 1 \wedge \hat{M}[y][\hat{j_2^0}] = N - 1
\end{aligned}$$

where in particular $\hat{j_1^0} = j_1^0 + (j_1^0 > i^0)$ and $\hat{j_2^0} = j_2^0 + (j_2^0 > i^0)$, and ρ_r^* is a quantifier-free simple R-assertion representing only the part of the model reduction relation from M to $\hat{M}$ that is necessary for obtaining $\hat{M}[\psi(v_1, v_2, w_1, w_2)]$. Now we can apply the small model method to prove the above quantifier-free simple R-assertion, which turns out to be true for all models $M(N)$ where $N \leq 6$. Thus we conclude that the induction formula is valid for all $M(N)$ where $N > 1$. Having established the soundness of the induction formula, we only need to prove I3 on small instances of Peterson's algorithm $S(N)$, and conclude by induction that I3 holds for all $S(N)$ where $N > 1$.

3.2 Automatic Generation of Invisible invariants

For an assertion to be an invariant of a given program, it can be implied by the reachable states, that is

$$reach \longrightarrow invariant$$

However an invariant is not necessarily inductive. Although it always satisfies the first INV rule I1, it may violate the second INV rule I2. We can always strengthen an invariant by excluding the states that violate I2, and derive a stronger invariant which lies between the reachable states and the original invariant. Repeating this process we can ultimately reach an inductive invariant for a finite state system, given the fact that this process is bounded by the strongest inductive invariant which is the set of reachable states itself.

Inductive invariants are essential for deductive verification. Their presence is ubiquitous in the inference rules. The successful application of rule INV requires an auxiliary inductive invariant which implies the property. In the remaining part of this chapter we dedicate our discussion to the problem of defining algorithms and heuristics to generate assertions that can serve as candidates for inductive invariants.

The problem of automatically constructing invariants from the program description has been intensively investigated in [KM76], [GW75], [BLS96], [GS96]. Here our focus is not on arbitrary algorithms for generating assertions. The types of inductive invariants we hope to generate also need to satisfy the logical form requirements imposed by small model properties for BDS systems. This restricts our study mainly to heuristics for deriving assertions of certain given logical forms. Such a restriction turned out to be not too limiting at all, instead assertions with these simple logical forms are usually very useful and in many cases sufficient for proving interesting properties.

The word “invisible” in the name *invisible invariants* refers to the fact that due to the automatic generation of these assertions we never “see” or even know their exact logical formula except for the logical forms that they assume.

3.2.1 Generating Invisible Invariants with Universal Quantifiers

Since for any program invariant we have $reach \rightarrow invariant$, or equivalently $(reach \wedge invariant) \leftrightarrow reach$, one can view the derivation of such invariants as a process of extracting useful conjunctive subformulas from $reach$. For the requirements of small model theorems we mainly are interested in subformulas with special logical forms. We start with state formulas involving only universal quantifiers, and later move on to heuristics for deriving formulas of other logical forms. These heuristics can be useful for general deductive verification if not for automatic parameterized verification.

Generating Assertions of the Form $\forall i : \psi(i)$

The intuitive idea behind this heuristic comes from the observation that usually an invariant of the form $\forall i : \psi(i)$ describes the common behavior of individual processes in a parameterized system. With a large enough instance $S(N)$ of a parameterized system (with N no less than the cutoff value computed by the small model theorem), we calculate the reachable states for the system $S(N)$ and project that onto a single process i_0 in the hope that the result will capture the behavior as described by $\psi(i_0)$. By applying universal quantification we can obtain an assertion with the desired logical form. Here is the heuristic for generating a (possibly inductive) assertion of the form $\forall i : \psi(i)$:

1. Compute the assertion $reach := \Theta \diamond \rho^*$ which characterizes all the reachable states of system $S(N_0)$, a small instance of the parameterized BDS system (usually we use the cutoff value for N_0).
2. Project $reach$ onto a single index, say index i_0, by projecting away all the references to variables subscripted by indices other than i_0. This usually results in a quantifier-free formula, ψ_{i_0}.
3. Generalize ψ_{i_0} to assertion ψ_i by generalizing the index i_0 to i.
4. Finally apply universal quantification to obtain an assertion $\forall i : \psi(i)$.

Figure 3.2: Heuristics A-1: Generating Invariants $\forall i : \psi(i)$

In the second step, projecting away a set of variables is performed by using the BDD existential quantification operation. In our implementation we build a relation for each variable which we do not project away, for example $next(u[i_0]) = u[i_0]$. Let $f(V, V')$ be the conjunction of these preservation relations, then ψ_{i_0} is given by $unprime(\exists V : reach(V) \wedge f(V, V'))$, the *successor* operation.

The second step sometimes might require slight modifications when a parameterized system is not very symmetrical. In such cases, a typical modification is that we first project $reach$ on to a set of different indices, $i_0, i_1, \ldots, i_k$, and obtain formulas $\chi_0, \chi_1, \ldots, \chi_k$. Then ψ_{i_0} can be defined as $\chi_0 \vee \chi_1[i_1 \rightarrow i_0] \vee \ldots \vee \chi_k[i_k \rightarrow i_0]$ where $\chi_k[i_k \rightarrow i_0]$ is the operation which renames the index i_k to i_0. We will illustrate the usage of such heuristics in later examples. All these techniques have produced many good experimental results.

As a heuristic, the algorithm in Fig. 3.2 does not guarantee the successful generation of an *inductive* invariant. However if $reach$ contains a universal quantified subformula, that is, $reach = reach \wedge \forall i : \psi(i)$ then the algorithm is likely to obtain a program invariant of the form $\forall i : \psi(i)$. The key lies in the projection operation performed in the second step. The following observations give supportive reasons for our selection of heuristics:

- The projection of a universally-quantified formula $\forall i : \psi(i)$ onto the index i_0 often gives us $\psi(i_0)$.
- The projection of an existentially-quantified formula $\exists i : \psi(i)$ onto the index i_0 usually gives us the tautology *true* due to the symmetry of the processes in a parameterized system. In the case of a not very symmetrical parameterized system, the modified step 2 may also produce the tautology *true*.
- The projection of $\forall i : \psi(i) \wedge \exists j : \varphi(j)$ onto i_0 usually yields $\psi(i_0)$ (a natural consequence of the first two observations), which effectively "filters" out the existentially-quantified subformula and preserves the content of the universally-quantified subformula.

Example: generating assertion $\forall i : \psi(i)$ for the parameterized program MUX-SEM

Here is an example of how to apply the heuristic of Fig. 3.2 to the parameterized system MUX-SEM shown in Fig. 2.1. To generate the assertion with a singly-indexed universal quantifier for MUX-SEM, we first calculate the reachable states for $S(4)$ and get

$$reach := (y + (\Sigma_{i=1}^{4}(\pi[i] \in \{2,3\})) = 1)$$

We then project $reach$ onto index 1 and obtain:

$$\psi(1) := (\pi[1] \in \{2,3\} \longrightarrow y = 0)$$

By generalizing 1 to i we get

$$\psi(i) := (\pi[i] \in \{2,3\} \longrightarrow y = 0)$$

Although the resulting assertion $\forall i : (\pi[i] \in \{2,3\} \longrightarrow y = 0)$ is an invariant, it is not an inductive one.

Example: generating inductive assertion $\forall i : \psi(i)$ for the augmented MUX-SEM

In the above example we generated a non-inductive assertion $\forall i : \psi(i)$ for program MUX-SEM. This example shows how we can generate an inductive invariant $\forall i : \psi(i)$ by augmenting program MUX-SEM with a global variable $last\_entered$ (Fig. 3.3).

in N : **integer where** $N > 1$
local y : **boolean where** $y = 1$
local $last\_entered$: $[1..N]$

$$\Big\|_{i=1}^{N} P[i] :: \left[\begin{array}{l} \textbf{loop forever do} \\ \quad \left[\begin{array}{ll} 0: & \textbf{noncritical} \\ 1: & \langle \textbf{request } y; \; last\_entered := i \rangle \\ 2: & \textbf{critical} \\ 3: & \textbf{release } y \end{array}\right] \end{array}\right]$$

Figure 3.3: Augmented Program MUX-SEM

So at location 1 we use an atomic operation to record the process index i whenever process $P[i]$ enters its critical section. To generate an assertion with a singly-indexed universal quantifier for the augmented MUX-SEM, we first calculate the reachable states for $S(5)$:

$$\begin{aligned} reach \;\; := \;\; & (y + (\Sigma_{i=1}^{5}(\pi[i] \in \{2,3\})) = 1) \wedge \\ & \forall j : (\pi[j] \in \{2,3\} \longleftrightarrow (y = 0 \wedge last\_entered = j)) \end{aligned}$$

We then project $reach$ onto index 1 and obtain:

$$\psi(1) := (\pi[1] \in \{2,3\} \longleftrightarrow y = 0 \wedge \mathit{last\_entered} = 1)$$

By generalizing 1 to i we get

$$\psi(i) := (\pi[i] \in \{2,3\} \longleftrightarrow y = 0 \wedge \mathit{last\_entered} = i)$$

The resulting assertion $\forall i : (\pi[i] \in \{2,3\} \leftrightarrow y = 0 \wedge last\_entered = i)$ turns out to be an inductive one and implies the safety property of mutual exclusion. This technique of augmenting a parameterized system with auxiliary global variables has also been successfully applied in the case of German's Cache Protocol [PRZ01], a most successful application of the invisible invariant method, to generate an inductive assertion $\forall i : \psi(i)$. The above observation might suggest to us the following conjecture:

Conjecture 1 *If a parameterized system $S(N)$ has an inductive assertion of the form $\forall i_0, \ldots, i_k : \psi(i_0, \ldots, i_k)$ then we can always derive an inductive assertion $\forall i_0, \ldots, i_m : \xi(i_0, \ldots, i_m)$ where $m < k$ by augmenting the system $S(N)$ with auxiliary global variables.*

Generating Assertions of the Form $\forall i \neq j : \psi(i, j)$

Now we would like to define a heuristic for generating invariants of the form $\forall i \neq j : \psi(i, j)$. It is a slight modification of the previous heuristic of Fig. 3.2:

1. Compute the assertion $reach := \Theta \diamond \rho^*$ which characterizes all the reachable states of system $S(N_0)$, a small instance of the parameterized BDS system.
2. Project $reach$ onto two *disjoint* indices, $i_0 \neq j_0$, by projecting away all the references to variables subscripted by indices other than i_0, j_0. This usually results in a quantifier-free formula, ψ_{i_0,j_0}.
3. Generalize ψ_{i_0,j_0} to $\psi_{i,j}$ by generalizing the index i_0 to i and the index j_0 to j.
4. Apply universal quantification to obtain an assertion $\forall i \neq j : \psi(i,j)$.

Figure 3.4: Heuristics A-2: Generating Invariants $\forall i \neq j : \psi(i,j)$

In step 2 we can usually project on any pair of two disjoint indices. However in some cases when the system is not very symmetrical it might be necessary to project on several pairs of disjoint indices and rename the results to that of a single pair i_0, j_0, and then to take the disjunction of the renamed results to obtain the final quantifier-free formula ψ_{i_0,j_0}. This modification of step 2 is very similar to the modification suggested for the previous heuristic in Fig. 3.2.

Example: generating assertion $\forall i \neq j : \psi(i,j)$ for MUX-SEM

To generate an assertion with doubly-indexed universal quantification for MUX-SEM, we first calculate the reachable states for $S(5)$ and get

$$reach := (y + (\Sigma_{i=1}^{5}(\pi[i] \in \{2,3\})) = 1)$$

We project $reach$ onto index 1 and 2 and obtain:

$$\psi(1,2) := \left(\begin{array}{ll} & \pi[1] \in \{2,3\} \longrightarrow (y = 0) \wedge \pi[2] \in \{0,1\} \\ \wedge & \pi[2] \in \{2,3\} \longrightarrow (y = 0) \wedge \pi[1] \in \{0,1\} \end{array} \right)$$

Finally we generalize 1 to i and 2 to j and get:

$$\psi(i,j) := \left(\begin{array}{ll} & \pi[i] \in \{2,3\} \longrightarrow (y = 0) \wedge \pi[j] \in \{0,1\} \\ \wedge & \pi[j] \in \{2,3\} \longrightarrow (y = 0) \wedge \pi[i] \in \{0,1\} \end{array} \right)$$

The generated assertion $\forall i \neq j : \psi(i,j)$ turns out to be an inductive invariant which suffices to prove the mutual exclusion safety property for MUX-SEM.

Generating Assertions with Multi-indexed Universal Quantifiers

The general assertions we would like to generate are of the form $\forall \vec{i^1}, \ldots, \vec{i^k} : \psi(\vec{i^1}, \ldots, \vec{i^k})$ where $\vec{i^k}$ denotes a vector of disjoint indices of type **type**$_k$. Consequently, assertions of the form $\forall i \neq j : \psi(i,j)$ can be viewed as a special case where i, j are two disjoint indices of the same type. The following heuristic shown in Fig. 3.5 is a natural extension of the heuristic in Fig. 3.4.

1. Compute the assertion $reach := \Theta \diamond \rho^*$ which characterizes all the reachable states of system $S(N_1^0, \ldots, N_k^0)$, a small instance of the parameterized BDS system (usually $N_1^0, \ldots, N_k^0$ are the bounds calculated by the small model property).
2. Project $reach$ onto **type**$_r$ indices $1, \ldots |\vec{i^r}|$ for all $r = 1, \ldots, k$, where $|\vec{i^r}|$ represents the size of the vector $\vec{i^r}$, by projecting away all the references to variables subscripted by other indices. This usually results in a quantifier-free formula $\hat{\psi}$.
3. Generalize $\hat{\psi}$ to assertion $\psi(\vec{i^1}, \ldots, \vec{i^k})$ by generalizing **type**$_r$ index j to i_j^r.
4. Apply universal quantification to obtain an assertion $\forall \vec{i^1}, \ldots, \vec{i^k} : \psi(\vec{i^1}, \ldots, \vec{i^k})$.

Figure 3.5: Heuristics A-k: Generating Invariants $\forall \vec{i^1}, \ldots, \vec{i^k} : \psi(\vec{i^1}, \ldots, \vec{i^k})$

In practical test cases these heuristics can often generate an assertion of the desired form which turns out to be a program invariant. It is especially true when the program indeed contains such an invariant of the desired logical forms. In cases where the generated assertion is not an *inductive* invariant, it still can serve as a good program invariant to assist in the proofs of various properties.

Example: Generating Invariants for Peterson's Mutual Exclusion Algorithm

Although the small model property for unstratified BDS systems differs from that of stratified BDS systems, the procedures for generating auxiliary inductive assertions remain the

same. Consider Peterson's mutual exclusion algorithm in Fig. 3.1. In order to prove the safety property of mutual exclusion for this system, we need an inductive assertion of the following form:

$$\varphi = \forall i_1, i_2, l_1, l_2 > 0 : \psi(i_1, i_2, l_1, l_2)$$

It is an assertion with only universal quantifications over two type Pid variables and two type $Level$ variables. To generate an inductive assertion of the form $\forall i_1, i_2, l_1, l_2 : \psi(i_1, i_2, l_1, l_2)$ we use the proposed heuristics on system $S(5)$ and $S(6)$. For $S(5)$ we first project the reachable states on processes 1, 2 and on the levels 3, 4 and follow the heuristics to generate the invariant. The resulting assertion turned out to be not inductive. It is not difficult to see that the projection on the level pair (3, 4) might capture different aspects of the system behavior than the projection on the level pair (2, 3) or the leval pair (2, 4). So we modify our heuristics to project the reachable states on level pairs (2, 3), (3, 4) and (2, 4), and then take the disjunction of the projection results for each value pair after renaming. During the generalization step we conjunct the results for all possible value pairs. This modification worked and generated an inductive invariant of the desired form for $S(5)$. For system $S(6)$ it turned out that it is sufficient to project only on level (3, 4) and to follow the usual generalization procedure. The generated invariant is shown to be inductive for $S(6)$ as we hoped. However the computational complexity in the case of $S(6)$ is so high that we need to provide a good initial BDD variable ordering in order to finish the computation.

3.2.2 Generating Invisible Invariants with Existential Quantifiers

The small model theorems we have established before assume that the properties and the inductive invariants are logical formulas with only universal quantification. However the same methodology can easily be extended to handle properties and invariants with existential quantification. In this section we discuss heuristics for generating assertions with only existential quantifiers.

It is a common mathematical practice to transform a new problem into the realm of problems where there are known solutions. The main observation here suggests that an

assertion with only existential quantification can be written as the negation of an assertion with only universal quantification. We hope that we can derive our new heuristics from the known heuristics of generating assertions with only universal quantifiers.

Generating Invariants with Singly-indexed Existential Quantifiers

Let us first look at a special case where $reach$ assumes the form $\exists i : \varphi(i)$. Here is a simple procedure to generate an assertion with existential quantifiers:
– Compute $reach$ for $S(N_0)$.
– Take the negation of $reach$ and project it onto a single index i_0 and get $\psi(i_0)$.
– Take the negation of $\psi(i_0)$ and generalize the index i_0 to i.
– Apply existential quantification and obtain an assertion $\exists i : \neg\psi(i)$.
The key step is to take the negation of $reach$ which gives us an assertion with a universal quantifier. This enables the next projection step to extract the needed subformula.

Unfortunately, the above procedure does not work in general cases where $reach$ assumes a more complicated form. In the simple case where $reach = \forall i : \varphi(i) \wedge \exists j : \psi(j)$, if we follow the procedure and take the negation of $reach$ then project on a single index i_0, the most likely result is the constant *true*. The loss of information originates in the negation of $reach$ where we lose the conjunctive form and get the disjunction of two negated subformulas. The heuristic as shown in Fig. 3.6 provides a remedy to such situations and renders a much more general procedure for generating invariants of the form $\exists j : \psi(j)$.

If $reach = \forall i : \varphi(i) \wedge \exists j : \psi(j)$ this procedure actually generates an invariant $invd = \exists j : (\psi(j) \vee \neg\varphi(j))$. One can check the validity of this invariant by showing that $invc \wedge invd = reach$ where $invc$ is the generated assertion $\forall i : \varphi(i)$.

1. Compute assertion $reach$ for the reachable states of system $S(N_0)$, a small instance of the parameterized BDS system.
2. Compute a universally-quantified invariant $invc = \forall \vec{i} : \varphi(\vec{i})$.
3. Compute $remains := invc \wedge \neg reach$.
4. Project $remains$ on the index i_0 and obtain rem_{i_0}. Let $\psi_{i_0} := \neg rem_{i_0}$.
5. Generalize ψ_{i_0} to ψ_i by generalizing i_0 to i.
6. Apply existential quantification to obtain the assertion $\exists i : \psi(i)$.

Figure 3.6: Heuristics E-1: Generating Invariants $\exists i : \psi(i)$

Example: generating assertion $\exists i : \psi(i)$ for program MUX-SEM

Here is an example of how to apply the heuristic of Fig. 3.6 to MUX-SEM shown in Fig. 2.1. We first calculate the reachable states for $S(5)$ and get

$$reach := (y + (\Sigma_{i=1}^{5}(\pi[i] \in \{2,3\})) = 1)$$

The inductive invariant $invc$ was generated in the previous example, $invc = \forall i \neq j : (\pi[i] \in \{2,3\} \rightarrow (y = 0) \wedge \pi[j] \in \{0,1\}) \wedge (\pi[j] \in \{2,3\} \rightarrow (y = 0) \wedge \pi[i] \in \{0,1\})$. Then we compute $\neg reach \wedge invc$ and get

$$rem = \forall i : (\pi[i] \in \{0,1\} \wedge (y = 0)).$$

Next we project rem onto index 1 and take its negation and obtain

$$\psi(1) := \neg(\pi[1] \in \{0,1\} \wedge y = 0).$$

We generalize $\psi(1)$ to $\psi(i)$ by generalizing 1 to i, and get

$$\psi(i) := (\pi[i] \in \{2,3\} \vee y = 1).$$

Using disjunction we obtain the existentially quantified assertion $\exists i : \pi[i] \in \{2,3\} \vee y = 1$, or equivalently $y = 0 \rightarrow \exists i : \pi[i] \in \{2,3\}$.

Generating Invariants with Multi-indexed Existential Quantifiers

The following heuristic is a generalization of the heuristic in Fig. 3.6 for generating assertions of the form $\exists i_1, \ldots, i_k : \psi(i_1, \ldots, i_k)$ where $i_1, \ldots, i_k$ are of the same index type.

1. Compute assertion $reach$ for the reachable states of system $S(N_0)$, a small instance of the parameterized BDS system.
2. Compute a universally-quantified invariant $invc = \forall \vec{i} : \varphi(\vec{i})$.
3. Compute $remains := invc \wedge \neg reach$.
4. Project $remains$ on disjoint indices $1, \ldots, k$ to obtain rem. Let $\hat{\psi} := \neg rem$.
5. Generalize $\hat{\psi}$ to $\psi(i_1, \ldots, i_k)$ by generalizing j to i_j for $j = 1, \ldots, k$.
6. Apply existential quantification to obtain $\exists i_1, \ldots, i_k : \psi(i_1, \ldots, i_k)$.

Figure 3.7: Heuristics E-k: Generating Invariants $\exists i_1, \ldots, i_k : \psi(i_1, \ldots, i_k)$

3.3 Verification of Waiting-for Properties of Parameterized Systems

So far we have discussed automatic deductive verification of invariance properties (the property assume the form of $\Box\, p$ for a state formula p) of parameterized systems using the method of invisible invariants. In this section we study another important class of safety properties which can be described by general waiting-for formulas. One example of such a property is bounded-overtaking, which measures in some sense the fairness of a system.

Typical properties for bounded overtaking are described by nested waiting-for LTL formulas $p \Rightarrow q_m \,\mathcal{W}\, q_{m-1} \ldots q_1 \,\mathcal{W}\, q_0$. For parameterized systems the nested waiting-for

properties often assume the form of a quantified temporal formula, such as $\varphi = \forall i, j : p(i,j) \Rightarrow q_m(i,j)\, \mathcal{W}\, \ldots q_1(i,j)\, \mathcal{W}\, q_0(i,j)$. In order to show that the parameterized system modeled by FDS $\mathcal{D}$ satisfies φ, we need to apply the following transformation:

$$\begin{aligned} & \mathcal{D} \models \forall i, j : p(i,j) \Rightarrow q_m(i,j)\, \mathcal{W}\, \ldots q_1(i,j)\, \mathcal{W}\, q_0(i,j) \iff \\ \forall i, j : \quad & \mathcal{D} \models p(i,j) \Rightarrow q_m(i,j)\, \mathcal{W}\, \ldots q_1(i,j)\, \mathcal{W}\, q_0(i,j) \end{aligned}$$

If the system is symmetric then we can arbitrary choose the indices i, j to be $1, 2$ and instead prove $\mathcal{D} \models p(1,2) \Rightarrow q_m(1,2)\, \mathcal{W}\, \ldots q_1(1,2)\, \mathcal{W}\, q_0(1,2)$. In general for rule INV we have two extra parameters i, j, which we need to take into consideration when calculating the cutoff value using the small model theorem.

The deductive method to verify nested waiting-for formulas is as follows:

1. Generate auxiliary assertions $\varphi_0, \varphi_1, \ldots, \varphi_m$.

2. Check that the premises for the inference rule NWAIT hold:

For integers $i, j,$ and assertions $p, q_0, q_1, \ldots, q_m,$ and $\varphi_0, \varphi_1, \ldots, \varphi_m,$

N1. $p \rightarrow \bigvee_{j=0}^{m} \varphi_j$
N2. $\varphi_i \rightarrow q_i$ for $i = 0, 1, \ldots, m$
N3. $\varphi_i \wedge \rho \rightarrow \bigvee_{j \le i} \varphi'_j$ for $i = 1, \ldots, m$

$$p \Rightarrow q_m\, \mathcal{W}\, q_{m-1} \cdots q_1\, \mathcal{W}\, q_0$$

There are two techniques to automatically generate φ's: forward propagation and backward propagation.

Forward propagation:

To generate an auxiliary assertion φ for a simple waiting-for formula $p \Rightarrow q\, \mathcal{W}\, r$, the forward propagation method computes

$$\varphi = successors(p \wedge \neg r, \rho \wedge \neg r')$$

Here the forward propagation starts from states $p \wedge \neg r$, then computes all the r-free states that are reachable and obtains the auxiliary assertion φ. Finally the inference rule checks that those states in φ must satisfy q.

Backward Propagation:
The backward propagation method on the other hand first computes the states $q \,\mathcal{W}\, r$ and then checks $p \Rightarrow (q \,\mathcal{W}\, r)$. In consequence, backward propagation calculates a bigger set of states than forward propagation.

Here is how we compute the auxiliary assertion φ for $p \Rightarrow (q \,\mathcal{W}\, r)$ using backward propagation:

$$\varphi = \neg(\neg r U(\neg q \wedge \neg r)) \text{ or}$$

$$\varphi = \neg((\neg r \wedge q) U(\neg q \wedge \neg r)).$$

To calculate $(\neg r U(\neg q \wedge \neg r))$ we use the $predecessors$ operator, i.e. $\neg r U(\neg q \wedge \neg r) = predecessors(\neg q \wedge \neg r, \rho \wedge \neg r)$.

For nested waiting-for formulas such as $p \Rightarrow q_2 \,\mathcal{W}\, (q_1 \,\mathcal{W}\, q_0)$ the forward method can not easily be generalized to generate auxiliary assertions. However, backward propagation easily computes the assertions in an iterative manner, that is it first computes $\varphi_1 = q_1 \,\mathcal{W}\, \varphi_0$ where $\varphi_0 = q_0$ and then computes $\varphi_2 = q_2 \,\mathcal{W}\, \varphi_1$.

In order to apply small model theorem on rule NWAIT we need to ensure that the φ_i's are AER-assertions. As we have noted before, due to the quantification over the waiting-for formula for a parameterized system, we have the additional parameters from the instantiation of the quantified variables $i_1, \ldots, i_n$. Fig. 3.8 is the heuristic for generating auxiliary assertions $\varphi_0, \ldots, \varphi_m$ of the form $\forall k : \psi(i_1, \ldots, i_n, k)$.

1. Compute the assertion $\varphi_0, \ldots, \varphi_m$ using the backward propagation method for system $S(N_0)$, a small instance of the parameterized BDS system.
2. Project φ_j (for $j = 0, \ldots, m$) onto a distinct index k_0, in addition to the indices $i_1, \ldots, i_n$, by projecting away all the references to variables subscripted by indices other than $i_1, \ldots, i_n, k_0$. This usually results in a quantifier-free formula, $\varphi_j(i_1, \ldots, i_n, k_0)$.
3. Generalize $\varphi_j(i_1, \ldots, i_n, k_0)$ to assertion $\varphi_j(i_1, \ldots, i_n, k)$ by generalizing the index k_0 to k.
4. Apply universal quantification to obtain $\forall k : \varphi_j(i_1, \ldots, i_n, k)$.

Figure 3.8: Heuristics NW-1: Generating Auxiliary AER-assertions for Rule NWAIT

Case Study 1: Checking 1-Bounded Overtaking for Peterson's Algorithm

We verified a two-process Peterson's algorithm (see Fig. 3.9) using both forward and backward propagation in TLV [PS96], a verification tool set built at the Weizmann Institute.

$$
\begin{array}{l}
\textbf{local} \quad y: \ \textbf{array}[1,2] \ \textbf{of boolean where} \ y = 0 \\
\qquad\quad\ \ s: \ [1,2] \\
\left\| \begin{array}{l} i=1 \\ \\ 2 \end{array} \right. P[i] :: \left[\begin{array}{l}
\textbf{loop forever do} \\
\quad \left[\begin{array}{ll}
\ell_0: & \textbf{noncritical} \\
\ell_1: & (y[i], s) := (1, i) \\
\ell_2: & \textbf{await} \ s \neq i \ \vee \ \forall j \neq i : \neg y[j] \\
\ell_3: & \textbf{critical} \\
\ell_4: & y[i] = 0
\end{array} \right]
\end{array} \right]
\end{array}
$$

Figure 3.9: Two-Process Peterson's Mutual Exclusion Algorithm

First we implemented in TLV the deductive inference rule for checking nested waiting-for formulas (see Fig. 2.5) and added it to the TLV rule files. The procedure `nwait` takes as parameters the assertion p, and the assertion arrays q and phi, both of size $asize$, and then checks the validity of premises N1, N2, N3 in sequence and returns a counter-example after encountering the first violated premise and sets the return code $success$ to 0.

```
Proc nwait(p, &q, &phi, asize, success);

  Let success := 1;

  ----------------------------------------
  -- N1: p -> phi[0] | ... | phi[asize-1] --
  ----------------------------------------
  Print "Checking Premise(nwait) N1\n";
  Let j := asize;
  Let all_phi := 0;
  While (j)
    Let j := j - 1;
    Let all_phi := all_phi | phi[j];
  End -- while (j)
  Let counter := p & !all_phi;
  If (counter)
    Print "Premise N1 is not valid.
           Counter-example =\n";
    Print counter;
    Let success := 0;
  Else
    ------------------------------------------------
    -- N2: phi[i] -> q[i] for i = 0, ..., asize-1  --
    ------------------------------------------------
    Print "Premise N1 is valid. Checking Premise N2.\n";
    Let j := asize;
    While (j)
      Let j := j - 1;
      Let counter := phi[j] & !q[j];
      If (counter)
        Print "Premise N2 is not valid at ", j, ".
               Counter-example =\n";
        Print counter;
        Let success := 0;
        Let j := 0;    -- For quick exit immediately.
      End -- if
    End -- Done checking N2
    If (success)
      ------------------------------------------------
      -- N3: phi[i] & rho -> phi[0]' | ... | phi[i]'  --
      ------------------------------------------------
      Print "Premise N2 is valid.  Checking Premise N3.\n";
      Let i := asize - 1;
```

```
      While (i)
        Let j := i;
        Let lower_phis := phi[j];
        While (j)
          Let j := j - 1;
          Let lower_phis := lower_phis | phi[j];
        End -- done calculating phi[0] | ... | phi[i]
        Let counter := phi[i] & total & !next(lower_phis);
        If (counter)
          Print "Premise N3 is not valid at phi", i, ".
                 Counter-example =\n";
          Print counter;
          Let success := 0;
          Let i := 0;  -- For quick exit immediately
        Else
          Let i := i - 1;
        End
      End -- Done checking N3
      If (success)
        Print "Premise N3 is valid.\n
               * * * Assertion p is invariant.\n";
      End
    End -- Done N2, N3.
   End -- Done N1, N2, N3.
End -- Proc nwait
```

The temporal formula for 1-bounded overtaking of the two-process Peterson's algorithm is as follows:

$$P[1].loc = 2 \longrightarrow \neg(P[2].loc = 3)\ \mathcal{W}\ P[2].loc = 3\ \mathcal{W}\ \neg(P[2].loc = 3)\ \mathcal{W}\ P[1].loc = 3$$

We apply backward propagation to automatically generate the needed auxiliary assertions $phi[0], \ldots, phi[3]$ to be used by procedure `nwait`. In order to satisfy the premise N3 we need to first generate the inductive auxiliary assertion `invc` which can be computed using heuristics A-K shown in Fig. 3.5. The variable `total` represents the transition relation ρ.

```
Print "Check for 1-bounded overtaking rule using deduction\n";
Let p := (P[1].loc = 2) & invc;
Let q[0] :=  (P[1].loc = 3) & invc ;
Let q[1] := !(P[2].loc = 3) & invc;
Let q[2] :=  (P[2].loc = 3)  & invc;
```

```
Let q[3] := !(P[2].loc = 3) & invc;

Let phi[0] := q[0];
Print "-- Calculate phi[1] = q[1] W q[0] ....";
Let phi[1] := invc & !predecessors(!q[0] & !q[1],
                                   total & q[1] & !q[0]);
Print "Done.\n\n";
Print "-- Calculate phi[2] = q[2] W phi[1] ....";
Let phi[2] := invc & !predecessors(!phi[1] & !q[2],
                                   total & q[2] & !phi[1]);
Print "Done.\n\n";
Print "-- Calculate phi[3] = q[3] W phi[2] ....";
Let phi[3] := invc & !predecessors(!phi[2] & !q[3],
                                   total & q[3] & !phi[2]);
Print "Done.\n\n";

Let phi[1] := phi[1] & !phi[0];
Let phi[2] := phi[2] & !phi[1] & !phi[0];
Let phi[3] := phi[3] & !phi[2] & !phi[1] & !phi[0];

Call nwait(p, q, phi, 4, success);
```

The entire computation is automatic and takes only a few seconds to complete.

Case Study 2: Checking 1-Bounded Overtaking for the RES-MP Resource Allocator Algorithm

Next we verified a parameterized resource sharing protocol RES-MP as shown in Fig. 3.10. The 1-bounded overtaking property states that

$$\forall i \neq j : at\_\ell_3[i] \Rightarrow \neg at\_\ell_4[j] \,\mathcal{W}\, at\_\ell_4[j] \,\mathcal{W}\, \neg at\_\ell_4[j] \,\mathcal{W}\, at\_\ell_4[i]$$

We implemented the same algorithm in TLV as was given in the Peterson's case for automatic generation of the auxiliary assertions. Then we use the heuristics in Fig. 3.8 to generate the auxiliary AER-assertions. Using the same deductive inference rule procedure `nwait`, we successfully proved the 1-bounded overtaking property for algorithm RES-MP.

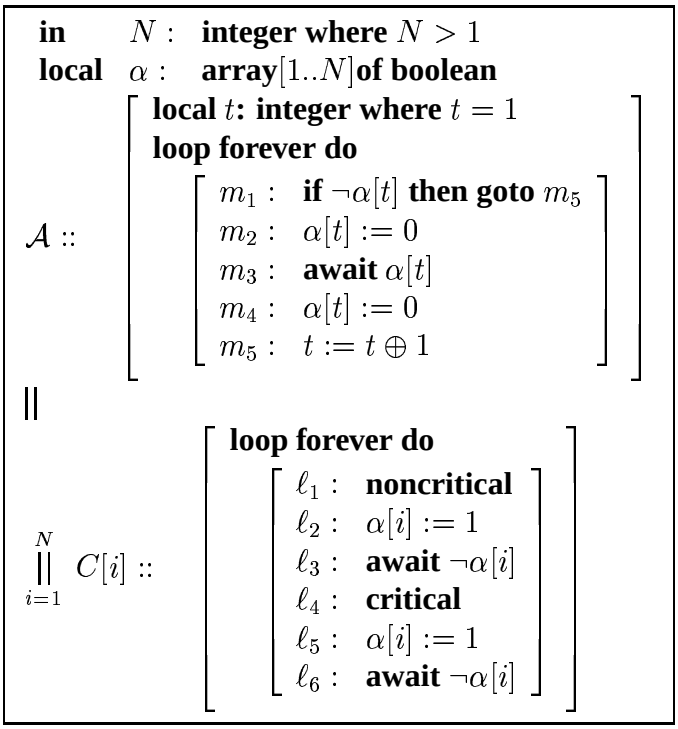


Figure 3.10: Program RES-MP with Asynchronous Shared Variables

3.4 Verification of General Safety Properties of Parameterized Systems

General safety properties can be expressed in the form $\Box\, p$ where p is a past formula. We consider two approaches for performing deductive verification of such formulas. The first approach builds a general tester for the negated safety formula and combines the tester with the system, then shows that the combined system is infeasible, i.e. that there is no valid computation for the combined system. This formulation can be summarized as follows:

- Let $\Box\, p$ be the safety property where p is a past formula. Build a tester T for the negation of the property, that is $\Diamond\, \neg p$.
- Synchronously compose the tester T with the original program $\mathcal{D}$ to obtain $\mathcal{D} \,|||\, T$.
- Prove that $\mathcal{D} \,|||\, T$ is infeasible, i.e., doesn't have a valid computation run.

The second approach builds a general tester for the past formula p in the safety property $\Box\, p$ and synchronously composes the tester with the original system, then applies the method of invisible invariants to check a safety property for the combined system. This formulation can be summarized as follows:

- Let $\Box\, p$ be the safety property where p is a past formula. Build a tester T for p.
- Synchronously compose the tester T with the original program $\mathcal{D}$.
- Show that $\mathcal{D} \,|||\, T$ satisfies the safety property $\Box\, \psi$ for some state formula ψ derived from p via the statification transformation (defined in the following section).

The first method can be applied to general temporal formulas, however it requires the use of fairness conditions in the proof. Therefore we choose to use the second method which doesn't require fairness conditions. It is a sound and complete method for proving general safety properties.

3.4.1 Construction of Temporal Testers

Let φ be a temporal formula with vocabulary U for which we wish to construct a temporal tester. A formula $p \in \varphi$ is called a *principally temporal subformula* if the main operator of p is temporal.

Let $\mathcal{T}(\varphi)$ denote the set of principally temporal subformulas of φ. Define a set of variables $X_\varphi : \{x_p | p \in \mathcal{T}(\varphi)\}$. We introduce a *statification transformation* χ, mapping subformulas of φ into boolean formulas over $U \bigcup X_\varphi$, as follows:

$$\chi(\psi) = \begin{cases} \psi & \text{for a state formula } \psi \\ \neg\chi(p) & \text{for } \psi = \neg p \\ \chi(p) \vee \chi(q) & \text{for } \psi = p \vee q \\ x_\psi & \text{for } \psi \in \mathcal{T}(\varphi) \end{cases}$$

The tester T_φ is given by

$$T_\varphi = \mathcal{D}_0 \,|||\, (|||_{\,\psi \in \mathcal{T}(\varphi)} \mathcal{D}[\psi]\,).$$

Since we are constructing the temporal tester for a past formula φ, we only need to consider the following basic cases:

– The FDS $\mathcal{D}[\psi]$ for $\psi = \ominus\, p$ is given by

$$\begin{array}{lcl} V & : & U \cup X_\psi \\ \Theta & : & \neg x_\psi \\ \rho & : & x'_\psi = \chi(p) \\ \mathcal{J} & : & \emptyset \\ \mathcal{C} & : & \emptyset \end{array}$$

– The FDS $\mathcal{D}[\psi]$ for $\psi = p\, \mathcal{S}\, q$ is given by

$$\begin{array}{lcl} V & : & U \cup X_\psi \\ \Theta & : & x_\psi = \chi(q) \\ \rho & : & x'_\psi = \chi(q)' \vee (\chi(p)' \wedge x_\psi) \\ \mathcal{J} & : & \emptyset \\ \mathcal{C} & : & \emptyset \end{array}$$

– The FDS $\mathcal{D}_0$ is given by

$$\begin{array}{lcl} V & : & U \cup X_\varphi \\ \Theta & : & \chi(\varphi) \\ \rho & : & 1 \\ \mathcal{J} & : & \emptyset \\ \mathcal{C} & : & \emptyset \end{array}$$

Other temporal testers can be derived from these basic forms.

Synchronous Parallel Composition of Fair Discrete Systems

Given two FDSs $\mathcal{D}_1 = \langle V_1, \Theta_1, \rho_1, \mathcal{J}_1, \mathcal{C}_1 \rangle$ and $\mathcal{D}_2 = \langle V_2, \Theta_2, \rho_2, \mathcal{J}_2, \mathcal{C}_2 \rangle$, the FDS $\mathcal{D} = \mathcal{D}_1 \,|||\, \mathcal{D}_2$ resulting from the synchronous composition of the two FDSs is defined to be:

$$\begin{aligned} V &= V_1 \cup V_2 \\ \Theta &= \Theta_1 \wedge \Theta_2 \\ \rho &= \rho_1 \wedge \rho_2 \\ \mathcal{J} &= \mathcal{J}_1 \cup \mathcal{J}_2 \\ \mathcal{C} &= \mathcal{C}_1 \cup \mathcal{C}_2 \end{aligned}$$

3.4.2 Invisible Invariants Method for General Safety Properties

For parameterized systems the general safety properties often assume the form of quantified temporal formulas, such as $\forall i, j : \Box\, \varphi(i, j)$. So we need to apply the following transformation:

$$\begin{aligned} \mathcal{D} \models \forall i, j : \Box\, \varphi(i, j) &\iff \forall i, j : (\mathcal{D} \models \Box\, \varphi(i, j)) \\ &\iff \forall i, j : (\mathcal{D} \,|||\, T_{\varphi(i,j)} \models \Box\, \chi_{\varphi(i,j)}) \end{aligned}$$

If the system is symmetric then we can arbitrary choose the indices i, j to be $1, 2$ and instead prove $\mathcal{D} \,|||\, T_{\varphi(1,2)} \models \Box\, \chi_{\varphi(1,2)}$. In general cases for rule INV we have two extra parameters i, j, which we need to take into consideration when calculating the cutoff value using the small model theorem.

Notice that the initial conditions and transition relations of the temporal testers only consist of unquantified assertions, thus the synchronous composition of a BDS system with the temporal testers should still satisfy the requirements of a BDS system to which we can apply small model theorems.

Example: Prove 1-Bounded Overtaking for Algorithm RES-MP

Here we apply the second approach for proving safety properties on algorithm RES-MP. The 1-bounded overtaking property we wish to prove is expressed by the following waiting-for formula:

$$\forall i,j : \Box(\ C[i].loc = 3 \longrightarrow \\ (C[j].loc \neq 4)\ \mathcal{W}\ (C[j].loc = 4)\ \mathcal{W}\ (C[j].loc \neq 4)\ \mathcal{W}\ (C[i].loc = 4))$$

which translates to the equivalent past formula:

$$\forall i,j : \Box(\ C[j].loc = 4 \longrightarrow \\ (C[j].loc = 4)\ \mathcal{B}\ (C[j].loc \neq 4)\ \mathcal{B}\ (C[i].loc \neq 3)\ \mathcal{B}\ (C[i].loc = 4)).$$

Since the system is symmetric, we can choose $i = 1, j = 2$. Let us define $q_1 = C[2].loc = 4$, $q_2 = C[2].loc \neq 4$, $q_3 = C[1].loc \neq 3$ and $r_3 = C[1].loc = 4$. We build temporal testers for the following past formulas:

1. $\mathcal{D}_3 = \mathcal{D}[q_3\ \mathcal{B}\ r_3]$
2. $\mathcal{D}_2 = \mathcal{D}[q_2\ \mathcal{B}\ (q_3\ \mathcal{B}\ r_3)]$
3. $\mathcal{D}_1 = \mathcal{D}[q_1\ \mathcal{B}\ (q_2\ \mathcal{B}\ (q_3\ \mathcal{B}\ r_3))]$

After synchronously composing the above testers with the original program

$$T = \mathcal{D}_0 \,|||\, \mathcal{D}_1 \,|||\, \mathcal{D}_2 \,|||\, \mathcal{D}_3$$

we can prove the invariance property $\Box\,\varphi(1,2) = \Box(q_1 \longrightarrow x_{q_1\ \mathcal{B}\ (q_2\ \mathcal{B}\ (q_3\ \mathcal{B}\ r_3))})$ for $\mathcal{D} \,|||\, T$, where $x_{q_1\ \mathcal{B}\ (q_2\ \mathcal{B}\ (q_3\ \mathcal{B}\ r_3))}$ is a statified variable representing the past formula $q_1\ \mathcal{B}\ q_2\ \mathcal{B}\ q_3\ \mathcal{B}\ r_3$.

3.5 Application of the Invisible Invariants Method to Clocked Systems

In order to apply the invisible invariant method to hardware verification we need to use real-time system models which can capture the metric aspect of time in a reactive system. The model must be able to measure the elapsed time between two events. *Clocked transition system* (CTS) is such a computational model for real-time systems [KMP00].

A CTS system $\Phi : \langle V, \Theta, \rho, \Pi \rangle$ consists of the following components:

- V: A finite set of system variables. The set $V = D \cup C$ is partitioned into $D = \{u_1, \ldots, u_n\}$ the set of *discrete variables* and $C = \{t_1, \ldots, t_k\}$ the set of *clocks*. Clocks always have the type *real*. The discrete variables can be of any type. We introduce a special clock $T \in C$, representing the *master clock*, as one of the system variables.

- Θ: The initial condition, an assertion (state formula) characterizing the set of initial states. It is required that

$$\Theta \rightarrow t_1 = \ldots = t_k = T = 0,$$

  i.e., all the clocks are set to zero at all initial states.

- ρ: A transition relation. An assertion $\rho(V, V')$, referring to both unprimed (present) and primed (next) versions of the state variables, relating a state $s \in \Sigma$ to its successor states. In case $T \in C$, it is required that $\rho \rightarrow T' = T$, i.e., the master clock is not modified by any process transition.

- Π: The time-progress condition. It is an assertion over V. The assertion is used to specify a global restriction on the progress of time.

Let $\Phi : \langle V, \Theta, \rho, \Pi \rangle$ be a CTS. We define the extended transition relation ρ_T associated with Φ as

$$\rho_T = \rho \vee \rho_{tick},$$

where ρ_{tick} is given by:

$$\rho_{tick} : \exists \Delta : \Omega(\Delta) \wedge D' = D \wedge C' = C + \Delta,$$

and $\Omega(\Delta)$ is given by

$$\Omega(\Delta) : \Delta > 0 \wedge \forall t \in [0, \Delta) : \Pi(D, C + t)$$

Let $D = \{u_1, \ldots, u_m\}$ be the set of discrete variables of Φ and $C = \{c_1, \ldots, c_k\}$ be the set of its clocks. Then, the expression $C' = C + \Delta$ is an abbreviation for $(c'_1 = c_1 + \Delta) \wedge \cdots \wedge (c'_k = c_k + \Delta)$, and $\Pi(D, C+t)$ is an abbreviation for $\Pi(u_1, \ldots, u_m, c_1 + t, \ldots, c_k + t)$.

A run of Φ is a finite or infinite sequence of states $\sigma : s_0, s_1, \ldots$ satisfying:

- Initiality: $s_0 \models \Theta$.
- Consecution: For each $j \in [0, |\sigma|)$ s_{j+1} is a ρ_T-successor of s_j.

A computation of Φ is an infinite run satisfying:

- Time Divergence: The sequence $s_0[T], s_1[T], \ldots$ grows beyond any bound.

A CTS Φ is called non-Zeno if every finite run of Φ can be extended into a computation. In many cases, the time-progress condition Π has the following special form $\Pi : \bigwedge_{i \in I}(p_i \rightarrow t^i < E_i)$ where the assertions p_i and E_i do not depend on the clocks and $t^i \in C$ is some clock.

Case Study: Program Fischer

In Fig. 3.11 we present Fischer's Mutual Exclusion Program in SPL language. Given a program P written in SPL language, the real-time version of P requires for each location l of P an upper and lower bound on the length of time during which an execution can stay at location l without taking a transition. These bounds are often denoted by $[L_l, U_l]$ for $0 \leq L_l \leq U_l \leq \infty$. Let's assume that each location has the same time bounds $[L_l, U_l]$. It can

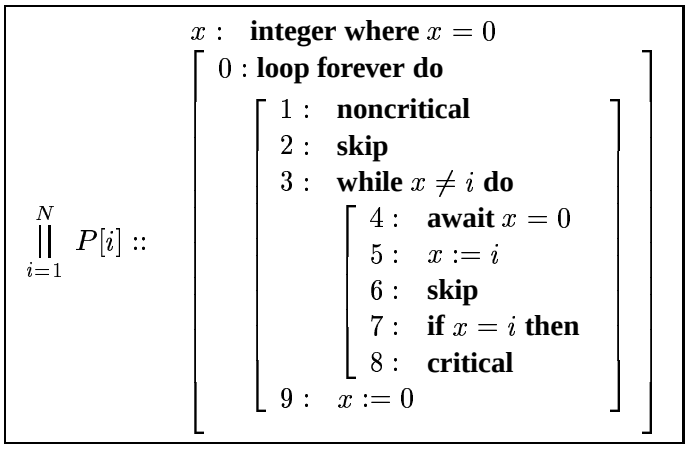


Figure 3.11: Fischer's Mutual Exclusion Algorithm

be shown that in order to have the program function correctly we must have $L_l \leq U_l \leq 2L_l$ to avoid race conditions. We implemented the algorithm using TLV as follows. Without loss of generality we choose $L = 3$ and $U = 5$. The following is an SMV program of Fischer with three processes. First we declare the time progress constraints `Low, High, Prog`, the global clock `T` and local clock array `t[1], ..., t[N]`, together with other program variables.

```
MODULE main
DEFINE
  N := 3;
  Low  := 3; -- The time bound is [Low, High] uniformly
  High := 5; -- require 2Low > High for the protocol to work
  Prog := P[1].prog & P[2].prog & P[3].prog;

VAR
  x:     0..N;
  t:     array 1..N of 0..6;
  T:     0..7;

  P[1]  : process proc(1,x,t[1],Low,High);
  P[2]  : process proc(2,x,t[2],Low,High);
  P[3]  : process proc(3,x,t[3],Low,High);
  Idle  : process MI;
  tick  : process Tick(Prog,t,T,Low,High);
```

Next we define process `proc` so that we can instantiate its parameters to obtain individual processes `P[1], P[2], P[3]`.

```
MODULE proc(id,x,my_t,Low,High)
DEFINE
  prog := my_t < High;  --Time progress condition for local clock

VAR
  loc: 1..9;

ASSIGN
  init(loc)   := 1;
  init(x)     := 0;
  init(my_t)     := 0;

  next(loc) :=  case
                  my_t < Low         : loc;
                  loc = 1            : {1,2};
                  loc in {2,5,6,9}   : (loc mod 9) + 1 ;
                  loc = 3 & x != id  : 4;
                  loc = 3            : 9;
                  loc = 4 & x = 0    : 5;
                  loc = 7 & x = id   : 8;
                  loc = 7            : 3;
                  loc = 8            : 3;
                  1                  : loc;
                esac;
  next(x) := case
                loc = 5 & next(loc) = 6    : id;
                loc = 9 & next(loc) != loc : 0;
                1                          : x;
             esac;

  next(my_t) := case
              my_t >= Low : 0; -- reset local clock
              1        : my_t;
             esac;
```

Last, we define the process `Tick` which advances the global clock `T` as well as the local clocks `t[1], t[2], t[3]`.

```
MODULE Tick(Prog,t,T,Low,High)

ASSIGN

  init(T)           := 0;
```

```
  next(t[1]) := case
                  Prog: t[1] + 1;
                  1   : t[1];
                esac;

  next(t[2]) := case
                  Prog: t[2] + 1;
                  1   : t[2];
                esac;

  next(t[3]) := case
                  Prog: t[3] + 1;
                  1   : t[3];
                esac;

  next(T) := case
                  Prog: (T + 1) mod 8;
                  1   : T;
              esac;


JUSTICE T = 0, T != 0
```

To generate the invariants with universal quantifiers in the form of $\forall i, j : \psi(i, j)$, we followed the previous heuristics by projecting the reachable states on indices $1, 2$ and then generalizing to the form $\forall i, j : \psi(i, j)$. Here is how we implement the procedure in TLV:

```
----- Calculate reach -------------
Print "Calculating reachable states.\n";
Let reach := successors(_i, total);

------Project on index 1, 2 -------------
Let proji1 := (next(P[1].loc) = P[1].loc)
            & (next(P[2].loc) = P[2].loc)
            & (next(x)=1 <-> x = 1) & (next(x) = 2 <-> x = 2)
            & (next(x) = 0 <-> x = 0) & (next(t[1]) = t[1])
            & (next(t[2]) = t[2]);
Let inv_12 := succ(proji1, reach);

------Generalize inv_12 to invc = \forall i, j: inv_ij ----
To gen_inv;
  Let invc := 1;
```

```
  Let i := 1;

  While (i <= N)
    Let proj1i := (next(P[i].loc) = P[1].loc)
                & (next(x) = i <-> x = 1)
              & (next(x) = 0 <-> x = 0) & (next(t[i]) = t[1]);
    Let j := 1;
    While (j <= N)
      If (j != i)
        Let proj1i2j := proj1i & (next(P[j].loc) = P[2].loc)
                       & (next(x) = j <-> x = 2)
                             & (next(t[j]) = t[2]);
        Let invc := invc & succ(proj1i2j, inv_12);
      End
      Let j := j + 1;
    End
    Let i := i + 1;
  End
End
```

Note that we didn't preserve the global variable T in our computation, the main reason is that since T doesn't participate when a process is making a move its value is always preserved. It also turned out the invariant we generated $invc$ is the same as the reachable states $reach$. This example demonstrates that the invisible invariant method is applicable to the real time models and that the only extra variables we need to take into consideration are the "local" clock variables.

In summary, parameterized verification using invisible invariants can be applied to BDS systems for proving general safety properties (and can be extended to liveness properties by adding fairness requirements). One important feature of this method is that the entire process is fully automatable. The heuristics for generation of invisible invariants provide automatic approaches to obtain important invariants not only in the context of parameterized verification but in the general context of formal methods.

Chapter 4
Parameterized Verification using Counter Abstraction

In this chapter we study an entirely different approach to the parameterized verification problem, the use of abstraction. *Abstraction* is a powerful method to deal with complexity issues in formal verification. It is natural to choose abstraction as a verification methodology for parameterized systems in the hope that it can reduce otherwise intractable problems into tractable ones that verify much simpler systems which, in the best cases, are finite state systems.

Generally speaking, abstraction works as follows: Given a concrete system P and an abstraction relation $\alpha \subseteq \Sigma_P \times \Sigma_{P^\alpha}$ which maps states in the concrete system P to states in the abstract system, a valid and safe abstraction of system P should satisfy the following requirements:

- For each computation $\sigma : s_0, s_1, \ldots$ of the concrete system P, there exists a computation $\sigma^\alpha : S_0, S_1, \ldots$ of the abstracted system P^α, such that $\alpha(s_i, S_i)$ holds for every $i = 0, 1, \ldots$.
- For a temporal property φ, there exists an (effectively desirable) corresponding abstracted temporal property φ^α such that if φ^α is P^α-valid then φ is P-valid.

If an abstraction can successfully reduce the parameterized original system to a finite state system then we can apply formal techniques such as model checking to prove temporal properties on the abstracted system.

Although there are many powerful abstraction paradigms, most of them suffer from the same weakness: the need for human ingenuity and assistance. The lack of automatic

abstraction methods has seriously hindered the use of abstraction-based methods in parameterized verification. This predicament motivates the search for abstraction methods that are *sound* and *automatic*. After inspecting various abstraction methods, we found that finitary abstraction requires less ad-hoc human assistance in comparison with other methods. For an FDS $\mathcal{D}$ and a property ψ, once the user defines a finitary abstraction mapping α from the variable domain V of a concrete FDS $\mathcal{D}$ to a finite abstract variable domain V_α, a safe abstraction "recipe" will automatically abstract the FDS $\mathcal{D}$ into a finite-state abstract FDS $\mathcal{D}^\alpha$ and the property ψ into a finitary abstract property ψ^α. It is guaranteed that $\mathcal{D}^\alpha \models \psi^\alpha$ implies that $\mathcal{D} \models \psi$ (but not vice versa). In the following sections we will present *counter abstraction*, a special finitary abstraction method which can be automated to perform parameterized verification.

4.1 The Method of Counter Abstraction

To better illustrate the ideas behind *counter abstraction*, we use a running example throughout our discussion of the method. The test case we chose is the parameterized program MUX-SEM with a semaphore y shared among N processes (see Fig. 2.1). The property we are most interested in proving is the liveness property which states that if a process is at location 1 then it will eventually have access to location 2, the critical section. As for all parameterized systems the explicit state transition graph for MUX-SEM grows exponentially with parameter N. From the state graph point of view, an effective abstraction maps a state in the original concrete system to an abstract state and compresses the original state graph to a much more compact finite state graph. For parameterized systems such an abstraction has to satisfy one additional requirement: an abstraction has to converge with the growth of the parameter N.

The basic idea of *counter abstraction* is to "count" the number of processes in each of the local program locations provided that all processes execute the same program. For each program location $0, 1, 2, 3$ in a local process we introduce the abstract variables $\kappa_0, \kappa_1, \kappa_2, \kappa_3$ respectively and keep the shared variable y in the abstract system. In order to make the ab-

straction finitary we keep in κ_i the number of processes currently executing at location ℓ_i, if this number is 0 or 1, and let $\kappa_i = 2$ for all bigger counts.

The effect of *counter abstraction* can be viewed through the following transformations of an explicit state graph of a concrete parameterized system $S(N)$ into the *counter-abstracted* state graph:

Stage-1 Compression

- Relabel the nodes in the concrete system by using the abstract variables in the abstract system.

  For example, for an instance of MUX-SEM, $S(5)$, all concrete states which represent that only one process is at location 2 (critical section) while the other processes are at location 0 (noncritical section), are relabeled by the same abstract state: $\kappa_0 = 4, \kappa_1 = 0, \kappa_2 = 1, \kappa_3 = 0, y = 0$.

- Merge all the nodes with the same label into a single node as follows:
  Any outgoing edge from an old node will become an outgoing edge from the new merged node. Any incoming edge to an old node node will become an incoming edge into the new node. Remove all the redundant edges so that between any given two nodes A and B there is at most one unidirectional edge from A to B.

Stage-2 Compression

- Apply further abstraction on the new abstract graph from Stage-1 by relabelling each of its nodes currently labeled using the abstract variables $\kappa_0, \kappa_1, \kappa_2, \kappa_3, y$ as follows: For any κ_l if $\kappa_l > 1$, reassign it to $\kappa_l = 2$, else keep its original value (either 0 or 1).
  A note on notation: Originally we choose to assign κ_l to ∞; however to be consistent with our implementation where we choose the number 2 to symbolically represent ∞, we will use 2 in place of ∞ in the sequel.

- Apply the same merge procedure as outlined in Stage-1 compression and obtain the final state transition graph of the abstract system.

For optimization we can combine the relabelling steps from the stage 1 and 2 compression into one single step. In the case of MUX-SEM the above transformations will converge to a finite state graph for the abstract system of the parameterized system $S(N)$ for $N \geq 5$. The abstract state transition graph is shown in Fig. 4.1. The edge label on a transition in the graph represents the number of transitions it corresponds to in the concrete system; and the final states with double circles represent that *some process is in the critical section*.

Counter Abstraction using Data Abstraction

An effective way to define counter abstraction is based on the finitary data abstraction method [KP00a].

Let $\mathcal{D} = \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C} \rangle$ be an FDS, and Σ denote the set of states of $\mathcal{D}$. Let $\alpha : \Sigma \to \Sigma_\alpha$ be a mapping of concrete states Σ into abstract states Σ_α. We say that α is a finitary abstraction mapping, if Σ_A is a finite set. We wish to derive the finite abstract system $\mathcal{D}^\alpha$ and the abstract property ψ^α. If we can model check that $\mathcal{D}^\alpha \models \psi^\alpha$ then we can infer that $\mathcal{D} \models \psi$.

For the case of parameterized systems we assume that each process has a finite number of locations L and there are a finite number b of shared finite-domain variables. For each local program location l we introduce an abstract variable, κ_ℓ. The abstract mapping $\mathcal{E}^a$ for counter abstraction is given by

$$\begin{array}{rcll} \kappa_\ell & = & \left\{ \begin{array}{ll} 0 & \forall i : [1..N] : \pi[i] \neq \ell \\ 1 & \exists i_1 : [1..N] : \pi[i_1] = \ell \ \wedge \ \forall i_2 \neq i_1 : \pi[i_2] \neq \ell \\ 2 & \text{otherwise} \end{array} \right\} & \text{for } \ell = 0, \ldots, L \\ Y_j & = & y_j & \text{for } j = 1, \ldots, b \end{array}$$

where $\pi[i]$ denotes the program counter for process i. Here $\kappa_\ell = 0$ if there are no processes at location ℓ, $\kappa_\ell = 1$ if there is exactly one process at location ℓ, and $\kappa_\ell = 2$ if there are two or more processes at location ℓ.

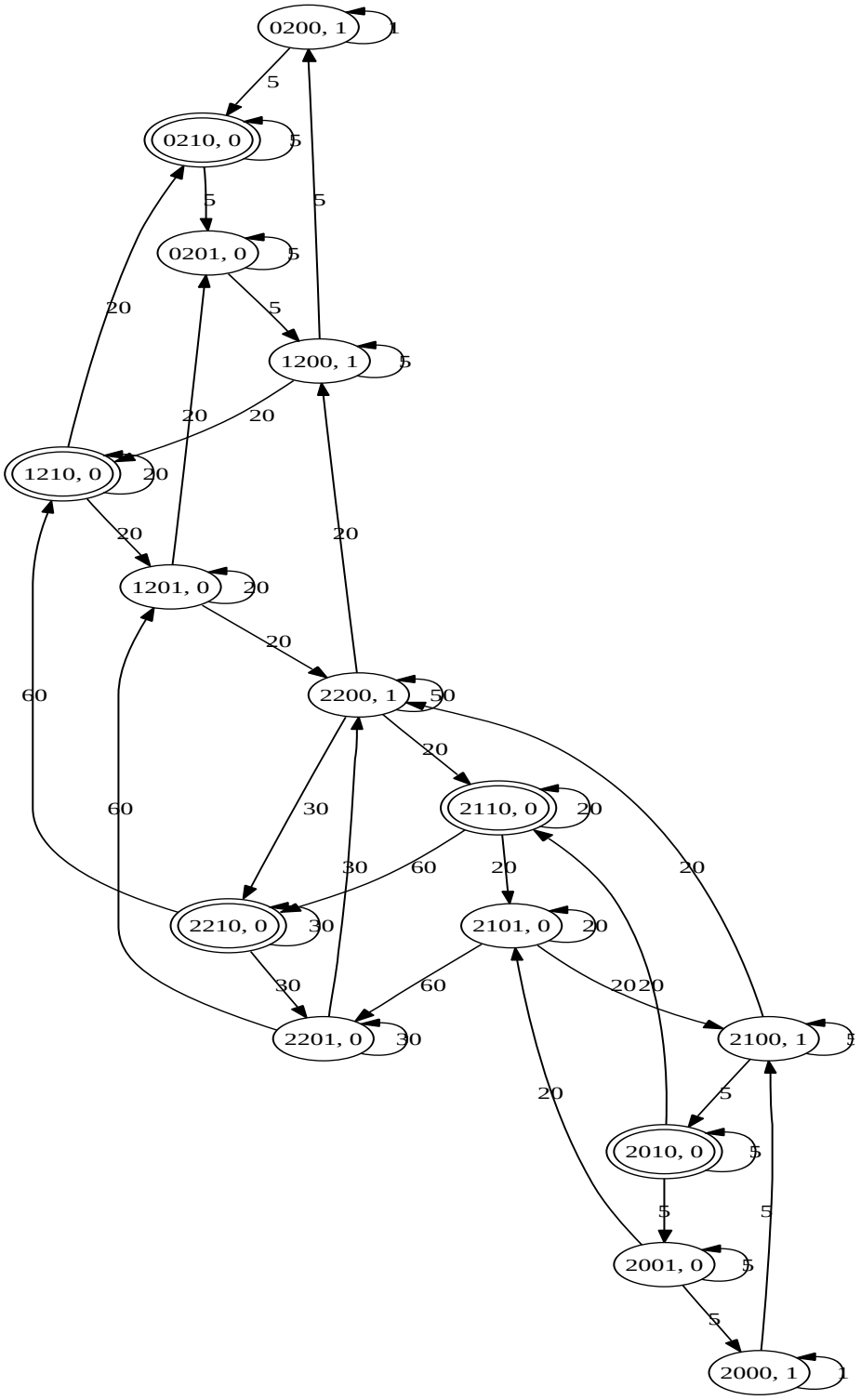


Figure 4.1: State Transition Graph of Abstract System MUX-SEM$^{\alpha}$ for $S(N)$ with $N \geq 5$

For example the abstract state of program MUX-SEM will have the form $\langle i_0\, i_1\, i_2\, i_3;\ \eta\rangle$, where $i_0, i_1, i_2, i_3 \in [0..2]$ are the values of $\kappa_0, \ldots, \kappa_3$, and $\eta \in \{0, 1\}$ is the value of y.

With the definition of the (counter) abstract mapping we can apply the safe data abstraction proposed in [KP00a] to calculate the abstract $\mathcal{D}^\alpha$.

– The initial condition is calculated as follows:

$$\Theta^\alpha = \alpha^+(\Theta) = \exists V : (V_A = \mathcal{E}^\alpha(V) \wedge \Theta(V))$$

For MUX-SEM where the initial condition for the concrete system is $(\forall i : at\_N[i]) \wedge (y > 0)$, the calculation yields $\Theta^\alpha = (\kappa_0 = 2) \wedge (\kappa_1 = 0) \wedge (\kappa_2 = 0) \wedge (\kappa_3 = 0) \wedge (y = 1)$ as the initial state $(2000; 1)$ in the abstract system.

– The transition relation is calculated as

$$\rho^\alpha = \alpha^{++}(\rho) = \exists V, V' : (V_A = \mathcal{E}^\alpha(V) \wedge V'_A = \mathcal{E}^\alpha(V') \wedge \rho(V, V'))$$

and results in the transition relation for the abstraction system:

$$\begin{aligned}
\rho^\alpha \;=\;& (\kappa_0 > 0) \wedge (\kappa'_0 = \kappa_0 \ominus 1) \wedge (\kappa'_1 = \kappa_1 \oplus 1) \wedge pres(\kappa_2, \kappa_3, y) \\
\vee\;& (\kappa_1 > 0) \wedge (y = 1) \wedge (\kappa'_1 = \kappa_1 \ominus 1) \wedge (\kappa'_2 = \kappa_2 \oplus 1) \wedge (y' = 0) \\
& \qquad \wedge pres(\kappa_0, \kappa_3) \\
\vee\;& (\kappa_2 > 0) \wedge (\kappa'_2 = \kappa_2 \ominus 1) \wedge (\kappa'_3 = \kappa_3 \oplus 1) \wedge pres(\kappa_0, \kappa_1, y) \\
\vee\;& (\kappa_3 > 0) \wedge (y = 0) \wedge (\kappa'_3 = \kappa_3 \ominus 1) \wedge (\kappa'_0 = \kappa_0 \oplus 1) \wedge (y' = 1) \\
& \qquad \wedge pres(\kappa_1, \kappa_2) \\
\vee\;& pres(\kappa_0, \kappa_1, \kappa_2, \kappa_3, y)
\end{aligned}$$

where the $\oplus$ and $\ominus$ operations are defined as follows:

$$\kappa_l \oplus 1 = \text{ if } \kappa_l < 2 \text{ then } \kappa_l + 1 \text{ else } 2$$

$$\kappa_l \ominus 1 = \text{ if } \kappa_l = 1 \text{ then } 0 \text{ else } \{1, 2\}$$

Note that $\ominus$ operation is partially defined on domain $\{1, 2\}$, i.e., excluding 0. And $2 \ominus 1$ produces the set of values $\{1, 2\}$.

– Abstracting the justice requirements as suggested in [KP00a]:

$$\mathcal{J}^\alpha = \{\alpha^+(J) | J \in \mathcal{J}\}$$

renders in the case of MUX-SEM: $\mathcal{J}^\alpha = \{\alpha^+(J_2[i]), \alpha^+(J_3[i]) \mid i \in [1..N]\}$ where

$$\alpha^+(J_2[i]) = \kappa_0 + \kappa_1 + \kappa_3 > 0, \qquad \alpha^+(J_2[i]) = \kappa_0 + \kappa_1 + \kappa_2 > 0$$

– Abstracting the compassion requirements using the recipe:

$$\mathcal{C}^\alpha = \{\langle \alpha^-(p), \alpha^+(q) \rangle | \langle p, q \rangle \in \mathcal{C}\}$$

gives us $\mathcal{C}^\alpha = \{\langle\, \alpha^-(p_1[j]), \alpha^+(q_1[j]) \,\rangle \mid i \in [1..N]\}$ where:

$$\alpha^-(p_1[j]) = y \wedge (\kappa_0 + \kappa_2 + \kappa_3 = 0), \qquad \alpha^+(q_1[j]) = \kappa_0 + \kappa_1 + \kappa_3 > 0$$

Unfortunately these abstract fairness conditions are too weak for proving the liveness property of MUX-SEM because the justice states include all the *reachable* states because all reachable states satisfy $\kappa_0 + \kappa_1 > 0$, thus admitting all viable runs as a legitimate computation, even those where there is one process "stuck" in the critical section (observable as a self loop on the state $(--1-, 0)$). In other words, any liveness property that is not valid for system MUX-SEM without the justice requirement $\mathcal{J}$ can not be proven by an abstraction which abstracts $\mathcal{J}$ into $\alpha^+(\mathcal{J})$as above. We need to develop stronger abstraction of fairness conditions in order to prove liveness properties.

4.2 Derivation of Abstract Justice Requirements

4.2.1 Justice Suppressing Assertions

The concept of *justice suppressing assertion* comes from the observation that for any abstract state s^α if we know all the concrete states that are mapped to s^α violate the same concrete justice requirement J then we can safely add the justice requirement $\neg s^\alpha$ to the abstract system. Here is the formal definition:

Definition 23 *Let φ be an abstract assertion, i.e., an assertion over the abstract state variables V_A. We say that φ **suppresses the concrete justice requirement** J if, for every two concrete states s_1 and s_2 such that s_2 is a ρ-successor of s_1, and both $\alpha(s_1)$ and $\alpha(s_2)$ satisfy φ, $s_1 \models \neg J$ implies $s_2 \models \neg J$.*

In the MUX-SEM example the abstract assertion $\varphi : \kappa_2 = 1$ suppresses the concrete justice requirement $J : \pi[i] \neq 2$. Here is the reasoning: Assume two states s_1 and s_2 such that s_2 is a successor of s_1 and both are counter-abstracted into abstract states satisfying $\kappa_2 = 1$. This implies that both s_1 and s_2 have precisely one process executing at location 2. If s_1 satisfies $\neg J = (\pi[i] = 2)$ then the single process executing at location 2 within s_1 must be $P[i]$. Since s_2 is a successor of s_1 and also has a single process executing at location 2, it must also be the same process $P[i]$, because it is impossible for $P[i]$ to exit location 2 and another process to enter the same location all within a single transition.

Definition 24 *The abstract assertion φ is defined to be **justice suppressing** if, for every concrete state s such that $\alpha(s) \models \varphi$, there exists a concrete justice requirement J such that $s \models \neg J$ and φ suppresses J.*

For example, the assertion $\kappa_2 = 1$ is justice suppressing, because every concrete state s whose counter-abstraction satisfies $\kappa_2 = 1$ must have a single process, say $P[i]$, executing at location 2. In that case, s violates the justice requirement $J : \pi[i] \neq 2$ which is suppressed by φ.

Theorem 25 (Safe Abstraction of Justice Requirements)
Let $\mathcal{D}$ be a concrete system and α be an abstraction that we apply to $\mathcal{D}$. Assume that

$\varphi_1, \ldots, \varphi_k$ is a list of justice suppressing assertions. Let $\mathcal{D}^\alpha$ be the abstract system obtained by following the data abstraction recipe described in Section 4.1 and then adding $\{\neg\varphi_1, \ldots, \neg\varphi_k\}$ to the set of abstract justice requirements. If $\mathcal{D}^\alpha \models \psi^\alpha$ then $\mathcal{D} \models \psi$.

Thus, we can safely add $\{\neg\varphi_1, \ldots, \neg\varphi_k\}$ to the set of justice requirement, while preserving the soundness of the method.

The proof of the theorem is based on the key observation that every abstraction of a concrete computation must contain infinitely many $\neg\varphi$-states for every justice suppressing assertion φ. Therefore, the abstract computations removed from the abstract system by the additional justice requirements can never correspond to any abstraction of concrete computations, and it is safe to remove them.

Theorem 25 is very general (not just restricted to counter abstraction) but it does not provide us with guidelines on how to generate justice suppressing assertions. For the case of counter-abstraction, we can have the following simple heuristics as our guidelines (later in this chaper we will give a more formal formulation of the generation of such assertions):

G1. If the concrete system contains the justice requirements $\neg(\pi[i] = \ell)$, then the assertion $\kappa_\ell = 1$ is justice suppressing.

G2. If the concrete system contains the justice requirements $\neg(\pi[i] = \ell \ \wedge \ c)$, where c is a condition on the shared variables (that are kept intact by the counter-abstraction), then the assertion $\kappa_\ell = 1 \ \wedge \ c$ is justice suppressing.

G3. If the concrete system contains the justice requirements $\neg(\pi[i] = \ell)$ and the only possible move from location ℓ is to location $\ell + 1$, then the two assertions $\kappa_\ell > 0 \ \wedge \ \kappa_{\ell+1} = 0$ and $\kappa_\ell > 0 \ \wedge \ \kappa_{\ell+1} = 1$ are justice suppressing.

G4. If the concrete system contains the justice requirements $\neg(\pi[i] = \ell \ \wedge \ c)$, where c is a condition on the shared variables, and the only possible move from location ℓ is to location $\ell+1$, then the assertions $\kappa_\ell > 0 \wedge \kappa_{\ell+1} = 0 \wedge c$ and $\kappa_\ell > 0 \wedge \kappa_{\ell+1} = 1 \wedge c$ are justice suppressing.

Example 1 According to the above guidelines we can add the following justice properties of MUX-SEM$^\alpha$. Since for MUX-SEM we have the justice $\neg(\pi[i] = 2)$, and since every move from location 2 leads to location 3, then by G1 and G2 the assertions $\kappa_2 = 1$, $\kappa_2 > 0 \ \wedge \ \kappa_3 = 0$, and $\kappa_2 > 0 \ \wedge \ \kappa_3 = 1$ are all justice suppressing and their negation can be added to $\mathcal{J}^\alpha$. Similarly, the justice requirement $\neg(\pi[i] = 3)$ leads to the justice suppressing assertions $\kappa_3 = 1$, $\kappa_3 > 0 \ \wedge \ \kappa_0 = 0$, and $\kappa_3 > 0 \ \wedge \ \kappa_0 = 1$. The concrete compassion requirement $\langle \pi[i] = 1 \ \wedge \ y, \pi[i] = 2 \rangle$ implies that the concrete assertion $\neg(\pi[i] = 1 \ \wedge \ y)$ is a justice requirement for system MUX-SEM. We can therefore add $\neg(\kappa_1 = 1 \ \wedge \ y)$ to the abstract justice requirement by G3. Since every move from location 1 leads to location 2, by G4 we can also add $\neg(\kappa_1 > 0 \ \wedge \ \kappa_2 = 0 \ \wedge \ y)$ to the abstract justice requirements.

In the next section we discuss the formal characterization of the abstract justice requirements which encompasses all the rules specified in the guidelines above.

4.2.2 Formal Characterization of Justice Suppressing Assertions

By the definition of justice suppressing assertions (Def. 24) we can characterize justice-suppressing assertions using the following characteristic functions.

Theorem 26 *Assertion φ suppresses justice requirement J if and only if it satisfies the assertion $suppj(\varphi, J)$:*

$$suppj(\varphi, J) = \forall V_1, V_2 : \varphi(\mathcal{E}^\alpha(V_1)) \wedge \varphi(\mathcal{E}^\alpha(V_2)) \wedge \rho(V_1, V_2) \wedge \neg J(V_1) \rightarrow \neg J(V_2).$$

Theorem 27 *An assertion φ is justice-suppressing if it and only if satisfies $supp(\varphi)$:*

$$supp(\varphi) = \forall V : \varphi(\mathcal{E}^\alpha(V)) \rightarrow \exists J : \neg J(V) \wedge suppj(\varphi, J).$$

Justice-suppressing assertions have a few interesting properties which are useful in forming such assertions.

Lemma 28 (anti-monotonicity) .
Given two assertions φ_1 and φ_2, if $\varphi_1 \rightarrow \varphi_2$, then $supp(\varphi_2) \rightarrow supp(\varphi_1)$.

Lemma 29 (monotonicity for disjoint unions) .
If two justice-suppressing assertions φ_1 and φ_2 are disjoint and there exists no transition that leads from a state in φ_1 to a state in φ_2 or vice versa, then their union $\varphi_1 \vee \varphi_2$ forms another justice-suppressing assertion. In other words, $supp(\varphi_1) \wedge supp(\varphi_2) \longrightarrow supp(\varphi_1 \vee \varphi_2)$.

In Lemma 29 it is crucial that there are no transitions that link the states between the two justice-suppressing assertions. The absence of this requirement could give rise to the following scenario: a concrete state s_1 violates only one justice requirement J and there is a transition from s_1 (which satisfies $\varphi_1(\alpha(s_1)) \wedge \neg J(s_1)$) to a concrete state s_2 satisfying $\varphi_2(\alpha(s_2)) \wedge J(s_2)$. Therefore $\varphi_1 \vee \varphi_2$ does not suppress the justice requirement J and fails the requirement of a justice suppressing assertion. Since the union of two justice suppressing assertions is not guaranteed to be justice suppressing, there might not exist a weakest justice suppressing assertion for a given system.

In some cases we can relax the requirement of two justice suppressing assertions being disjoint and still achieve the safe union of two assertions. For example, if two justice suppressing assertions φ_1, φ_2 suppress the same set of justice requirements and if any transition from a state in φ_1 to a state in φ_2 (and vice versa) keeps the same justice requirement suppressed, or more precisely, satisfies the following assertion (which can be viewed as an extension to the definition of $suppj(\varphi, J)$):

$$suppj(\varphi_1, \varphi_2, J) = \forall V_1, V_2 : \varphi_1(\mathcal{E}^\alpha(V_1)) \wedge \varphi_2(\mathcal{E}^\alpha(V_2)) \wedge \rho(V_1, V_2) \wedge \neg J(V_1) \rightarrow \neg J(V_2)$$

then the union of φ_1 and φ_2 is also a justice suppressing assertion.

Lemma 30 (monotonicity for safe unions) .
Given two justice-suppressing assertions φ_1 and φ_2, if the following conditions are satisfied:

- *Both φ_1 and φ_2 suppress the same set of justice requirements $\mathcal{J}^*$.*
- *For any $J \in \mathcal{J}^*$ we have $suppj(\varphi_1, \varphi_2, J)$ and $suppj(\varphi_2, \varphi_1, J)$.*

then their union $\varphi_1 \vee \varphi_2$ forms another justice-suppressing assertion, that is, $supp(\varphi_1) \wedge supp(\varphi_2) \longrightarrow supp(\varphi_1 \vee \varphi_2)$.

In particular, Lemma 29 can be viewed as a special case of Lemma 30.

Although there are no general formulas for computing justice-suppressing assertions, we still can derive many interesting justice suppressing assertions in special cases. The following theorem is an example of deriving a set of justice-suppressing assertions, each of which includes a single state.

Theorem 31 *The following assertion $JS(V_A, \mathcal{J})$ defines a set of abstract states such that each state s in the set constitutes a justice-suppressing assertion:*

$$JS(V_A, \mathcal{J}) = \forall V : V_A = \mathcal{E}^{\alpha}(V) \rightarrow (\exists J \in \mathcal{J} : \neg J(V) \wedge suppj(V_A, J))$$

According to Theorem 25, for each state in $JS(V_A, \mathcal{J})$ we can safely add $\neg s$ as a justice requirement to the abstract system. One simple consequence of the justice requirement $\neg s$ is the effective removal of all self-loops on abstract state s in $JS(V_A, \mathcal{J})$. A self-loop on abstract state s means there exists a computation in the concrete system such that every state s_i in the computation maps to s and $s_i \models \neg J$ for some justice requirement J because s is justice-suppressing. However, this shows that this computation is unjust, therefore there can't be any self-loop on s.

Using Lemma 29 we can combine the single states in $JS(V_A, \mathcal{J})$ which don't have any transitions among them into a single assertion when possible. Also, given any justice-suppressing assertion, we can always apply Lemma 28 to obtain a stronger assertion which is guaranteed to be justice-suppressing. As a consequence we may assume that any justice-suppressing assertion only contains the states that are in $JS(V_A, \mathcal{J})$.

We can derive stronger justice suppressing assertions using the components $\mathcal{J}_1, \ldots, \mathcal{J}_m$ of the concrete justice requirements $\mathcal{J} = \mathcal{J}_1 \bigcup \ldots \bigcup \mathcal{J}_m$ as shown in the following Lemma:

Lemma 32 *Let $\mathcal{J} = \mathcal{J}_1 \bigcup \ldots \bigcup \mathcal{J}_m$ be the set of concrete justice requirements. Define*

$$supp_{\mathcal{J}_k}(\varphi) = \forall V : \varphi(\mathcal{E}^\alpha(V)) \ \rightarrow \exists J \in \mathcal{J}_k : \neg J(V) \ \wedge \ supp(\varphi_k, J)$$

Then we claim $supp_{\mathcal{J}_k}(\varphi) \rightarrow supp(\varphi)$.

For program MUX-SEM, we can partition the justice requirements for the concrete system into three subgroups:

- $\mathcal{J}_2$: the justice requirements on location 2. That is, $\{\neg(\pi[i] = 2) \mid \forall i \in [1..N]\}$.
- $\mathcal{J}_3$: the justice requirements on location 3. That is, $\{\neg(\pi[i] = 3) \mid \forall i \in [1..N]\}$.
- $\mathcal{J}_\mathcal{C}$: the justice requirements derived from the compassion requirements $\mathcal{C}$. That is, $\{\neg(\pi[i] = 1 \wedge y > 0) \mid \forall i \in [1..N]\}$.

We can derive $JS(V_A, \mathcal{J}_\mathcal{C}), JS(V_A, \mathcal{J}_2), JS(V_A, \mathcal{J}_3)$ using Theorem 31:

1. $JS(V_A, \mathcal{J}_\mathcal{C}) = \forall V : V_A = \mathcal{E}^\alpha(V) \rightarrow \exists i : (\pi[i] = 1 \wedge y > 0) \wedge (\forall V_1, V_2 : V_A = \mathcal{E}^\alpha(V_1) \wedge$
   $V_A = \mathcal{E}^\alpha(V_2) \wedge \rho(V_1, V_2) \wedge (\pi[i] = 1 \wedge y_1 > 0) \rightarrow (\pi[i] = 1 \wedge y_2 > 0))$
2. $JS(V_A, \mathcal{J}_2) = \forall V : V_A = \mathcal{E}^\alpha(V) \rightarrow \exists i : \pi[i] = 2 \wedge (\forall V_1, V_2 : V_A = \mathcal{E}^\alpha(V_1) \wedge$
   $V_A = \mathcal{E}^\alpha(V_2) \wedge \rho(V_1, V_2) \wedge \pi[i] = 2 \rightarrow \pi[i] = 2)$
3. $JS(V_A, \mathcal{J}_3) = \forall V : V_A = \mathcal{E}^\alpha(V) \rightarrow \exists i : \pi[i] = 3 \wedge (\forall V_1, V_2 : V_A = \mathcal{E}^\alpha(V_1) \wedge$
   $V_A = \mathcal{E}^\alpha(V_2) \wedge \rho(V_1, V_2) \wedge \pi[i] = 3 \rightarrow \pi[i] = 3)$

It can be shown that all the states in $JS(V_A, \mathcal{J}_2)$ are connected through "safe" transitions, therefore $JS(V_A, \mathcal{J}_2)$ is justice-suppressing by Lemma 30. The same holds for $JS(V_A, \mathcal{J}_\mathcal{C})$ and $JS(V_A, \mathcal{J}_3)$. Finally we take their negations $\neg JS(V_A, \mathcal{J}_\mathcal{C}), \neg JS(V_A, \mathcal{J}_2), \neg JS(V_A, \mathcal{J}_3)$ as justice requirements and this turns out to be sufficient for proving liveness properties for the counter-abstracted system.

4.2.3 Proving Liveness Properties

Safety properties such as mutual exclusion for the counter-abstracted system usually can be checked easily by inspecting every state in the (finite) counter-abstracted graph (as shown in the state graph of MUX-SEM). The liveness property one usually associates with the parameterized systems is *individual accessibility*, such as $\forall i : \Box(\pi[i] = 1 \rightarrow \Diamond(\pi[i] = 2))$ in our example. Unfortunately, counter-abstraction does not allow us to observe the behavior of an individual process. Therefore, the property of individual accessibility cannot be expressed and verified in a counter-abstracted system. In Section 4.2.4 we show how to extend counter-abstraction to handle individual accessibility properties.

There are, however, liveness properties that *are* expressible and verifiable by counter abstraction. Such are *live-lock freedom* (or *communal accessibility*) properties of the form $\psi : \ \Box(\exists i : \pi[i] = 1 \rightarrow \Diamond(\exists i : \pi[i] = 2))$, stating that if *some* process is at location ℓ_1, then eventually *some* process (not necessarily the same) will enter 2. The counter-abstraction of such a property is $\psi^\alpha : \ \Box(\kappa_1 > 0 \rightarrow \Diamond(\kappa_2 > 0))$. Model checking that ψ^α holds over Sys^α can be accomplished by standard model checking techniques of response properties. E.g., the procedure in [LP85] suggests extracting from the state-transition graph the subgraph *pending states* and showing that it contains no infinite fair path. A pending state for a property $p \rightarrow \Diamond\, q$ is any state which is reachable from a p-state by a q-free path.

Example 2 Consider the system MUX-SEM$^\alpha$ of program MUX-SEM and the abstract live-lock freedom property $\chi : \Box(\kappa_1 > 0 \rightarrow \Diamond(\kappa_2 > 0))$. In Fig. 4.2, we present the subgraph of pending states for the property χ over the system MUX-SEM$^\alpha$.

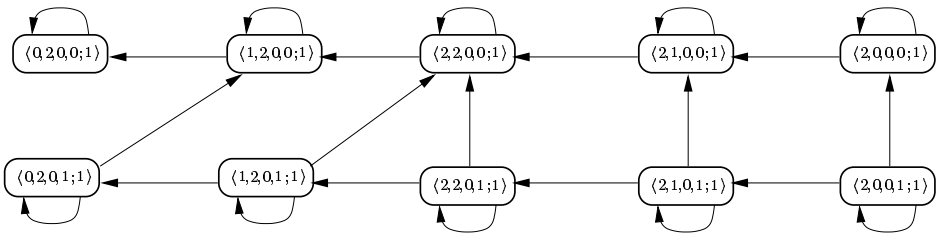


Figure 4.2: Reachability Graph for χ

To show that this graph contains no infinite fair path we decompose the graph into maximal strongly connected components and show that each of them is *unjust*. A strongly connected subgraph (SCS) S is unjust if there exists a justice requirement which is violated by all states within the subgraph. In the case of the graph in Fig. 4.2 there are ten maximal SCS's. Each of these subgraphs is unjust towards the abstract justice requirement $\neg(\kappa_1 > 0 \ \wedge \ y)$ derived in Example 1.

We conclude that the abstract property $\Box(\kappa_1 > 0 \rightarrow \Diamond(\kappa_2 > 0))$ is valid over system MUX-SEM$^\alpha$ and, therefore, the property $\Box(\exists i : \pi[i] = 1 \rightarrow \Diamond(\exists i : \pi[i] = 2))$ is valid over MUX-SEM.

4.2.4 Proving Individual Accessibility

As previously indicated, individual accessibility cannot be directly verified by standard counter abstraction because the abstraction cannot observe individual processes. To prove individual accessibility for the generic process $P[t]$, we abstract the system by counter abstracting all the processes except for $P[t]$, whom we leave intact. We then prove that the abstracted system satisfies the liveness property (the abstraction of which leaves it unchanged, since it refers to $\pi[t]$ that is kept intact by the abstraction), from which we derive that the concrete system satisfies it as well.

The new abstraction, which is "counter abstraction save one", is denoted by γ. As before, we assume for simplicity that the processes possess no local variables except for their program counter. The abstract variables for γ are given by $\kappa_0, \ldots, \kappa_L : [0..2], \ \Pi : [0..L]; \ Y_1, \ldots, Y_b$ and the abstraction mapping $\mathcal{E}^\gamma$ is given by

$$\kappa_\ell \ = \ \left\{ \begin{array}{ll} 0 & \forall r : [1..N] - \{t\} : \pi[r] \neq \ell \\ 1 & \exists r : [1..N] - \{t\} : \pi[r] = \ell \ \wedge \ \forall j \notin \{r, t\} : \pi[t] \neq \ell \\ 2 & \text{otherwise} \end{array} \right\} \quad \text{for } \ell = 0, \ldots, L$$

$$\Pi \ = \ \pi[t]$$

$$Y_k \ = \ y_k \qquad \text{for } k = 1, \ldots, b$$

We obtain ρ^γ as usual. For $\mathcal{J}^\gamma$, we include all the justice requirements obtained by the recipe of [KP00b] and the guidelines of subsection 4.2.1, along with all the requirements in $\mathcal{J}$ that relate to $P[t]$. For $\mathcal{C}^\gamma$ we take all the requirements in $\mathcal{C}$ that relate only to $P[t]$.

Without loss of generality we will show $\pi[0] = 1 \Rightarrow \diamondsuit\pi[0] = 2$. We change the compression/abstraction procedure to single out process 0 and then compress/abstract the rest of the system $S(N)$. The abstract transition graph (Fig. 4.3) converges after $N \geq 6$ (instead of $N \geq 5$ when proving the safety property). In the abstract graph the edge labels represent the number of transitions they correspond to in the concrete system; the square states represent the states when process 0 is in the trying stage; the target states marked by double ovals represent the states when process 0 enters the critical section.

To prove $\varphi : \square(\Pi = 1 \rightarrow \diamondsuit(\Pi = 2))$, consider the subgraph of $\mathcal{D}^\gamma$ that consists of all the abstract states that are reachable from a $(\Pi = 1)$-state by a $(\Pi = 2)$-free path, and show, as before, that this subgraph contains no infinite fair path.

Example 3 Consider the system MUX-SEM$^\gamma$ and the liveness property $\varphi^\gamma : \square((\Pi = 1) \rightarrow \diamondsuit(\Pi = 2))$. The subgraph of pending states is presented in Fig. 4.4. Each state in this graph is labeled by a tuple which specifies the values assigned by the state variables $\langle \kappa_0, \kappa_1, \kappa_2, \kappa_3; \Pi; Y \rangle$.

Unlike the previous case, this system has the compassion requirement $\langle \Pi = 1 \wedge Y, \Pi = 2 \rangle$ associated with $P[t]$. Since the subgraph contains no state satisfying $\Pi = 2$, no fair path can pass infinitely often through any state satisfying $\Pi = 1 \wedge Y$. Therefore, as a first step, we remove from the graph all states satisfying $\Pi = 1 \wedge Y$. This leads to the graph presented in Fig. 4.5.

This graph consists of ten maximal SCS's, each of which is unjust towards the abstract justice requirements $\neg(\kappa_2 = 1)$ or $\neg(\kappa_3 = 1)$, derived according to rule 1 of the guidelines of subsection 4.2.1.

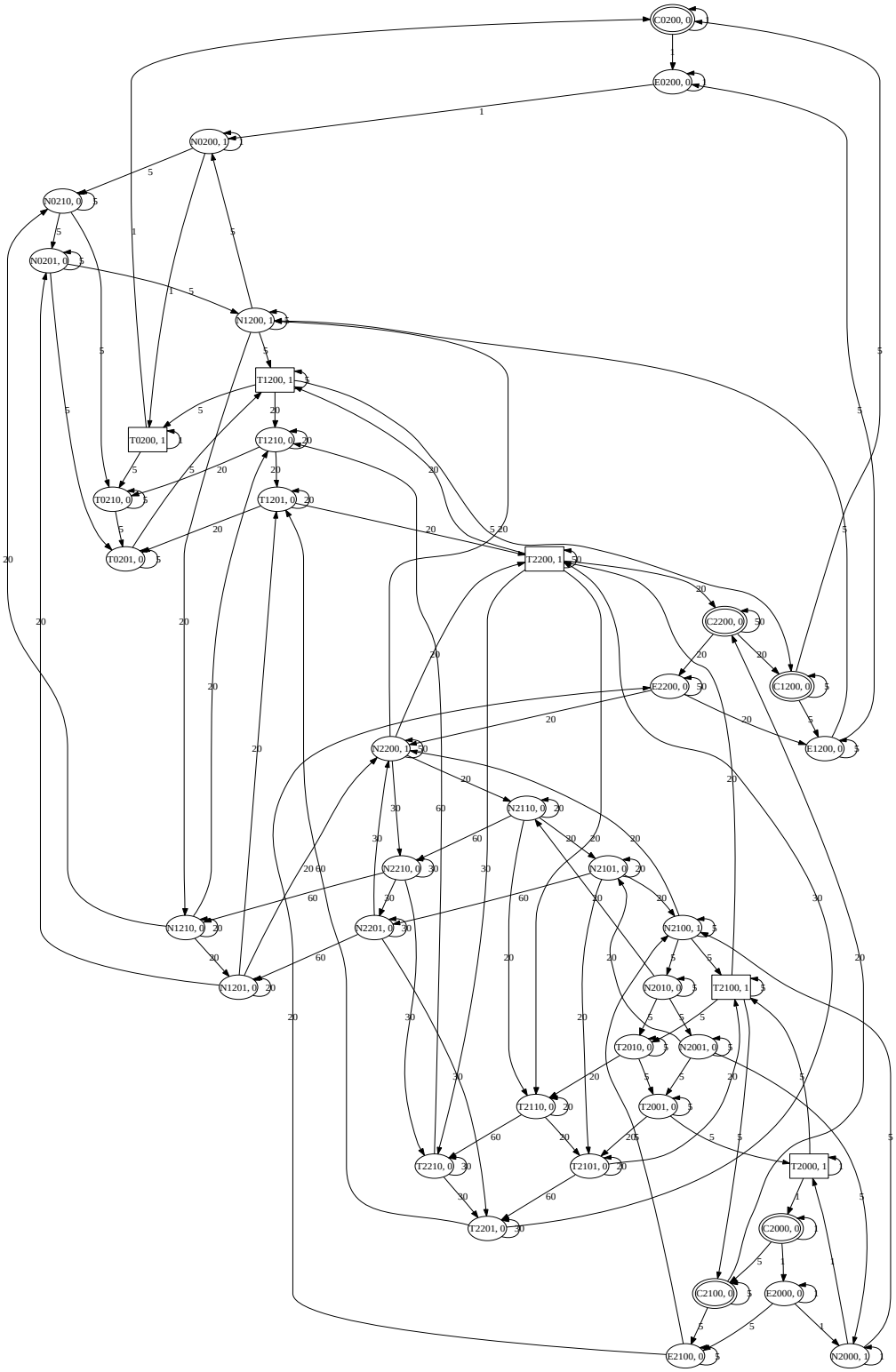


Figure 4.3: State Transition Graph of Abstract System MUX-SEM$^{\gamma}$ for $S(N)$ with $N \geq 6$

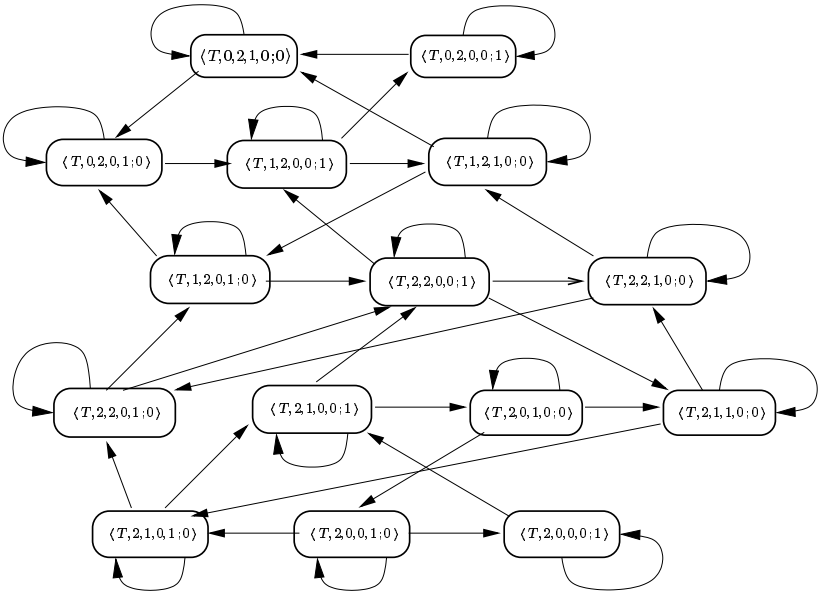


Figure 4.4: Pending States of MUX-SEM$^{\gamma}$ with Respect to φ

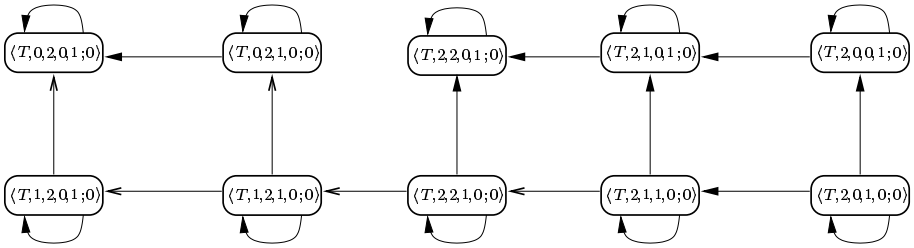


Figure 4.5: Pending states of MUX-SEM$^{\gamma}$ After Removal of All $(\Pi = 1 \wedge Y)$-states

We conclude that the abstract property $\Box((\Pi = 1) \rightarrow \Diamond(\Pi = 2))$ is valid over MUX-SEM$^{\gamma}$ and, therefore, $\Box((\pi[t] = 1) \rightarrow \Diamond(\pi[t] = 2))$ is valid over MUX-SEM.

4.3 Derivation of Abstract Compassion Requirements

In subsection 4.2.1 we showed how to derive additional justice requirements for a counter-abstracted system. We now turn to abstract compassion requirements that do not always correspond to concrete compassion requirements. Here we derive abstract compassion requirements which reflect well-founded properties of some of the concrete data domains. Consider program TERMINATE presented in Fig. 4.6.

$$\textbf{in} \quad N : \textbf{integer where } N > 1$$
$$\Big\|_{i=1}^{N} P[i] :: \quad \left[\begin{array}{ll} 0: & \textbf{skip} \\ 1: & \end{array}\right]$$

Figure 4.6: Program TERMINATE

Here the execution of the command **skip** does nothing but advancing the program counter, and the empty instruction at location 1 only allows stuttering, that is, process $P[i]$ will be stuck at location 1 forever. The liveness property we would like to establish for this program is given by the formula $\varphi : \diamondsuit(\forall i : \pi[i] = 1)$, stating that eventually, all processes reach location 1. The counter abstraction for all $S(N)$ where $N \geq 4$ is given by the following graph Fig. 4.7, representing system TERMINATE$^\alpha$:

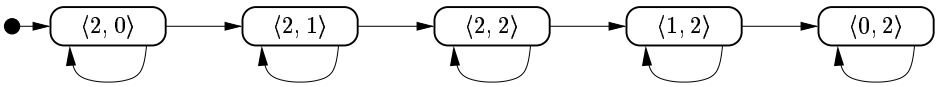


Figure 4.7: State Transition Graph of TERMINATE$^\alpha$ for $S(N)$ with $N \geq 4$

In this graph, each state s is labeled by the values state s assigns to the abstract variables κ_0 and κ_1. The abstracted property φ^α is given by $\diamondsuit(\kappa_0 = 0)$. However, this property is not valid over TERMINATE$^\alpha$, even when we take into account all the justice requirements derived according to subsection 4.2.1. These justice requirements force the computation to eventually exit states $\langle 2, 0\rangle$, $\langle 2, 1\rangle$, and $\langle 1, 2\rangle$, but they do not prevent the computation from staying forever in state $\langle 2, 2\rangle$.

To obtain a fairness requirement which will force the computation to eventually exit state $\langle 2, 2\rangle$ we augment the system with two additional abstract variables and a corresponding compassion requirement that governs their behavior.

Definition 33 *Let $Sys : \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C}\rangle$ be an* FDS *representing a concrete parameterized system, where the locations of each process are $[0..L]$. We define an* ***augmented system***

$\mathcal{D}^* : \langle V^*, \Theta^*, \rho^*, \mathcal{J}^*, \mathcal{C}^* \rangle$ *as follows:*

$$
\begin{array}{lcl}
V^* & : & V \cup \{\mathit{from}, \mathit{to} : [-1..L]\} \\
\Theta^* & : & \Theta \wedge (\mathit{from} = -1) \wedge (\mathit{to} = -1) \\
\rho^* & : & \rho \wedge \left(\begin{array}{lll} & \exists i : [1..N], \ell_1 \neq \ell_2 : [0..L] : & \pi[i] = \ell_1 \wedge \pi'[i] = \ell_2 \wedge \\ & & (\mathit{from}' = \ell_1) \wedge (\mathit{to}' = \ell_2) \\ \vee & \forall i : [1..N] : \pi'[i] = \pi[i] \wedge & (\mathit{from}' = -1) \wedge (\mathit{to}' = -1) \end{array} \right) \\
\mathcal{J}^* & : & \mathcal{J} \\
\mathcal{C}^* & : & \mathcal{C} \cup \{\langle \mathit{from} = \ell, \ \mathit{to} = \ell \rangle \mid \ell \in [0..L]\}
\end{array}
$$

Thus, system Sys is augmented with two auxiliary variables, *from* and *to*. Whenever a transition causes some process to move from location ℓ_1 to location $\ell_2 \neq \ell_1$, the same transition sets *from* to ℓ_1 and *to* to ℓ_2. Transitions that cause no process to change its location set both *from* and *to* to -1. For every $\ell \in [0..L]$, we add to Sys^* the compassion requirement $\langle \mathit{from} = \ell, \ \mathit{to} = \ell \rangle$. This compassion requirement represents the obvious fact that, since the overall number of processes is bounded (by N), processes cannot leave location ℓ infinitely many times without processes entering location ℓ infinitely many times.

Comparing the observable behavior of Sys and Sys^*, we can prove the following:

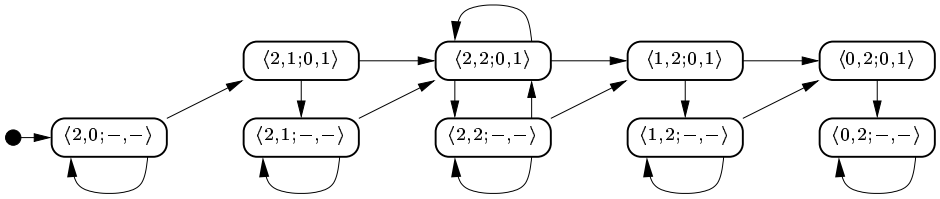
Theorem 34 *Let σ be an infinite sequence of V-states. Then σ is a computation of Sys iff it is a V-projection of a computation of system Sys^*.*

proof: Note that this compassion requirement is a global requirement on the entire parameterized system Sys^*. It is not hard to show that the V-projection of a computation of system Sys^* is a computation of Sys. For a computation σ of system Sys we can construct a computation σ^* in Sys^* by augmenting each state with the variables $from$ and to. We can show that this computation satisfies the compassion requirement $\langle from = i, to = i \rangle$ due to the boundedness of the finite number of processes N. So σ^* is a fair computation of Sys^*. It is obvious that the V-projection of σ^* is σ. Thus, the augmentation of Sys does not change its observable behavior. ⌟

Consequently, instead of counter-abstracting the system Sys, we can counter-abstract Sys^*. We denote by $Sys^\dagger = (Sys^*)^\alpha$ the counter abstraction of the augmented system Sys^*. In the presence of the auxiliary variable *from*, we can derive an even sharper justice requirement. We can replace the guidelines of subsection 4.2.1 by the following single rule:

G5. If the concrete system contains the justice requirements $\neg(\pi[i] = \ell \ \wedge \ c)$, where c is a condition on the shared variables and process indices other than i, then we may add to $Sys^\dagger$ the justice requirement $\mathit{from} = \ell \ \vee \ \neg(\kappa_\ell > 0 \wedge c^\alpha)$, where c^α is the counter abstraction of c.

Reconsider program TERMINATE. Applying the augmented abstraction, we obtain the following abstraction TERMINATE†, where each state s is labeled by the values s assigns to the variables $\langle \kappa_0, \kappa_1, \mathit{from}, \mathit{to} \rangle$ (symbol "$-$" stands for -1):



This system is augmented by the additional justice requirement $(\mathit{from} = 0) \ \vee \ (\kappa_0 = 0)$ (generated according to G5) and the compassion requirement $\langle \mathit{from} = 0, \ \mathit{to} = 0 \rangle$.

A detailed analysis of the SCSs in the above graph reveals that all of them with the exception of $\{\langle 0, 2; -, - \rangle\}$ are unfair. For instance, subgraphs $\{\langle 2, 0; -, - \rangle\}$, $\{\langle 2, 1; -, - \rangle\}$, and $\{\langle 1, 2; -, - \rangle\}$ violate the justice requirement $(\mathit{from} = 0) \ \vee \ (\kappa_0 = 0)$. And subgraph $\{\langle 2, 2; 0, 1 \rangle, \langle 2, 2; -, - \rangle\}$ violates the compassion requirement $\langle \mathit{from} = 0, \ \mathit{to} = 0 \rangle$. The standard remedy for this is to remove the state satisfying $\mathit{from} = 0$, namely $\langle 2, 2; 0, 1 \rangle$ this leaves us with the SCS $\{\langle 2, 2; -, - \rangle\}$ which violates the justice requirement $(\mathit{from} = 0) \ \vee \ (\kappa_0 = 0)$. The remaining SCS's ($\{\langle 2, 1; 0, 1 \rangle\}$, $\{\langle 1, 2; 0, 1 \rangle\}$, and $\{\langle 0, 2; 0, 1 \rangle\}$) are singleton subgraphs which are not connected to themselves so the fair run cannot visit them more than once.

It follows that every fair run traversing the subgraph of TERMINATE† must eventually reach state $\langle 0, 2; -, - \rangle$ and remain there forever. It follows that TERMINATE† satisfies $\diamondsuit(\kappa_0 = 0)$ and, therefore, that the concrete system TERMINATE satisfies $\forall i : \diamondsuit(\pi[i] \neq 0)$.

4.4 Implementing Counter Abstraction using TLV

In this section we describe how to implement counter abstraction using TLV, the programmable model checker from Weizmann Institute of Science in Israel. TLV is a verification tool set designed by Elad Shahar [PS96] based on the SMV system [McM92] with all the model checking algorithms removed. It provides an interpretive, weakly typed scripting language which uses BDDs to represent expressions in propositional logic. One great benefit of TLV is that it allows the user to program his own procedures and functions for formal verification tasks such as model checking and deductive verification using simple and intuitive scripting language. For this reason we choose it as our tool for the development of new verification paradigms. In the following sections we will describe some implementation details. For clarity we use a distinctive font to distinguish the TLV scripts from the rest of the text.

Implementing Counter Abstraction Mapping

Suppose we have the concrete system `CS` which consists of `Nproc` `MP` processes `P[1], ..., P[Nproc]` , and the shared variable `y`. Each process `MP` has `Nloc` locations.

```
CS: system conc-muxsem(Nloc,Nproc);
MODULE conc-muxsem(Nloc,Nproc)
VAR
  P : array 1..Nproc of process MP(y,Nloc);
  y : boolean;
```

We would like to use counter abstraction to obtain the abstract system `AS` with the shared variable `YY`.

```
AS: system abs-muxsem(Nloc);
MODULE abs-muxsem(Nloc)
```

```
VAR
  YY : boolean;
  C  : array 0..Nloc of 0..2;  -- abstract (counter) variables
```

To implement the abstract mapping we count the number processes in each location using the function `how-many`.

```
Func how-many(jloc);
  Local count := 0;
  For (k in 1..Nproc)
    Let count := count + (CS.P[k].loc=jloc);
  End
  Return count;
End
```

The procedure *prepare* is used to implement the abstract mapping and store it in the variable `abst`.

```
To prepare;

  --Load the TLV rule file for counter abstraction
  Load "auto-abs.tlv";

  Let abst := (YY = y); --here YY = AS.YY, y = CS.y
  For (i in 0..Nloc)
    Print "\n Computing abstraction of C[",i,"]\n";
    Let sum := how-many(i);
    Let abst := abst & (AS.C[i] = case
                                    sum=0 : 0;
                                    sum=1 : 1;
                                    1     : 2;
                                  esac);
  End
End -- To prepare;
```

Implementing Data Abstraction in TLV

In the sequel we present the main functions and procedures defined in the TLV rule file `auto-abs.tlv` for automatic counter abstraction (see Appendix 5.2 for the entire script). Before abstracting the concrete system, we need to first define “expanding” abstraction and

"contracting" abstraction in TLV. Let `vars1` be the set of present concrete variables, and `all_vars1` be the set of all present and next-state concrete variables. The function `abs-assert` implements the expanding α^+ abstraction for an assertion `phi`:

```
Func abs-assert(phi);
  Local result := (phi & abst) forsome vars1;
  Return result;
End
```

Similarly the contracting α^- abstraction is defined as:

```
Func cont-abs-assert(phi);
  Local result := !abs-assert(!phi);
  Return result;
End
```

The α^{++} abstraction for a transition relation `rho` is defined using function `abs-trans`:

```
Func abs-trans(rho);
  -- abst is the abstract mapping,
  -- and abstp is the prime (or future) version of abst
  Local result := (rho & abst & abstp) forsome all_vars1;
  Return result;
End
```

Now we can perform the data abstraction on the concrete FDS $\mathcal{D}$ to obtain the abstract FDS $\mathcal{D}^\alpha$. The procedure `abs-sys` abstracts the initial condition, the transition relation, the justice and compassion requirements of the concrete system `s[1]` into the corresponding components of the abstraction system `s[2]` according to the data abstraction recipe in [KP00b].

```
To abs-sys;
  Print "\n Start abstraction\n";
  Let vars1 := _s[1].v;
  Let all_vars1 := set_union(vars1,prime(vars1));
  Let abstp := prime(abst);

  --abstract the initial condition
  Let conci := _s[1].i;
  Let _s[2].i := abs-assert(conci);
```

```
  --abstract the transition relation, since the transition
  --relation is represented by tn disjunctive partitions, we
  --abstract each partition separately.
  Let _s[2].tn := _s[1].tn;
  For (i in 1.._s[1].tn)
    Print "\n Abstract transition t[",i,"]\n";
    Let conct := _s[1].t[i];
    Let _s[2].t[i] := abs-trans(conct);
  End -- For (i in 1.._s[1].tn)

  --abstract the justice requirements using KP formulation
  Let _s[2].jn := _s[1].jn;
  For (i in 1.._s[1].jn)
    Print "\n Abstract justice requirement J[",i,"]\n";
    Let concj := _s[1].j[i];
    Let _s[2].j[i] := abs-assert(concj);
  End

  --abstract the compassion using KP formulation
  Let _s[2].cn := _s[1].cn;
  For (i in 1.._s[1].cn)
    Print "\n Abstract compassion requirement PQ[",i,"]\n";
    Let _s[2].cp[i] := cont-abs-assert(_s[1].cp[i]);
    Let _s[2].cq[i] := abs-assert(_s[1].cq[i]);
  End
End -- To abs-sys;
```

Implementing Counter Abstraction of Justice Requirements

The justice and compassion requirements for the abstract system obtained using the above data abstraction method proposed in [KP00b] are too weak for proving accessibility in parameterized systems such as MUX-SEM. We need to strengthen the abstract justice requirements by computing justice suppressing assertions and then adding their negated forms to the abstract justice requirements. There are two possible methods for generating justice suppressing assertions.

The first one is by implementing heuristics G1 – G4:

– Scan all the justice requirements stored in `_s[1].t[]` for patterns $\neg(CS.P[i].loc =$

l) and $\neg(CS.P[i].loc = l \wedge c)$ where c is a condition on the shared variables.

- If we find a match in the first pattern, then we add the justice requirement $\neg(AS.C[l] = 1)$ by G1, and $\neg(AS.C[l] > 0 \wedge AS.C[l+1] < 2)$ by G3 if the only possible move from location l is to location $l+1$.

- If we find a match in the second pattern, then we add the justice requirement $\neg(AS.C[l] = 1 \wedge c)$ by G2, and $\neg(AS.C[l] > 0 \wedge AS.C[l+1] < 2 \wedge c)$ by G4 if the only possible move from location l is to location $l+1$.

The second method is to compute the candidates for justice suppressing assertions according to the formula provided in Theorem 31. We implement a function `calcJ` which takes the parameters `cjust` and `jcount` and calculates an assertion `aJ` which represents $JS(V_A, \mathcal{J}_{cjust})$. The parameter `cjust` picks the justice requirement $cjust$ in a concrete process. The parameter `jcount` counts the total number of justice requirements for a concrete process.

```
Func calcJ(cjust, jcount);
  Local accum := 0;
  Local curj := cjust;

  -- Combine the disjunctive partitions of the transition
  -- relation into a single one.
  Local trans_cur := 0;
  For (i in 1.._s[1].tn)
     Let trans_cur := trans_cur | _s[1].t[i];
  End

  -- Calculate JS(V_A, J_cjust)--------------------------
  -- The loop-k is used to cycle through the number cjust
  -- justice requirement for process 1..Nproc
  ------------------------------------------------------
  For (k in 1..Nproc)
   Local temp := trans_cur & abst & abstp & !(_s[1].j[curj])
                 & prime(_s[1].j[curj]);
   Let temp2 := !(temp forsome all_vars1);
   Let accum := accum | !(_s[1].j[curj]) & temp2;
   Let curj := curj + jcount;
  End
```

```
  Local aJ := !((abst & !accum) forsome vars1);
  Return aJ;
End -- Func CalcJ
```

Similarly we implement a function `calcJC` which takes the parameters `ccomp` and `ccount` and calculates an assertion `aJ` which represents $JS(V_A, \mathcal{J}_{\mathcal{C}_{ccomp}})$, where $\mathcal{J}_{\mathcal{C}_{ccomp}}$ represents the justice requirement obtained from the compassion requirement $ccomp$. The parameter `ccount` counts the total number of compassion requirements for a concrete process.

```
Func calcJC(ccomp, ccount);
  Local accum := 0;
  Local curc := ccomp;
  Local trans_cur := 0;

  For (i in 1.._s[1].tn)
    Let trans_cur := trans_cur | _s[1].t[i];
  End

  -- Calculate JS(V_A, J_(C_ccomp))----------------------
  -- The loop-k is used to cycle through the number ccomp
  -- compassion requirement for process 1..Nproc
  -------------------------------------------------------
  For (k in 1..Nproc)
   Local temp := 0;
   Let temp := (trans_cur & abst & abstp & (_s[1].cp[curc])
               & !prime(_s[1].cp[curc])) forsome all_vars1;

   Let accum := accum | (_s[1].cp[curc]) & !temp;
   Let curc := curc + ccount;
  End

  Local aJ := !((abst & !accum) forsome vars1);
  Return aJ;
End -- Func CalcJC
```

The assertions calculated using the functions `calcJ` and `calcJC` include the justice suppressing states, and often the assertions themselves are justice suppressing by Lemma 30 or Lemma 29. Adding these fairness conditions to the abstract system can be crucial for proving liveness properties of parameterized systems.

4.5 Examples

We applied the counter abstraction methods to several interesting parameterized systems for which few methods had succeeded proving their liveness properties.

4.5.1 Szymanski's Mutual Exclusion Algorithm

Our first test case is a slight variation from the mutual exclusion algorithm due to B. Szymanski [Szy88]. The original algorithm used two local boolean variables: s and w in each process. However, since the values of these variables can be determined by the location of the process owning them, it is possible to represent the algorithm using pure locations, which is the version presented in Fig. 4.8.

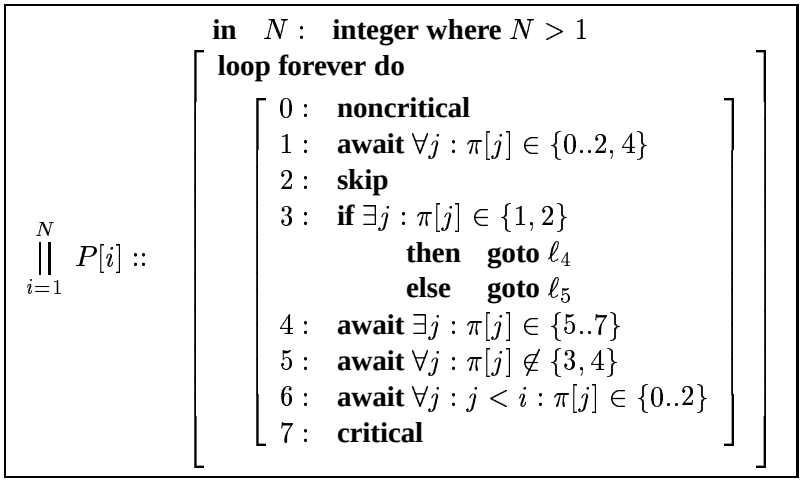


Figure 4.8: Parameterized Mutual Exclusion Algorithm SZYMANSKI

Strictly speaking, the algorithm is not symmetric since the instruction in location ℓ_6 depends on the id of the executing process i. Furthermore, the waiting-for condition in ℓ_6, $\forall j : j < i : \pi[j] \in \{0..2\}$ can not be tracked by counter abstraction. In order to express this condition in our abstracted system we need to introduce an additional abstract variable $lmin$ to our abstraction mapping. We define

$$lmin = \begin{cases} i & \forall j : (\pi[j] \geq 3 \rightarrow i \leq j) \\ 0 & \forall j : \pi[j] < 3 \end{cases}$$

This variable maintains the location of the process with the minimal index among those in $3..7$; *lmin* is 0 when there are no processes in locations $3..7$. Once the abstraction mapping of this variable is defined, the abstract transition relation is automatically computed using data abstraction. Without additional justice abstraction the counter abstracted system trivially establishes the safety property of mutual exclusion of the Szymanski algorithm as presented in Fig. 4.8.

In order to prove the live-lock freedom property $\chi : (\exists i : \pi[i] = 1) \rightarrow \diamondsuit(\exists i : \pi[i] = 7)$, szyman† contains the abstract justice requirements $\mathit{from} = \ell \vee \neg En(\tau)$ for each abstract transition τ departing from location $\ell \neq 0$, where $En(\tau)$ is the enabling condition for τ. Consider, for example, the transition corresponding to a process moving from location 6 to location 7. The enabling condition for the corresponding abstract transition is $\kappa_6 > 0 \ \wedge \ lmin = 6$. Consequently, we include in the abstract FDS the justice requirement $\mathit{from} = 6 \ \vee \ \neg(\kappa_6 > 0 \ \wedge \ lmin = 6)$. In addition, we included in the abstracted system the compassion requirements $\langle \mathit{from} = \ell, \mathit{to} = \ell \rangle$, for each $\ell = 0, \ldots, 7$. Using the abstraction szyman†, we were able to verify the property χ ensuring freedom from live-lock. We present the resulting abstract FDS szyman†, following the recipe of Section 4.3 in Appendix 5.2. Note that the auxiliary variable $lmin$ is essential only for the proof of mutual exclusion. Live-lock freedom can be established without this auxiliary variable.

To prove individual accessibility, we applied γ abstraction to szyman*. Some bookkeeping was needed to maintain the value of *lmin* to store the location of the non-t process whose index is minimal among those in locations $3..7$ (including t.)

4.5.2 The Bakery Algorithm

Consider a variant of Lamport's Bakery Algorithm for mutual exclusion presented in Fig. 4.9. An interesting feature of the algorithm is that the infinite data domain–processes choose "tickets" whose values are unbounded. To counter abstract bakery, we used a variable $lmin$, with a role similar to the one used in $szymanski$. Here $lmin$ is the location of the

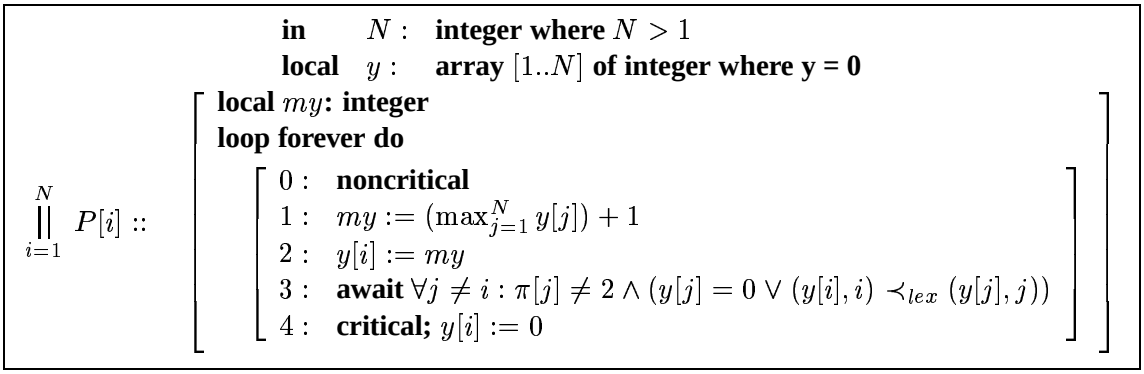


Figure 4.9: The Bakery Algorithm

process j whose $(y[j], j)$ is (lexicographically) minimal among those in locations $2..4$. The system bakery† is described in Appendix 5.2. The mutual exclusion property of the protocol, $\Box(\kappa_4 < 2)$, as well as the live-lock freedom property, $\Box(\kappa_1 > 0 \rightarrow \Diamond(\kappa_4 > 0))$ were easily established in TLV (the Weizmann Institute programmable model checker [PS96]). To establish the individual accessibility property, $\pi[t] = 1 \rightarrow \Diamond(\pi[t] = 4)$, we applied γ abstraction to bakery*. As in the case of $szymanski$, some bookkeeping was needed to maintain the value of $lmin$ to store the location of the non-t process whose index is minimal among those in locations $2..4$ (including t.)

Conclusions

In Table 4.1 we give the user run-time (in seconds) results of our various experiments. All verifications were carried out in TLV.

	MUX-SEM (sec)	szyman (sec)	bakery (sec)
Mutual Exclusion and Live-lock Freedom	0.02	0.69	0.12
Mutual Exclusion and Individual Accessibility (γ abstraction)	0.03	95.87	1.06

Table 4.1: Counter Abstraction Run Time Results

These results are encouraging because they demonstrate that a sound abstraction method, *counter abstraction*, can automatically abstract and prove safety as well as liveness properties of parameterized systems. In particular we present the derivation of additional fairness

conditions (both weak and strong) which enable us to perform automatic verification of interesting liveness properties of non-trivial parameterized systems. The abstraction and derivation are both automated using TLV, and this distinguishes this abstraction method from others. Further research performed by the ACSys group at NYU demonstrated the successful application of counter abstraction on some interesting probabilistic protocols [APZ03]. Still, applying this method to real world problems can hit complexity hurdles. For example if each process in the parameterized system consists of both a lot of locations and shared variables then the complexity of counter abstraction can quickly exceed the capacity of a software tool, such as TLV.

Chapter 5
Conclusion and Future Work

5.1 Conclusion

In the previous chapters we described automatic approaches toward the problem of parameterized verification. In the invisible invariants method we established the small model property for Bounded Data Systems and reduced parameterized verification into model-checking inference rules for small instances of BDS systems. In order to automate the whole process we proposed various heuristics to automatically generate inductive assertion candidates. This approach has been successfully applied to many well-known protocols to check safety properties ranging from invariance to waiting-for formulas. Further research done by the NYU Verification group has shown that by adding justice requirements to the BDS system, this approach can be successfully applied to liveness properties as well.

For stratified BDS systems the invisible invariant method is sound but incomplete. Thus, for certain correct safety properties we might not be able to generate an inductive assertion with desired logical form for the small model property. The major difficulty in the method is the automatic generation of the needed auxiliary inductive assertions. The heuristics we have proposed in this thesis cover formulas of simple logical forms. The heuristics don't guarantee the generation of the right inductive assertions, and failure in proving a safety property leaves two possible scenarios:

- The property is not true.
- We failed to generate the right set of inductive assertions.

Normally we first try to rule out the second scenario by using different heuristics to generate new assertions or by strengthening the assertions to make them inductive.

We have also extended the invisible invariant method to verify unstratified BDS systems, which is illustrated by verifying the safety property of Peterson's Mutual Exclusion Algorithm. We presented a sound method and demonstrated its successful application to Peterson's Algorithm.

Applying small model properties is a powerful strategy in parameterized verification because it often reduces the problem to finite-state domain and produces sound verification methods. In this thesis we use small model properties for deductive verification of parameterized systems. The bounds obtained from the small model properties allow us to apply efficient verification techniques such as model checking to discharge verification conditions in deductive methods. This mixed approach is powerful because the expressive power of deductive methods allows us to model and solve problems with an infinite state space and the model checking technique allows us to quickly and automatically discharge proof obligations in small instances. This is a perfect example of the appropriate combination of methods in formal verification to solve complex verification problems. The cutoff values computed using small model properties for BDS systems are sometimes compared with the cutoff values obtained using other methodologies. These comparisons are only meaningful in a restricted context because most small model theorems are only valid for a given verification method. Arguably the bound can reflect some intrinsic properties of the parameterized system itself and might suggest that the entire system behavior can be captured using small instances of the parameterized system.

The second approach, counter abstraction, defines a special "counter" abstraction relation which counts the number of processes at all program locations using three values $0, 1$ and $2 = \infty$. Safe data abstraction ([KP00b]) is performed on the concrete parameterized system to obtain a finite abstract system. Usually such an abstract system is sufficient for proving safety properties about resource-sharing; such as mutual exclusion. However, the abstract justice and compassion requirements obtained using the [KP00b] data abstraction are often too weak to prove liveness properties, such as communal accessibility. We strengthen the abstract justice requirements by adding the negation of justice-suppressing

assertions as new justice requirements. This effectively breaks many loop conditions on abstract states, especially many self-loops on a single state, and proves sufficient for proving liveness in many test cases. This justice-strengthening procedure is safe, it guarantees that the resulting new system is still a valid abstraction of the concrete system.

We also studied how to abstract the compassion requirements that are needed in proving many liveness properties. Like many other abstractions, counter abstraction will sometimes group and "collapse" too many concrete states into an abstract state and introduce loops due to the abstraction procedure itself and not any true computation. An augmented counter abstraction method is proposed by adding two auxiliary variables $to, from$ to record the location change of the current transition. This allows us to "split" some otherwise super abstract states and break unwanted loops introduced by the abstraction.

Counter abstraction can be used to prove "communal" properties about the entire system. In order to prove properties for an individual process we can preserve one single process in the system and counter abstract the rest of the system; this method works well in checking individual liveness properties.

5.2 Future Work

The method of invisible invariants has demonstrated an interesting automatic method for verifying parameterized systems. Heuristics developed for generating inductive assertions can be applied in the general deductive verification context. We should continue the study of developing heuristics for the automatic generation of inductive assertions for more complicated logical forms. Other approaches for generating inductive assertions should also be explored. Also we should put the entire method in a general automatic framework, which can automatically and systematically apply different heuristics to generate auxiliary assertions, and discharge the premises for various inference rules. Extension of this method to other temporal formulas is a very interesting and promising area to look into. Despite

our success in various test cases it is clear that we still need to work hard to contain the complexity issue. Our current tool set needs to incorporate more advanced reachability algorithms in order to tackle the complexity issue.

There are several interesting applications of counter abstraction, such as proving the liveness of Szymanski algorithm and the Bakery algorithm. It has been successfully applied in probabilistic verification as well. In many test cases, auxiliary variables are often introduced to encode program conditions that are necessary to guarantee an abstraction strong enough to prove the target properties. However, the addition of these auxiliary variables is often ad-hoc. We should look into techniques to automate these procedures. Besides the justice-suppressing assertions we should find other "safe" ways to abstract and strengthen the justice requirements. Abstracting compassion requirements is still mostly an open problem. Unlike justice, a lot of compassion requirements are global requirements which often require the use of additional auxiliary variables to encode some program conditions and to memorize the execution history, all of these being computationally expensive. These challenges must be addressed when considering the abstraction of compassion requirements. Finally, just as in the invisible invariant method we need a flexible general framework under which the automatic abstraction can be performed. It should be general enough to accommodate abstract relations other than counter abstraction and to provide a systematic way to strengthen the fairness requirements for proving the liveness properties.

In a word, the work has just begun.

Appendix A: Description of szyman†

The transition system szyman† : $\langle V, \Theta, \rho, \mathcal{J}, \mathcal{C}\rangle$ is defined as follows (Each statement of the form $(v'_1, \ldots, v'_k) := (E_1, \ldots, E_k)$ is an abbreviation for $v'_1 = E_1 \ \wedge \ \ldots \ \wedge \ v'_k = E_k \ \wedge \ \forall v \in V - \{v_1, \ldots, v_k\} : v' = v$.):

$$
\begin{array}{lcl}
V & : & \{\kappa_0, \ldots, \kappa_7 : [0..2],\ lmin : [0, 3..7],\ \mathit{from}, \mathit{to} : [-1..7]\} \\
\Theta & : & \kappa_0 = 2 \ \wedge \ \forall \ell \in \{1..7\} : \kappa_\ell = 0 \ \wedge \ lmin = 0 \ \wedge \ \mathit{from} = -1 \ \wedge \ \mathit{to} = -1 \\
\rho & : & (\kappa_0 := \kappa_0) \vee (\kappa_0 > 0 \ \wedge \ (\kappa_0, \kappa_1, \mathit{from}, \mathit{to}) := (\kappa_0 \ominus 1, \kappa_1 \oplus 1, 0, 1)) \\
& \vee & E_1 : (\kappa_1 > 0 \ \wedge \ \kappa_3 + \kappa_5 + \kappa_6 + \kappa_7 = 0) \wedge \\
& & \qquad (\kappa_1, \kappa_2, \mathit{from}, \mathit{to}) := (\kappa_1 \ominus 1, \kappa_2 \oplus 1, 1, 2) \\
& \vee & E_2 : (\kappa_2 > 0) \ \wedge \ (\kappa_2, \kappa_3, \mathit{from}, \mathit{to}, lmin) := (\kappa_2 \ominus 1, \kappa_3 \oplus 1, 2, 3, 3) \\
& \vee & E_2 : (\kappa_2 > 0) \ \wedge \ (\kappa_2, \kappa_3, \mathit{from}, \mathit{to}) := (\kappa_2 \ominus 1, \kappa_3 \oplus 1, 2, 3) \\
& \vee & E_{34} : (\kappa_3 > 0 \ \wedge \ \kappa_1 + \kappa_2 > 0) \ \wedge \ lmin \neq 3 \wedge \\
& & \qquad (\kappa_3, \kappa_4, \mathit{from}, \mathit{to}) := (\kappa_3 \ominus 1, \kappa_4 \oplus 1, 3, 4) \\
& \vee & E_{34} : (\kappa_3 > 0 \ \wedge \ \kappa_1 + \kappa_2 > 0) \ \wedge \ lmin = 3 \wedge \\
& & \qquad (\kappa_3, \kappa_4, \mathit{from}, \mathit{to}, lmin) := (\kappa_3 \ominus 1, \kappa_4 \oplus 1, 3, 4, 4) \\
& \vee & E_{34} : (\kappa_3 > 0 \ \wedge \ \kappa_1 + \kappa_2 > 0) \ \wedge \ lmin = 3 \ \wedge \ \kappa_3 > 1) \wedge \\
& & \qquad (\kappa_3, \kappa_4, \mathit{from}, \mathit{to}, lmin) := (\kappa_3 \ominus 1, \kappa_4 \oplus 1, 3, 4, 3) \\
& \vee & E_{35} : (\kappa_3 > 0 \ \wedge \ \kappa_1 + \kappa_2 = 0) \ \wedge \ lmin \neq 3 \wedge \\
& & \qquad (\kappa_3, \kappa_5, \mathit{from}, \mathit{to}) := (\kappa_3 \ominus 1, \kappa_5 \oplus 1, 3, 5) \\
& \vee & E_{35} : (\kappa_3 > 0 \ \wedge \ \kappa_1 + \kappa_2 = 0) \ \wedge \ lmin = 3 \wedge \\
& & \qquad (\kappa_3, \kappa_5, \mathit{from}, \mathit{to}, lmin) := (\kappa_3 \ominus 1, \kappa_5 \oplus 1, 3, 5, 5) \\
& \vee & E_{35} : (\kappa_3 > 0 \ \wedge \ \kappa_1 + \kappa_2 = 0) \ \wedge \ lmin = 3 \ \wedge \ \kappa_3 > 1 \wedge \\
& & \qquad (\kappa_3, \kappa_5, \mathit{from}, \mathit{to}, lmin) := (\kappa_3 \ominus 1, \kappa_5 \oplus 1, 3, 5, 3)
\end{array}
$$

$$
\begin{aligned}
\vee \quad & E_4 : (\kappa_4 > 0 \ \wedge \ \kappa_5 + \kappa_6 + \kappa_7 > 0) \ \wedge \ lmin \neq 4 \wedge \\
& \qquad (\kappa_4, \kappa_5, \mathit{from}, \mathit{to}) := (\kappa_4 \ominus 1, \kappa_5 \oplus 1, 4, 5) \\
\vee \quad & E_4 : (\kappa_4 > 0 \ \wedge \ \kappa_5 + \kappa_6 + \kappa_7 > 0) \ \wedge \ lmin = 4 \wedge \\
& \qquad (\kappa_4, \kappa_5, \mathit{from}, \mathit{to}, lmin) := (\kappa_4 \ominus 1, \kappa_5 \oplus 1, 4, 5, 5) \\
\vee \quad & E_4 : (\kappa_4 > 0 \ \wedge \ \kappa_5 + \kappa_6 + \kappa_7 > 0) \ \wedge \ lmin = 4 \ \wedge \ \kappa_3 > 2 \wedge \\
& \qquad (\kappa_4, \kappa_5, \mathit{from}, \mathit{to}, lmin) := (\kappa_4 \ominus 1, \kappa_5 \oplus 1, 4, 5, 4) \\
\vee \quad & E_5 : (\kappa_5 > 0 \ \wedge \ \kappa_3 + \kappa_4 = 0) \ \wedge \ lmin \neq 5 \wedge \\
& \qquad (\kappa_5, \kappa_6, \mathit{from}, \mathit{to}) := (\kappa_5 \ominus 1, \kappa_6 \oplus 1, 5, 6) \\
\vee \quad & E_5 : (\kappa_5 > 0 \ \wedge \ \kappa_3 + \kappa_4 = 0) \ \wedge \ lmin = 5 \wedge \\
& \qquad (\kappa_5, \kappa_6, \mathit{from}, \mathit{to}, lmin) := (\kappa_5 \ominus 1, \kappa_6 \oplus 1, 5, 6, 6) \\
\vee \quad & E_5 : (\kappa_5 > 0 \ \wedge \ \kappa_3 + \kappa_4 = 0) \ \wedge \ lmin = 5 \ \wedge \ \kappa_5 > 1 \wedge \\
& \qquad (\kappa_5, \kappa_6, \mathit{from}, \mathit{to}, lmin) := (\kappa_5 \ominus 1, \kappa_6 \oplus 1, 5, 6, 6) \\
\vee \quad & E_6 : (\kappa_6 > 0) \ \wedge \ lmin = 7 \ \wedge \ (\kappa_6, \kappa_7, \mathit{from}, \mathit{to}) := (\kappa_6 \ominus 1, \kappa_7 \oplus 1, 6, 7) \\
\vee \quad & E_7 : (\kappa_7 > 0) \ \wedge \ \kappa_3 + \kappa_5 + \kappa_6 + \kappa_7 = 1 \wedge \\
& \qquad (\kappa_7, \kappa_0, \mathit{from}, \mathit{to}, lmin) := \kappa_7 \ominus 1, \kappa_0 \oplus 1, 7, 0, 0) \\
\vee \quad & E_7 : (\kappa_7 > 0) \ \wedge \ \kappa_3 + \kappa_5 + \kappa_6 + \kappa_7 > 1 \ \wedge \ \kappa_3 > 0 \wedge \\
& \qquad (\kappa_7, \kappa_0, \mathit{from}, \mathit{to}, lmin) := \kappa_7 \ominus 1, \kappa_0 \oplus 1, 7, 0, 3) \\
\vee \quad & E_7 : (\kappa_7 > 0) \ \wedge \ \kappa_3 + \kappa_5 + \kappa_6 + \kappa_7 > 1 \ \wedge \ \kappa_4 > 0 \wedge \\
& \qquad (\kappa_7, \kappa_0, \mathit{from}, \mathit{to}, lmin) := \kappa_7 \ominus 1, \kappa_0 \oplus 1, 7, 0, 4) \\
\vee \quad & E_7 : (\kappa_7 > 0) \ \wedge \ \kappa_3 + \kappa_5 + \kappa_6 + \kappa_7 > 1 \ \wedge \ \kappa_5 > 0 \wedge \\
& \qquad (\kappa_7, \kappa_0, \mathit{from}, \mathit{to}, lmin) := \kappa_7 \ominus 1, \kappa_0 \oplus 1, 7, 0, 5) \\
\vee \quad & E_7 : (\kappa_7 > 0) \ \wedge \ \kappa_3 + \kappa_5 + \kappa_6 + \kappa_7 > 1 \ \wedge \ \kappa_6 > 0 \wedge \\
& \qquad (\kappa_7, \kappa_0, \mathit{from}, \mathit{to}, lmin) := \kappa_7 \ominus 1, \kappa_0 \oplus 1, 7, 0, 6) \\
\vee \quad & E_7 : (\kappa_7 > 0) \ \wedge \ \kappa_3 + \kappa_5 + \kappa_6 + \kappa_7 > 1 \ \wedge \ \kappa_7 > 1 \wedge \\
& \qquad (\kappa_7, \kappa_0, \mathit{from}, \mathit{to}, lmin) := \kappa_7 \ominus 1, \kappa_0 \oplus 1, 7, 0, 7)
\end{aligned}
$$

$$
\mathcal{J} \ : \ \left\{ \mathit{from} = \ell \ \vee \ \neg E_\ell \ \mid \ \ell \in \{1,2,4,5,6,7\} \right\} \ \cup \ \{\mathit{from} = 3 \ \vee \ \neg E_{34}\} \ \cup \ \{\mathit{from} = 3 \ \vee \ \neg E_{35}\}
$$

$$
\mathcal{C} \ : \ \bigcup_{\ell=0}^{7} \langle \mathit{from} = \ell, \ \mathit{to} = \ell \rangle
$$

Appendix B: Description of $\mathsf{bakery}^{\dagger}$

The transition system $\mathsf{bakery}^{\dagger}$: $\langle V, \Theta, \rho, \mathcal{J}, \mathcal{C}\rangle$ is defined as follows:

$$
\begin{array}{lcl}
V & : & \{\kappa_0, \ldots, \kappa_4 : [0..2],\ lmin : [0, 2..4],\ \mathit{from}, \mathit{to} : [-1..4]\} \\
\Theta & : & \kappa_0 = 2 \ \wedge\ \forall \ell \in \{1..4\} : \kappa_\ell = 0 \ \wedge\ lmin = 0 \ \wedge\ \mathit{from} = -1 \ \wedge\ \mathit{to} = -1 \\
\rho & : & (\kappa_0 := \kappa_0) \vee (\kappa_0, \kappa_1, \mathit{from}, \mathit{to}) := (\kappa_0 \ominus 1, \kappa_1 \oplus 1, 0, 1)) \\
 & & \quad \vee \quad E_1 : (\kappa_1 > 0) \ \wedge\ lmin \neq 0 \ \wedge\ (\kappa_1, \kappa_2, \mathit{from}, \mathit{to}) := (\kappa_1 \ominus 1, \kappa_2 \oplus 1, 1, 2) \\
 & & \quad \vee \quad E_1 : (\kappa_1 > 0) \ \wedge\ lmin = 0 \wedge \\
 & & \qquad\qquad (\kappa_1, \kappa_2, \mathit{from}, \mathit{to}, lmin) := (\kappa_1 \ominus 1, \kappa_2 \oplus 1, 1, 2, 2) \\
 & & \quad \vee \quad E_2 : (\kappa_2 > 0) \ \wedge\ lmin \neq 2 \ \wedge\ (\kappa_2, \kappa_3, \mathit{from}, \mathit{to}) := (\kappa_2 \ominus 1, \kappa_3 \oplus 1, 2, 3) \\
 & & \quad \vee \quad E_2 : (\kappa_2 > 0) \ \wedge\ lmin = 2 \wedge \\
 & & \qquad\qquad (\kappa_2, \kappa_3, \mathit{from}, \mathit{to}, lmin) := (\kappa_2 \ominus 1, \kappa_3 \oplus 1, 2, 3, 3) \\
 & & \quad \vee \quad E_2 : (\kappa_2 > 0) \ \wedge\ lmin = 2 \ \wedge\ \kappa_2 > 1 \wedge \\
 & & \qquad\qquad (\kappa_2, \kappa_3, \mathit{from}, \mathit{to}, lmin) := (\kappa_2 \ominus 1, \kappa_3 \oplus 1, 2, 3, 2) \\
 & & \quad \vee \quad E_3 : (\kappa_3 > 0 \ \wedge\ lmin = 3) \wedge \\
 & & \qquad\qquad (\kappa_3, \kappa_4, \mathit{from}, \mathit{to}, lmin) := (\kappa_3 \ominus 1, \kappa_4 \oplus 1, 3, 4, 4) \\
 & & \quad \vee \quad E_4 : (\kappa_4 > 0) \ \wedge\ \kappa_2 + \kappa_3 + \kappa_4 = 1 \wedge \\
 & & \qquad\qquad (\kappa_4, \kappa_0, \mathit{from}, \mathit{to}, lmin) := \kappa_4 \ominus 1, \kappa_0 \oplus 1, 4, 0, 0) \\
 & & \quad \vee \quad E_4 : (\kappa_4 > 0) \ \wedge\ \kappa_2 > 0 \wedge \\
 & & \qquad\qquad (\kappa_4, \kappa_0, \mathit{from}, \mathit{to}, lmin) := \kappa_4 \ominus 1, \kappa_0 \oplus 1, 4, 0, 2) \\
 & & \quad \vee \quad E_4 : (\kappa_4 > 0) \ \wedge\ \kappa_3 > 0 \wedge \\
 & & \qquad\qquad (\kappa_4, \kappa_0, \mathit{from}, \mathit{to}, lmin) := \kappa_4 \ominus 1, \kappa_0 \oplus 1, 4, 0, 3) \\
 & & \quad \vee \quad E_4 : (\kappa_4 > 0) \ \wedge\ \kappa_4 > 1 \wedge \\
 & & \qquad\qquad (\kappa_4, \kappa_0, \mathit{from}, \mathit{to}, lmin) := \kappa_4 \ominus 1, \kappa_0 \oplus 1, 4, 0, 4) \\
\mathcal{J} & : & \Big\{ \mathit{from} = \ell \ \vee\ \neg E_\ell \ \mid\ \ell \in [1..4] \Big\}_4 \\
\mathcal{C} & : & \bigcup_{\ell=0} \langle \mathit{from} = \ell,\ \mathit{to} = \ell \rangle
\end{array}
$$

Appendix C: TLV Rule File for Counter Abstraction

```
--  Auto-abs.tlv
--  Auxiliary file for abstracting an FDS
--

Func abs-assert(phi);
  Local result := (phi & abst) forsome vars1;
  Return result;
End -- Func abs-assert(phi);

Func cont-abs-assert(phi);
  Local result := !abs-assert(!phi);
  Return result;
End -- Func cont-abs-assert(phi);

Func abs-trans(rho);
  Local result := (rho & abst & abstp) forsome all_vars1;
  Return result;
End -- Func abs-assert(phi);

-----------------------------------------------------
-- Automatic Data Abstraction According to KP[00]
-----------------------------------------------------
To abs-sys;
  Print "\n Start abstraction\n";
  Let vars1 := _s[1].v;
  Let all_vars1 := set_union(vars1,prime(vars1));
  Let abstp := prime(abst);

  Let conci := _s[1].i;
  Let _s[2].i := abs-assert(conci);

  Let _s[2].tn := _s[1].tn;
  For (i in 1..._s[1].tn)
    Print "\n Abstract transition t[",i,"]\n";
    Let conct := _s[1].t[i];
    Let _s[2].t[i] := abs-trans(conct);
  End -- For (i in 1..._s[1].tn)

  Let _s[2].jn := _s[1].jn;
```

```
  For (i in 1..._s[1].jn)
    Print "\n Abstract justice requirement J[",i,"]\n";
    Let concj := _s[1].j[i];
    Let _s[2].j[i] := abs-assert(concj);
  End -- For (i in 1..._s[1].jn)

  Let _s[2].cn := _s[1].cn;
  For (i in 1..._s[1].cn)
    Print "\n Abstract compassion requirement PQ[",i,"]\n";
    Let _s[2].cp[i] := cont-abs-assert(_s[1].cp[i]);
    Let _s[2].cq[i] := abs-assert(_s[1].cq[i]);
  End -- For (i in 1..._s[1].cn)

End -- To abs-sys;

Func concs(st);
-- Concretize abstract state "st"
  Local r := (st & abst) forsome vars2;
  Return r;
End -- Func concs(st);

To check_counter;
-- Check for counter example. If there exists one, print concrete version
  Let vars2 := _s[2].v;
  If(exist(ce[1]))
    Print "\n Concrete counter-example follows:\n";
    Local L := length(ce);
    Let cce[1] := concs(ce[1]) & _s[1].i;
    Local ic := 1;
    While (ic<L)
      Local nxst := succ1(cce[ic],1) & concs(ce[ic+1]);
      If(nxst)
        Let ic := ic+1;
        Let cce[ic] := nxst;
      Else
        Break;
      End -- If(nxst)
    End -- While (ic<L)
    Let cce[ic] := fsat(cce[ic],vars1);
    Let jc := ic - 1;
    While (jc>0)
      Let cce[jc] := fsat(pred1(cce[jc+1],1) & cce[jc],vars1);
      Let jc := jc - 1;
```

```
    End -- While (jc>0)
    -- Print counter example.
    For (i in 1...ic)
      Print "\n---- State no. ", i," =\n", cce[i];
    End -- For (i in 1...ic)
  End -- If(exist(ce))
End -- To check_counter;

------------------------------------------------------------
--Functions for abstracting justice suppressing assertions
------------------------------------------------------------

Func calcJ(cjust, jcount);
  Local accum := 0;
  Local curj := cjust;

  -- Combine the disjuctive partitions of the transition
  -- relation into a single one.
  Local trans_cur := 0;
  For (i in 1.._s[1].tn)
     Let trans_cur := trans_cur | _s[1].t[i];
  End

  -- Calculate JS(V_A, J_cjust)--------------------------
  -- The loop-k is used to cycle through the number cjust
  -- justice requirement for process 1..Nproc
  ------------------------------------------------------
  For (k in 1..Nproc)
   Local temp := trans_cur & abst & abstp & !(_s[1].j[curj])
                 & prime(_s[1].j[curj]);
   Let temp2 := !(temp forsome all_vars1);
   Let accum := accum | !(_s[1].j[curj]) & temp2;
   Let curj := curj + jcount;
  End

  Local aJ := !((abst & !accum) forsome vars1);
  Return aJ;
End -- Func CalcJ

Func calcJC(ccomp, ccount);
  Local accum := 0;
  Local curc := ccomp;
  Local trans_cur := 0;
```

```
  For (i in 1.._s[1].tn)
    Let trans_cur := trans_cur | _s[1].t[i];
  End

  -- Calculate JS(V_A, J_(C_ccomp))----------------------
  -- The loop-k is used to cycle through the number ccomp
  -- compassion requirement for process 1..Nproc
  ------------------------------------------------------
  For (k in 1..Nproc)
   Local temp := 0;
   Let temp := (trans_cur & abst & abstp & (_s[1].cp[curc])
               & !prime(_s[1].cp[curc])) forsome all_vars1;

   Let accum := accum | (_s[1].cp[curc]) & !temp;
   Let curc := curc + ccount;
  End

  Local aJ := !((abst & !accum) forsome vars1);
  Return aJ;
End -- Func CalcJC
```

References

[ABJN99] P.A. Abdulla, A. Bouajjani, B. Jonsson, and M. Nilsson. Handling global conditions in parametrized system verification. In *CAV'99, LNCS 1633*, pages 134–145, 1999.

[AK86] K. R. Apt and D. Kozen. Limits for automatic program verification of finite-state concurrent systems. *Information Processing Letters*, 22(6), 1986.

[APR+01] T. Arons, A. Pnueli, S. Ruah, J. Xu, and L. Zuck. Parameterized verification with automatically computed inductive assertions. In *CAV'01, LNCS 2102*, pages 221–234, 2001.

[APZ03] T. Arons, A. Pnueli, and L. Zuck. Parameterized verification by probabilistic abstraction. In Andrew D. Gordon, editor, *FoSSaCS*, volume *2620* of *Lecture Notes in Computer Science*, pages 87–102, Springer, 2003.

[BBM95] N. Bjørner, I.A. Browne, and Z. Manna. Automatic generation of invariants and intermediate assertions. In 1st *Intl. Conf. on Principles and Practice of Constraint Programming*, volume 976 of *Lect. Notes in Comp. Sci.*, pages 589–623. Springer-Verlag, 1995.

[BCG86] M.C. Browne, E.M. Clarke, and O. Grumberg. Reasoning about networks with many finite state processes. In *Proc. 5th ACM Symp. Princ. of Dist. Comp.*, pages 240–248, 1986.

[BBLS00] K. Baukus, S. Bensalem, Y. Lakhnesche, and K. Stahl. Abstracting WS1S Systems to Verify Parameterized Networks. In S. Graf and M. Schwarzbach, editors, *TACAS'00*, volume 1785. Springer, 2001.

[BLS96] S. Bensalem, Y. Lakhnesche, and H. Saidi. Powerful Techniques for the Automatic Generation of Invariants. In *Proceedings of the 8th International Conference on Computer Aided Verification*, 1996.

[BLS01] K. Baukus, Y. Lakhnesche, and K. Stahl. Verification of parameterized protocols. *Journal of Universal Computer Science*, 7(2):141–158, 2001.

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

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th Annual Symposium on Principles of Programming Languages*. ACM Press, 1977.

[CE81] E. M. Clarke and E. A. Emerson. Design and Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of Programs: Workshop, Yorktown Heights, NY, May 1981*, volume 131 of *Lecture Notes in Computer Science*. Springer-Verlag, 1981.

[CEFJ96] E.M. Clarke, R. Enders, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. *Formal Methods in System Design*, 9(1/2), 8 1996.

[CFJ93] E.M. Clarke, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. In *C. Courcoubetis, ed. Proc. CAV '93*, vol. 697 LNCS, pp. 450-461, Springer-Verlag, 1993.

[CGJ95] E.M. Clarke, O. Grumberg, and S. Jha. Verifying parametrized networks using abstraction and regular languages. In *6th International Conference on Concurrency Theory (CONCUR'95)*, pages 395–407, 1995.

[CGL92] E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. In *Proc. of Principles of Prog. Lang.*, 1992.

[CLM98] E.M. Clark, D.E. Long, and K.L. McMillan. Compositional model checking. In *Proc. of the 4th Symp. on Logic in Computer Science*, pages 353–362, June 1989.

[CR00] S.J. Creese and A.W. Roscoe. Data independent induction over structured networks. In *Proc. of the Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'00)*, Las Vegas, June 2000. CSREA Press.

[EK00] E.A. Emerson and V. Kahlon. Reducing model checking of the many to the few. In *17th International Conference on Automated Deduction (CADE-17)*, pages 236–255, 2000.

[EN95] E. A. Emerson and K. S. Namjoshi. Reasoning about rings. In *Proc. 22th ACM Conf. on Principles of Programming Languages, POPL'95,* San Francisco, 1995.

[EN96] E.A. Emerson and K.S. Namjoshi. Automatic verification of parameterized synchronous systems. In *CAV'96, LNCS 1102*, 1996.

[ES96] E. A. Emerson and A. P. Sistla. Symmetry and model checking. *Formal Methods in System Design*, 9(1/2), 8 1996.

[ES97] E. A. Emerson and A. P. Sistla. Utilizing symmetry when model checking under fairness assumptions. *ACM Trans. Prog. Lang. Sys.*, 19(4), 1997.

[FPPZ04] Y. Fang, N. Piterman, A. Pnueli, and L. Zuck. Liveness with invisible ranking. In *Proc. of the* 5^{th} *International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume *2937* of *LNCS*, 2004.

[GS96] S. Graf and H. Saïdi Verifying Invariants Using Theorem Proving In *Proceedings of the 8th Conference on Computer-Aided Verification*, August 1996.

[GS92] S.M. German and A.P. Sistla. Reasoning about systems with many processes. *J. ACM*, 39:675–735, 1992.

[GS97] V. Gyuris and A. P. Sistla. On-the-fly model checking under fairness that exploits symmetry. In *CAV'97, LNCS 1254*, 1997.

[GW75] S.M. German and B. Wegbreit. A synthesizer of inductive assertions. In IEEE Trans. On Software Engineering, 1:68-75, March 1975.

[GZ98] E.P. Gribomont and G. Zenner. Automated verification of szymanski's algorithm. In B. Steffen, editor, *TACAS'98, LNCS 1384*, pages 424–438, 1998.

[ID96] C.N. Ip and D. Dill. Verifying systems with replicated components in Murφ. In *CAV'96, LNCS 1102*, 1996.

[JL98] E. Jensen and N.A. Lynch. A proof of burn's n-process mutual exclusion algorithm using abstraction. In *TACAS'98, LNCS 1384*, pages 409–423, 1998.

[JN00] B. Jonsson and M. Nilsson. Transitive closures of regular relations for verifying infinite-state systems. In *TACAS'00, LNCS 1785*, 2000.

[KM76] S. Katz and Z. Manna A heuristic approach to program verification. In *proc. 3rd Int. Joint Conf. on Artificial Intelligence*, Stanford, CA, 1976.

[KMM+97] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. In *CAV'97, LNCS 1254*, pages 424–435, 1997.

[KMP00] Y. Kesten, Z. Manna, and A. Pnueli. Verification of Clocked and Hybrid Systems. In *Acta Inf.*, 36(11):837–912, 2000.

[KP00a] Y. Kesten and A. Pnueli. Control and data abstractions: The cornerstones of practical formal verification. *Software Tools for Technology Transfer*, 2(1):328–342, 2000.

[KP00b] Y. Kesten and A. Pnueli. Control and data abstractions: The cornerstones of practical formal verification. *Software Tools for Technology Transfer*, 4(2):328–342, 2000.

[KP00c] Y. Kesten and A. Pnueli. Verification by finitary abstraction. *Information and Computation, a special issue on Compositionality*, 163:203–243, 2000.

[LHR97] D. Lesens, N. Halbwachs, and P. Raymond. Automatic verification of parameterized linear networks of processes. In *24th ACM Symposium on Principles of Programming Languages, POPL'97*, Paris, 1997.

[LP85] O. Lichtenstein and A. Pnueli. Checking that finite-state concurrent programs satisfy their linear specification. In *Proc. 12th ACM Symp. Princ. of Prog. Lang.*, pages 97–107, 1985.

[LS97] D. Lesens and H. Saidi. Automatic verification of parameterized networks of processes by abstraction. In *2nd International Workshop on the Verification of Infinite State Systems (INFINITY'97)*, 1997.

[MAB+94] Z. Manna, A. Anuchitanukul, N. Bjørner, A. Browne, E. Chang, M. Colón, L. De Alfaro, H. Devarajan, H. Sipma, and T.E. Uribe. STeP: The Stanford Temporal Prover. Technical Report STAN-CS-TR-94-1518, Dept. of Comp. Sci., Stanford University, 1994.

[McM92] K.L. McMillan. Symbolic Model Checking, An approach to the state explosion problem. PhD Thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, May 1992.

[McM98] K.L. McMillan. Verification of an implementation of Tomasulo's algorithm by compositional model checking. In *CAV'98, LNCS 1427*, pages 110–121, 1998.

[MP95] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, New York, 1995.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proceedings of the Eighteenth Annual Symposium on Foundations of Computer Science*, pages 46-57. IEEE, New York, *Theoretical Computer Science*, 13:45-60, 1981.

[PRZ01] A. Pnueli, S. Ruah, and L. Zuck. Automatic deductive verification with invisible invariants. In *TACAS'01*, volume 2031, pages 82–97, 2001.

[PS96] A. Pnueli and E. Shahar. A platform for combining deductive with algorithmic verification. In *CAV'96, LNCS 1102*, pages 184–195, 1996.

[PS00] A. Pnueli and E. Shahar. Liveness and acceleration in parameterized verification. In *CAV'00, LNCS 1855*, pages 328–343, 2000.

[QS81] J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in CÆSAR. In *Proc. of Fifth ISP*, 1981.

[Szy88] B. K. Szymanski. A simple solution to Lamport's concurrent programming problem with linear wait. In *Proc. 1988 International Conference on Supercomputing Systems*, pages 621–626, St. Malo, France, 1988.

[WL89] P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, volume 407 of *Lect. Notes in Comp. Sci.*, pages 68–80. Springer-Verlag, 1989.