ABSTRACT

Title of Dissertation: AUTOMATED SYMBOLIC ANALYSIS OF REACTIVE SYSTEMS

Tevfik Bultan, Doctor of Philosophy, 1998

Dissertation directed by: Professor Richard Gerber
Department of Computer Science

In recent years, there has been a surge of progress in automated verification methods based on state exploration. In areas like hardware design, these technologies are rapidly augmenting key phases of testing and validation. To date, one of the most successful of these methods has been symbolic model checking, in which large finite-state machines are encoded into compact data structures such as binary decision diagrams (BDDs) – and are then checked for safety and liveness properties.

However, these techniques have not realized the same success on software systems. One limitation is their inability to deal with infinite-state programs – even those with a single unbounded integer. A second problem is that of state-space explosion, i.e., the exponential blow-up suffered when systems have large numbers of variables and concurrent components.

In this dissertation, we report our progress on these problems. First, we present an integer-based model checker, which uses formulas composed of affine constraints, logical connectives, and quantifiers (Presburger formulas) as its underlying state representation. With this model, we can compactly represent huge (even unbounded) sets of integer states in multiple dimensions. Next, we present our composite analyzer, which combines the strengths of both BDD (for Boolean and enumerated types), and Presburger formula (for integers) representations. Using this technique, a system's states are encoded via both representations, where the choice depends on the variable types.

We also present conservative approximation techniques for addressing the undecidability of model checking on infinite-state systems. Combined with reachability analysis, conservative approximation techniques form a wide variety of heuristics for analyzing large, complex systems.

Finally, we show how we used these methods to verify the properties of some subtle, nontrivial infinite-state systems – the type of systems usually analyzed by hand or via testing. Specifically, we discuss our experiments from domains such as requirements engineering, protocol analysis, and concurrent programming.

AUTOMATED SYMBOLIC ANALYSIS OF REACTIVE SYSTEMS

by

Tevfik Bultan

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
1998

Advisory Committee:

Professor Richard Gerber, Chairman/Advisor
Professor John Gannon
Professor William Pugh
Professor Jeffrey K. Hollingsworth
Professor Joseph F. Já Já

DEDICATION

To my parents Şükran and Hayrettin Bultan

ACKNOWLEDGEMENTS

I would like to thank my advisor Rich Gerber for his strong support during this study. He has prevented me from getting lost in details by always putting things in perspective during our countless discussions. Also, he has given me the freedom to experiment with new ideas – this dissertation reports the ones which worked.

I would like to thank John Gannon, Jeff Hollingsworth and Bill Pugh for their constructive comments that helped me to improve the dissertation.

Special thanks go to Bill Pugh and the Omega Library team. Their tool was the enabling technology for the contributions reported in this work.

I would also like to thank Jeff Fischer – our discussions about model checking were very informative. I was also very lucky to use his extensive collection of papers and books on automated verification.

I would like to thank William Chan for his comments about this dissertation, and for all the insightful discussions we had.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

AUTOMATED SYMBOLIC ANALYSIS
OF REACTIVE SYSTEMS

Tevfik Bultan

August 3, 1998

This comment page is not part of the dissertation.

Typeset by LaTeX using the `dissertation` style by Pablo A. Straub, University of Maryland.

Chapter 1

Introduction

Computer systems are used extensively in safety critical applications. As recent examples show – e.g., the Ariane 5 explosion [Lio96], the Pathfinder control software failure [Wil97] – bugs in such systems can be very costly. It is crucial to get the designs right. With the high complexity of the computer systems being designed, there is an increasing need for automated analysis tools that can help designers in this task. In hardware design, traditional testing and simulation techniques are breaking down because of the sizes of the systems being developed. For software systems, new safety critical applications make correctness an increasingly important issue.

Automated analysis techniques help designers to uncover properties of systems. Such analysis can be used to improve the performance of a system (as in data flow analysis) or to check its correctness (as in type checking). In this dissertation we focus on correctness. The motivation is to provide designers with easy and fast feedback, so that the errors can be detected as early as possible.

The idea of proving correctness of computer systems goes back to the 60's. Floyd [Flo67] and Naur [Nau66] showed that by associating formal semantics to programs their properties can be proved. Later, the seminal work of Hoare

[Hoa69] presented an axiomatic definition of program execution. Hoare logic, and later its generalization by Dijkstra using weakest preconditions [Dij75], paved the way for automated generation of proofs about properties of programs. The most influential work in automated theorem proving was by Boyer and Moore [BM79, BM88]. Theorem provers are still an active research area both in hardware and software verification [BY96, KM97, ORS92, ORR$^+$96].

In the late 70's, Pnueli observed that the notions of partial correctness and termination, which dominated the earlier work in automated verification, were not suitable for characterizing properties of reactive systems [Pnu77]. Reactive systems are those which keep interacting with their environment without terminating. Most systems for which correctness is crucial are reactive systems. Typical examples are protocols, concurrent programs, operating systems and hardware systems. Temporal logics were introduced by Pnueli as a formalism for characterizing properties of such systems [Pnu77, Pnu81].

In recent years there has been notable progress in automated verification of temporal properties. Techniques based on state exploration demonstrated that automated verification of complex systems is feasible. Several reasons for this success are: (1) the development of powerful techniques such as model checking, which can efficiently verify safety and liveness properties [CE81, CES86, QS82, VW86]; (2) innovative new data structures such as binary decision diagrams (BDDs) that symbolically encode large sets of states in compact formats [BCM$^+$90, Bry86, CBM89, CMB90, McM93]; and (3) new ways of carrying out compositional and local analysis, to assuage the state explosion usually associated with concurrency [BFG96, CLM89, GL94, God94]. These results have been applied to a wide variety of systems including hardware systems, requirements specifications, and protocols. But when transition systems are not restricted to be finite, most of these techniques are no longer applicable, as they inherently depend on all underlying types being bounded. Also, general safety and liveness properties become undecidable for infinite transition systems.

In any computer system, variables are eventually mapped to finite representations. Thus, it might be argued that integer variables can be given a finite range, and programs can then be analyzed as finite-state machines – for example, using BDDs [BCM$^+$90, McM93]. For several reasons, however, this may not always be the best way to proceed. First, mapping integer variables and operations to their binary implementations may lead to highly inefficient static analysis. More importantly, one may wish to analyze an algorithm as an abstraction, and prove its correctness in a general sense for any implementation of integers.

Given a finite but very large transition system, an analysis technique which has a worst case complexity proportional to the size of the input transition system will eventually run out of memory when the worst case is realized. Complexity analysis of model checking for all interesting temporal logics shows that worst case complexity is at least linear in the size of the input transition system. I.e., although model checking finite state systems is decidable (as opposed to infinite state systems) for very large systems it can be intractable. Hence, from a practical stand point, techniques for analyzing infinite state systems are also applicable to very large finite state systems.

Since the late 70's there has been considerable research on static analysis of infinite state systems. Abstract interpretation – a theoretical framework for static analysis – investigates the issue of computing fixpoints on infinite lattices [CC76, CC77, CC92a, CC92b]. State space exploration techniques used in most model checkers correspond to fixpoint computations. Hence, it is possible to adopt the abstract interpretation techniques for model checking. Recently, increasingly more researchers are observing this connection between automated verification and static analysis [DGG97, HH95, Sch98]. With this insight, we claim that one does not have to either prove the programs "correct" or give up all the hope of automated verification. The methods from static analysis and automated verification form a spectrum of tools that could be used incrementally in analyzing computer systems.

In this dissertation our goal is to extend the applicability of model checking to software systems. First, we concentrate on model checking systems with infinite integer domains. We use formulas composed of affine constraints, logical connectives and quantifiers (i.e., Presburger formulas) as a symbolic representation. This enables us to encode unbounded integer variables. Exploiting the abovementioned connection between model checking and abstract interpretation, we present conservative approximation techniques for computing fixpoints on infinite transition systems. Our results show that model checking is feasible for infinite domains – it is not limited to finite state systems as is traditionally assumed [CW96]. We also combine different symbolic representations in one composite model checker using type-specific encodings. To the best of our knowledge, this is the first such effort. This enables us to analyze multi-typed systems efficiently.

To demonstrate our results, we developed a prototype analysis tool (Appendix A) and analyzed several example systems. A summary of our automated analysis approach is shown in Figure 1.1, and it depicts how we structure the three main phases of translation, partitioning and analysis.

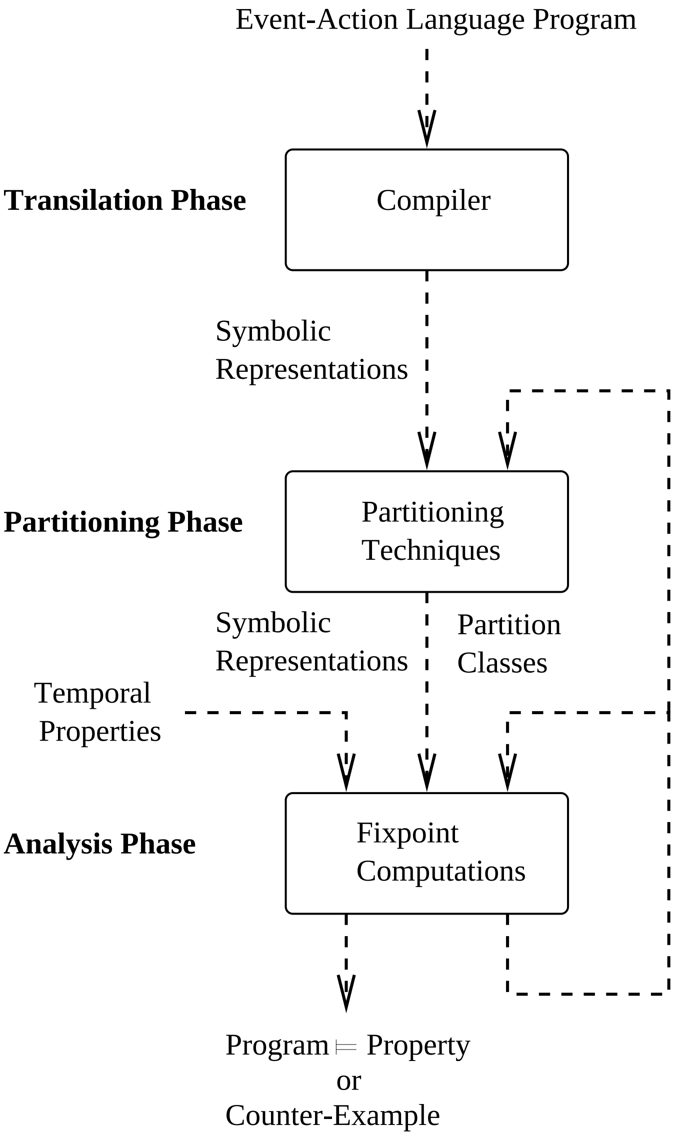


Figure 1.1: Overview of Our Analysis Approach

In the **translation phase**, our system accepts a program written in a simple event action language as input; then it produces a symbolic representation of the program's underlying transition relation. This symbolic representation consists of Boolean and Presburger formulas.

In the **partitioning phase**, the program's states and transitions are segregated into a set of partition classes, with the objective of reducing the complexity of verification and improving the accuracy of our approximation techniques. During the fixpoint computations, the number of constraints in our state representations could grow unmanageable. By grouping these constraints into partition classes, we simplify their manipulation. Also we eliminate variables with small domains by partitioning the state space based on their valuations. These decompositions do not alter the underlying transition relation of the original program; hence, they preserve all the temporal properties of the system. Partitioning is also useful in increasing the precision of our conservative approximation techniques.

In the **analysis phase**, fixpoint computations are used to prove (or disprove) temporal assertions about the input program. We use exact, approximate, forward or backward fixpoint computations, which can be combined in various ways to prove a single property. Such high-level strategy decisions are left to the user. In the following chapters, we demonstrate situations in which human input can help; specifically, we show a set of programs (and properties) which helped reveal the relative strengths and weaknesses of each approach.

1.1 Summary of Contributions

Below we list some of the significant contributions in this dissertation.

- We developed a model checker that symbolically encodes transition relations and sets of states using affine constraints on integer variables, logical connectives and quantifiers (i.e., Presburger formulas). The model checker derives truth sets of temporal logic formulas by computing the corresponding fixpoints using this representation. We developed partitioning techniques for efficient computation of the fixpoints. Presburger encoding enables us to analyze systems with unbounded integer domains.

- We developed a composite model checker that uses two different symbolic representations together. BDDs have been very successful in encoding Boolean domains, whereas our Presburger encoding enables us to analyze systems with unbounded integer variables. Our composite model checker exploits strengths of both BDD and Presburger representations. Our method forms a general framework for combining multiple type-specific symbolic representations in a single model checker.

- We developed a framework for conservatively approximating temporal properties. This enables us to analyze large systems incrementally, and deal with the undecidability of temporal properties on infinite transition systems. It is also the underlying framework for combining techniques from abstract interpretation and model checking.

- We developed new techniques for reachability analysis. Using these techniques, a set of temporal properties which we named reachability properties, can be computed faster.

1.2 Outline of the Dissertation

In Chapter 2, we give an overview of the related work. Further discussion about the related work is presented in following chapters after presenting our contributions. Chapter 3 discusses the mathematical models we use to represent reactive systems and their properties. In this chapter we also present an event-action language for specifying reactive systems. In Chapter 4, we first discuss the general principles used in model checking. Then, we describe our Presburger arithmetic model checker which enables us to analyze systems with unbounded integer variables. We demonstrate its effectiveness on some example systems. At the end of this chapter we discuss the limitations of our Presburger model checker: (1) Its inefficiency in encoding unordered variable types, and (2) the undecidability of model checking for unbounded integer domains. Following two chapters present our efforts in addressing these limitations. Chapter 5 presents a composite model checker that uses BDD and Presburger formula encodings together. In this approach we use type-specific encodings to improve the efficiency of model checking computations. Chapter 6 presents a general framework for conservatively approximating the undecidable fixpoint computations in infinite state model checking. In Chapter 7, we present heuristics for analyzing a set of properties called reachability properties. Effectiveness of all these contributions are demonstrated on example systems. Particularly, in Chapter 8 we present two case studies: Analysis of (1) a requirements specification for a nuclear reactor's water pressure system, and (2) a transport protocol. In Chapter 9, we summarize our results and point out future research directions.

Chapter 2

Related Work in Automated Verification and Static Analysis

In this chapter we give an overview of the related work. In the following chapters we discuss the related work in more detail after presenting our own contributions.

Temporal Logics and Model Checking This dissertation relies on temporal logics as a formalism for specifying properties of reactive systems – a concept introduced by Pnueli [Pnu77, Pnu81]. Reactive systems interact with their environment continuously. Partial correctness and termination are not suitable abstractions for characterizing properties of such systems. Properties of reactive systems involve notions of **invariance**, **eventuality**, and **fairness**. It is possible to express such properties with simple temporal operators using temporal logics [Pnu77, Pnu81].

Soon after the introduction of temporal logics it was observed that, for finite transition systems, temporal properties can be verified automatically [CE81, QS82]. Procedures which check if a transition system is a model for a temporal logic formula are called **model checkers**. There are two different approaches for model checking. One is the state exploration approach – using exhaustive state space search to verify a property [CES86]. This approach is a generalization of the state exploration techniques that has been used for protocol verification [Wes78]. The other approach is automata-based. It involves converting a temporal property to an automaton, and then comparing it with the automaton that represents the system [VW86].

Model checkers based on state space exploration had a breakthrough with the introduction of symbolic search techniques [BCM+90, CBM89, CMB90, McM93]. Symbolic representations significantly increased the sizes of the transition systems that could be analyzed by model checking [BCM+90]. The main idea is to represent sets of states and transition relations symbolically (implicitly), rather than using an explicit representation that uses a fixed amount of storage per state or transition. Boolean functions are the first (and the most common) symbolic representation used for this purpose. Symbolic state exploration techniques which use this encoding rely heavily on efficient data structures for manipulating Boolean functions. **Binary decision diagrams** (BDDs) [Bry86] were shown to be suitable for this task [BCM+90, McM93]. BDDs efficiently support all the functionality required for model checking [Bry86].

In this dissertation we present a new symbolic representation for unbounded integer domains. We also address the issue of using more than one symbolic representation in symbolic state exploration.

Abstract interpretation Our work was also influenced by well known techniques from abstract interpretation [CC76, CC77]. Specifically, we use an approximation technique called **widening** first developed for that domain. Most properties of programs can be formulated as least fixpoints. If the state space of the program is infinite, these fixpoints may not be computable. Abstract interpretation provides a way of approximating these fixpoints by computing their upper bounds in finite time.

The basic idea in abstract interpretation is to evaluate a given program on abstract domains. Each program point is associated with an abstract context that maps each program variable to a set of abstract values. An abstract evaluation is computed starting with null abstract contexts, and then iteratively generating an increasing chain of contexts reflecting the behavior of the program, until a fixpoint is reached. Since abstract domains are not restricted to be finite, it is possible to get an infinite increasing chain without reaching a fixpoint. To avoid this, a widening operator is used, which computes a majorizing sequence to the chain of abstract contexts. This sequence is guaranteed to converge in finite time. The computed abstract contexts have the following property: In every actual execution of the program, the set of values that a variable can realize at a certain program point is a subset of the set of values found by applying the concretization function to the abstract context at that program point.

In our work, we used the widening technique extensively. Since basic temporal operators correspond to least fixpoints, we were able to use the widening technique to compute upper bounds for them.

Symbolic Execution. Another approach for automatically verifying assertions about a program is symbolic execution [Dil90, HK76]. In this approach, specifications about a program's behavior are inserted into the code as correctness assertions, and then a symbolic execution tree is generated to verify them. In practice, this method may end up generating an infinite number of nodes, and thus never terminate. This limitation can be overcome by having the user specify assertions about a process's behavior which can be verified locally. Then the local proofs can be checked for cooperation [Dil90]. Although this method has the benefit of incrementally proving correctness (as opposed to generating all possible interleavings), it relies on users to come up with the correct assertions.

Our method has some similarities with the symbolic execution technique. One difference in our approach is the use of automated approximation techniques instead of user specified inductive assertions. Another difference is that we are using a temporal logic to specify the properties of programs, which enables us to check a richer set of properties.

Presburger Arithmetic Verification Using Presburger arithmetic in verifying programs is not a new idea. In [Coo71], Cooper developed a technique which converts queries about a program's properties to validity checks in Presburger arithmetic. The decision procedure used by Cooper was computationally very expensive which made the validity checks intractable. Also proving correctness as a single Presburger decision problem is not a method that can scale very well due to the complexity of general Presburger solvers. We have found it more beneficial to use model checking as our primary technology, and use a Presburger solver for some subservient set-theoretic computations.

Abstraction in Model Checking In [CGL92], Clarke *et al.* present a conservative abstraction method for model checking infinite-state programs. The main idea is to produce a finite model of the program using a suitable abstraction technique (e.g., congruence modulo an integer, single bit abstraction, symbolic abstraction), and then to check the property of interest on the abstraction. The abstractions are conservative for universal temporal properties (temporal properties which state conditions for all execution paths). When the properties are restricted to this subset, if the abstracted transition system satisfies a property, then the original transition system also satisfies the property. It is also shown that, if the equivalence relations induced by the abstraction function are congruences with respect to operations used in the program, then the method is exact for all temporal properties [CGL92].

Abstraction techniques require the user to select the appropriate abstractions [CGL92, DF95, WVF95]. For complex systems finding a good abstraction may be quite difficult. On the other hand, a good abstraction can simplify the model checking computations significantly. In fact, in most practical applications of model checking some type of formal or informal abstraction has been used. We see abstraction techniques as being orthogonal to our approach. There may be cases where abstractions may simplify the structure of the state space without achieving a finite abstraction. In these cases our model checker can be used on the infinite abstract models.

Model Checking Hybrid Systems Our encoding of program states and our application of the widening technique is similar to those used in verification of hybrid systems [ACHH93, ACH$^+$95, AHH96, HH95, HRP94]. A hybrid system is a discrete program which interacts with an analog environment. It is modeled as a finite automaton that has continuously evolving variables according to some dynamical laws. If activities, invariants and transition relations of an hybrid system can be defined by linear expressions, then it is called a linear hybrid system. The properties of these systems can be specified as temporal logic formulas, and for linear hybrid systems, these properties can be analyzed using symbolic model checking [ACHH93, ACH$^+$95, AHH96]. The symbolic model checker presented in [ACH$^+$95] encodes the states of an hybrid system by a set of linear equations. Temporal properties are evaluated using fixpoint computations on this representation. Reachability properties of linear hybrid systems can be computed approximately using abstract interpretation [AHH96, HH95, HRP94]. Reachability properties are stated as least fixpoint computations and upper bounds to these fixpoints are computed using the widening technique.

The fundamental difference between our work and the work on model checking hybrid systems is that we encode sets of integers as opposed to the real numbers used in hybrid systems. Hence, we use Presburger formulas as our symbolic representation. Also, we adopt the widening technique to recursive fixpoint computations using our conservative approximation framework. This enables us to handle all temporal properties, i.e., we do not restrict our approximations to reachability properties.

Chapter 3

Modeling Reactive Systems: Transition Systems and Temporal Logics

In this chapter we first present an event-action language for specifying reactive systems. The semantics of this language is described by transition systems. We discuss temporal logics as a formalism for specifying properties of reactive systems. We also discuss the fixpoint semantics of temporal operators.

3.1 Specifying Reactive Systems: An Event-Action Language

We use a simple **event-action language** to specify reactive systems [Sha93]. A reactive system $P = (V, I, E)$ is represented by (1) a finite **set of variables** V; (2) an **initial condition** I, which specifies the starting states of the program; and (3) a finite **set of events** E, where each event is considered atomic. A state of the system is determined by the values of its variables, i.e., each state corresponds to a unique valuation of variables. Each event defines a transformation on the variables of the system. Given a state, its next states are defined by the event set.

```
PROGRAM   Bakery Algorithm
VARIABLES
    $a, b$ : positive integer
    $pc_1 : \{T_1, W_1, C_1\}$     $pc_2 : \{T_2, W_2, C_2\}$
INITIAL CONDITION
    $a = b = 0 \ \wedge \ pc_1 = T_1 \ \wedge \ pc_2 = T_2$
EVENTS
    $e_{T_1}$   :   $pc_1 = T_1 \ \wedge \ pc_1' = W_1 \ \wedge \ a' = b + 1$
    $e_{W_1}$   :   $pc_1 = W_1 \ \wedge \ pc_1' = C_1 \ \wedge \ (a < b \ \vee \ b = 0)$
    $e_{C_1}$   :   $pc_1 = C_1 \ \wedge \ pc_1' = T_1 \ \wedge \ a' = 0$
    $e_{T_2}$   :   $pc_2 = T_2 \ \wedge \ pc_2' = W_2 \ \wedge \ b' = a + 1$
    $e_{W_2}$   :   $pc_2 = W_2 \ \wedge \ pc_2' = C_2 \ \wedge \ (b < a \ \vee \ a = 0)$
    $e_{C_2}$   :   $pc_2 = C_2 \ \wedge \ pc_2' = T_2 \ \wedge \ b' = 0$
```

Figure 3.1: The Bakery Algorithm

3.1.1 The Bakery Algorithm

For example, Figure 3.1 shows the specification of the **bakery algorithm** [And91] for two processes in our event-action language. This algorithm ensures mutual-exclusion between two processes in accessing a critical section.

Note that in our event-action language, the primed identifiers correspond to the next state values, and the unprimed identifiers correspond to the current state values of the variables. Hence, if a variable v is used in an event, then the symbol v' denotes the new value of v after the execution of the event. If v' is not mentioned in an event, then we assume that value of variable v is not altered by that event.

In the bakery algorithm, when a process wants to enter the critical section, it first gets a ticket, which will be higher than those of all other processes currently in the critical section or waiting for entry. In the above system, variables a and b hold the ticket values for processes 1 and 2, respectively; a process gets its ticket by simply adding one to the highest outstanding ticket number (events e_{T_1} and e_{T_2}). Note that variables a and b can increase without bound, i.e., this is not a finite-state program. The bakery algorithm is known to preserve both **mutual-exclusion** (two processes never access the critical section at the same time) and **starvation-freedom** (if a process wants to enter the critical section it eventually gets in). In the following sections we show how these properties can be formalized using temporal logics.

3.2 A Mathematical Model for Reactive Systems: Transition Systems

Transition systems are simple, general mathematical models for computer systems. A **transition system** $T = (S, I, R)$ consists of a **set of states** S, a set of **initial states** $I \subseteq S$, and a **transition relation** $R \subseteq S \times S$. Contrary to most work in model checking, we allow S to be infinite, i.e., we are able to model systems with unbounded variable domains. Also, we do not restrict R to be total, i.e., we do not assume that each state has at least one successor.

3.2.1 Event-Action Language Semantics

Given a system $P = (V, I, E)$ in our event-action language, we define its semantics as a transition system $T = (S, I, R)$, where S is obtained by taking Cartesian

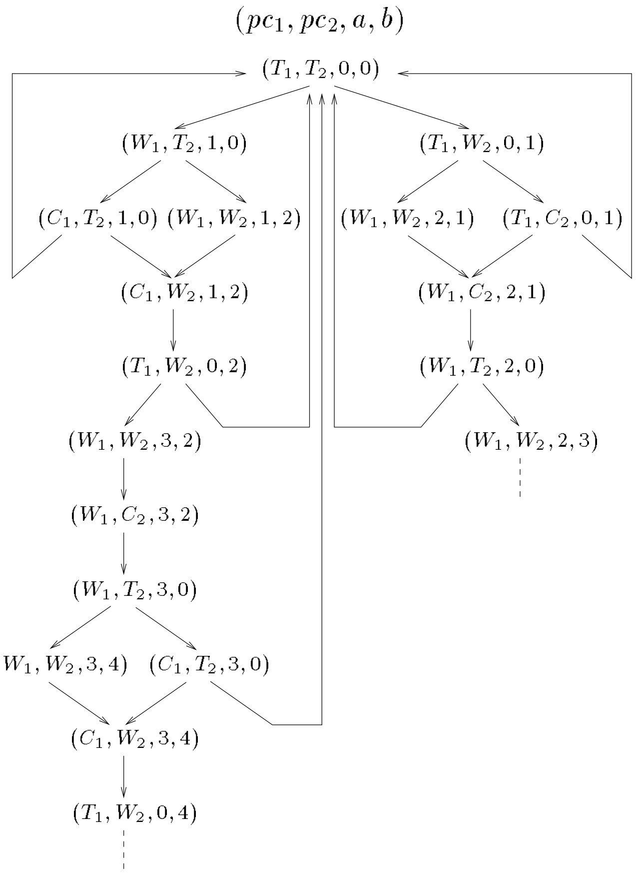


Figure 3.2: Part of the Infinite Transition System of the Bakery Algorithm

product of domains of all program variables, i.e., given a program with n variables $V = \{v_1, v_2, \ldots, v_n\}$, each state $s \in S$ corresponds to a valuation of all the variables of the program

$$s \;\equiv\; \bigwedge_{i=1}^{n} v_i = x_i$$

where $x_i \in \mathbf{domain}(v_i)$. The set of initial states are the states which satisfy the initial condition.

For example, for the bakery algorithm given in Figure 3.1, a possible state of the system is:

$$s \;\equiv\; pc_1 = T_1 \;\wedge\; pc_2 = C_2 \;\wedge\; a = 0 \;\wedge\; b = 1$$

This is a state where process one is thinking and process two is in the critical section. For the bakery algorithm, the set of initial states has only one state in it, which is:

$$s \;\equiv\; pc_1 = T_1 \;\wedge\; pc_2 = T_2 \;\wedge\; a = 0 \;\wedge\; b = 0$$

Figure 3.2 shows a part of the infinite transition system for the bakery algorithm.

In our event action language, every event $e \in E$ defines a binary relation on the program's states, $R_e \subseteq S \times S$, such that when $(s, s') \in R_e$, s and s' denote program's states before and after the execution of event e, respectively. We define the domain and the range of an event e as:

$$\begin{aligned} \textbf{domain}(e) &= \{\, s \mid \exists (s, s') \,.\, (s, s') \in R_e \,\} \\ \textbf{range}(e) &= \{\, s \mid \exists (s', s) \,.\, (s', s) \in R_e \,\} \end{aligned}$$

The global transition relation is $R \equiv \bigcup_{e \in E} R_e$. Note that this is an interleaving model, where each transition represents execution of a single event, i.e., only one event can occur at a time.

3.2.2 Paths in Transition Systems

Temporal properties of a transition system are specified on its paths. A **path** $(s_0, s_1, s_2, \ldots)$ in a transition system $T = (S, I, R)$, is a finite or infinite sequence of states, such that for each successive pair of states $(s_i, s_{i+1}) \in R$. We call s' a **successor** of state s if $(s, s') \in R$. Similarly, s' is called a **predecessor** of state s if $(s', s) \in R$.

A **maximal path** is one which is either infinite, or ends with a state that has no successors. An **execution path** is a path starting from an initial state, i.e., $(s_0, s_1, s_2, \ldots)$ is an execution path if $s_0 \in I$. The **length** of a path is the number of states in the path.

3.3 Specifying Properties of Reactive Systems: Temporal Logics

Temporal logics are based on modal logics [Eme90, Pnu77, Pnu81]. Modal logics were originally proposed to study situations where truth of a statement depends on its mode. For example, an assertion p may be false **now**, but given a mode **future**, **future** p could be true, i.e., p may become true in the future. Pnueli recognized that such logics were suitable formalisms for specifying properties of reactive systems [Pnu77, Pnu81]. This view has been confirmed by their extensive use in the last two decades.

Temporal logics are composed of a nontemporal part for specifying basic properties of states, plus a set of temporal operators for specifying temporal properties. Nontemporal properties can be checked on each individual state by observing the values of the variables in that state, whereas verifying temporal properties involves investigating paths of the system.

First, we discuss nontemporal portion of temporal logics, i.e., specification of basic properties of states. Given a transition system $T = (S, I, R)$, we define BP to be the set of **basic state properties** of the system. We assume that we have a procedure $L : S \times BP \rightarrow \{\mathbf{true}, \mathbf{false}\}$ such that given a property $p \in BP$ and a state $s \in S$, procedure L can decide if $s \models p$, i.e., $s \models p$ if and only if $L(s, p) = \mathbf{true}$.

In propositional temporal logics BP consists of propositional logic formulas, whereas in a first order temporal logic it includes first order logic formulas. In this chapter, we do not define the set BP. We just state that there exists a decision procedure L which can determine the truth value of the formulas in BP. The definition of the formulas in BP are given in the following chapters after the presentation of symbolic representations.

3.3.1 Temporal Logic CTL

Various temporal logics have been defined in the last two decades [Eme90]. In this dissertation we use the **computation tree logic** (CTL) [CES86] which is one of the most popular temporal logics. We first present four temporal operators G, F, U, X with the following intuitive meanings:

G p : Henceforth p (or Globally p), i.e., assertion p holds in every state.

F p : Eventually p (or Future p), i.e., assertion p will hold in a future state.

p U q : p Until q, i.e., assertion q will hold in a future state and assertion p will hold until q holds.

X p : Next p, i.e., assertion p will hold in the next state.

CTL temporal operators are generated from these basic temporal operators by adding one of the path quantifiers A or E as a prefix. Path quantifier A refers to all paths, whereas E refers to some paths.

The following grammar defines the syntax of CTL, where terminals $\neg$, $\wedge$, and $\vee$ are logical connectives; terminals EX, AX, EU and AU are temporal operators; terminal bp denotes a basic property, (i.e., bp $\in BP$); and nonterminal TF denotes a CTL formula:

$$\begin{aligned} \text{TF} \ ::= \ & \text{bp} \mid \text{TF} \ \wedge \ \text{TF} \mid \text{TF} \ \vee \ \text{TF} \mid \neg \ \text{TF} \mid \text{EX TF} \mid \text{AX TF} \mid \\ & \text{TF EU TF} \mid \text{TF AU TF} \end{aligned}$$

We have the following equivalences for CTL operators,

$$\begin{aligned} \text{EF}\ p &\equiv \textbf{true}\ \text{EU}\ p & \text{EG}\ p &\equiv \neg(\textbf{true}\ \text{AU}\ \neg p) \\ \text{AF}\ p &\equiv \textbf{true}\ \text{AU}\ p & \text{AG}\ p &\equiv \neg(\textbf{true}\ \text{EU}\ \neg p) \\ \text{AX}\ p &\equiv \neg\text{EX}\neg p \end{aligned}$$

Based on these equivalences, the set $\{\neg, \vee, \text{EX}, \text{EU}, \text{AU}\} \cup BP$ forms a basis for CTL, i.e., we can express any CTL property using only symbols from this set.

In Table 3.1 we define the semantics of CTL. Based on the rules given in Table 3.1, we can decide if a state in a transition system satisfies a CTL formula. Semantics of the temporal operators EF, AF, EG and AG follow from the equivalences given above and the rules given in Table 3.1.

We also define the truth value of CTL formulas for transition systems:

> Given a transition system $T = (S, I, R)$, and a temporal formula p,
> $T \models p$ if for all states $s \in I$, $s \models p$.

Hence, the transition system satisfies the temporal formula if all its initial states satisfy the formula.

Now, we can define the truth value of temporal formulas for systems specified in our event action language. Given a reactive system $P = (V, I, E)$, and its corresponding transition system $T = (S, I, R)$, and a temporal formula p, $P \models p$ if $T \models p$.

In Figure 3.3, we show the truth values of temporal properties EFp, AFp, EGp and AGp for a particular state on four simple transition systems. The

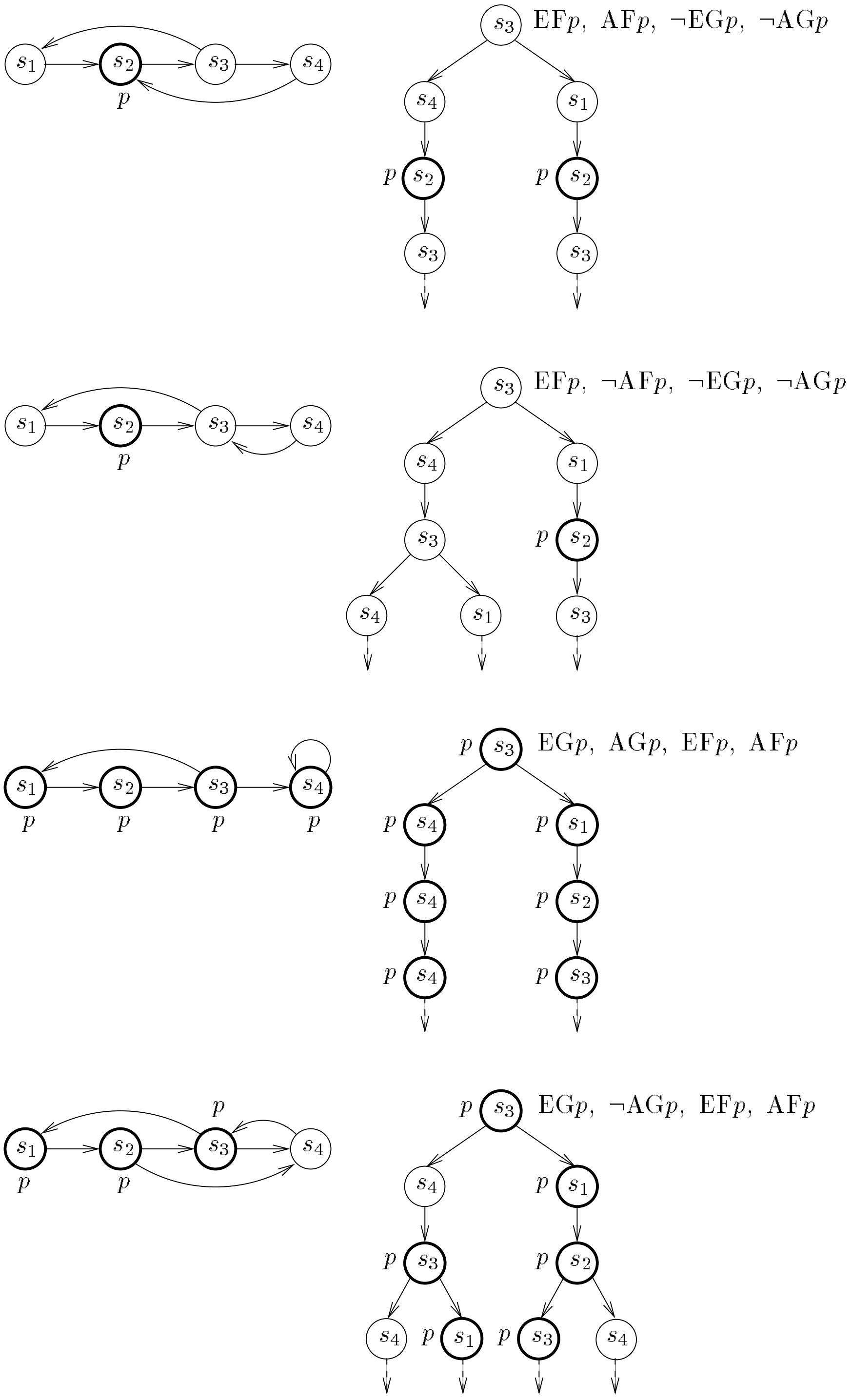


Figure 3.3: CTL Operators EF, AF, EG and AG

$s \models p$	iff	$L(s,p) = \mathbf{true}$ where $p \in BP$.
$s \models \neg p$	iff	not $s \models p$.
$s \models p \wedge q$	iff	$s \models p$ and $s \models q$.
$s \models p \vee q$	iff	$s \models p$ or $s \models q$.
$s_0 \models$ EX p	iff	for some maximal path $(s_0, s_1, s_2, \ldots)$, with length ≥ 2, $s_1 \models p$.
$s_0 \models$ AX p	iff	for all maximal paths $(s_0, s_1, s_2, \ldots)$, with length ≥ 2, $s_1 \models p$.
$s_0 \models p$ EU q	iff	for some maximal path $(s_0, s_1, s_2, \ldots)$, there exists an i, $s_i \models q$, and for all $j < i$, $s_j \models p$.
$s_0 \models p$ AU q	iff	for all maximal paths $(s_0, s_1, s_2, \ldots)$, there exists an i, $s_i \models q$, and for all $j < i$, $s_j \models p$.

Table 3.1: CTL Semantics

computation trees to the right of the transition systems are generated by unwinding the transition relation. Note that all the paths originating from state s_3 are included in these computation trees. If we assume that s_3 is an initial state, then these paths correspond to execution paths. One nice property of the CTL is that for finite-state transition systems, finding the truth value of a CTL formula has linear time complexity in the size of the transition system and the formula.

Consider the two properties of the bakery algorithm discussed above. The safety property – two processes are never at the critical section at the same time – can be expressed in CTL as follows:

$$\text{(B1)} \quad \mathrm{AG}(\neg(pc_1 = C_1 \ \wedge \ pc_2 = C_2))$$

Now recall the progress property: If a process starts waiting for entry to the critical section, it eventually gets in. For the first process, this can be expressed as:

$$\text{(B2)} \quad \mathrm{AG}(pc_1 = W_1 \rightarrow \mathrm{AF}(pc_1 = C_1))$$

3.3.2 Fixpoint Characterization of Temporal Formulas

All temporal properties in CTL can be expressed using only the logical connectives $\neg$, $\vee$, the existential next state operator EX, and a least fixpoint operator. The logic which includes these operators is called the **μ-calculus** [Koz83], and it subsumes most temporal logics, including CTL and more powerful logics such as CTL* [Dam94, Eme90]. The power of the μ-calculus comes from its fixpoint operators.

Since μ-calculus formulas are constructed from simple operators, they look quite cumbersome. However, they are closely related to the procedures we use to compute the truth sets of temporal properties. Looking at a μ-calculus formula one can see how our model checker would compute it. Hence, μ-calculus formulas are very useful in explaining the automated verification procedures presented in the next chapter.

The following grammar defines the syntax of the μ-calculus, where terminals $\neg$, $\wedge$, and $\vee$ are logical connectives; terminal EX is the existential next state temporal operator; terminal AX is the universal next state temporal operator; terminal μ is the least fixpoint operator; terminal ν is the greatest fixpoint operator; terminal x is a propositional variable; terminal bp denotes a basic property, (i.e., bp $\in BP$); and nonterminal MF denotes a μ-calculus formula:

$$\begin{aligned} \mathrm{MF} \; ::= \; & \mathrm{bp} \mid \mathrm{MF} \wedge \mathrm{MF} \mid \mathrm{MF} \vee \mathrm{MF} \mid \neg\, \mathrm{MF} \mid \mathrm{EX}\ \mathrm{MF} \mid \mathrm{AX}\ \mathrm{MF} \mid \\ & \mu x\ .\ \mathrm{MF} \mid \nu x\ .\ \mathrm{MF} \end{aligned}$$

where, in the last two production rules, MF must be syntactically monotone in the variable x, i.e., all free occurrences of x in MF must occur in the scope of an even number of negations $\neg$. The notions of scope, bound and free occurrences are the same as the first order predicate logic, where fixpoint operators μ and ν are treated as quantifiers. We have the following equivalences

$$\mathrm{AX}\ p \ \equiv\ \neg\mathrm{EX}\neg p \quad \text{and} \quad \nu x \ .\ \mathcal{F}\ x \ \equiv\ \neg\mu x \ .\ \neg\mathcal{F}\ \neg x$$

Hence, we can represent any μ-calculus formula using the basis $\{\neg, \vee, \mathrm{EX}, \mu\} \cup BP$.

To explain the interpretation of μ-calculus formulas, we need a short discussion on the lattice theory. Given a transition system $T = (S, I, R)$, the power set of S, 2^S may be viewed as a complete lattice $(2^S, S, \emptyset, \subseteq, \cup, \cap)$, with the top element S, the bottom element $\emptyset$, intersection $\cap$ as the meet operator, union $\cup$ as the join operator, and the set containment $\subseteq$ as the partial ordering.

We can interpret every temporal logic formula as a representation of its truth set. Hence, **false** corresponds to the bottom element $\emptyset$ of the lattice, and **true** corresponds to the top element S (we use temporal formulas and their truth sets interchangeably). Then next operators EX and AX correspond to functionals from 2^S to 2^S. Therefore, given a set of states denoted by a proposition p, EX p maps p to the states which can reach p after traversing a single transition, i.e.,

$$\mathrm{EX}\ p \ \equiv\ \{\ s \mid \exists s' \ .\ s' \in p \ \wedge\ (s, s') \in R\ \}.$$

We can denote this functional as λx . EX x, where x denotes a set of states (we drop λx when it is clear from the context).

Given a functional $\mathcal{F} : 2^S \rightarrow 2^S$, $\mathcal{F}\ p$ denotes the application of functional $\mathcal{F}$ to set $p \subseteq S$. Also, $\mu x \ .\ \mathcal{F}\ x$ denotes the least fixpoint of $\mathcal{F}$, i.e., the smallest x such that $x \equiv \mathcal{F}\ x$. Similarly, $\nu x \ .\ \mathcal{F}\ x$ denotes the greatest fixpoint of $\mathcal{F}$, i.e., the greatest x such that $x \equiv \mathcal{F}\ x$. We call a functional $\mathcal{F}$ **monotonic**, if $p \subseteq q$ implies $\mathcal{F}\ p \subseteq \mathcal{F}\ q$. We have the following property from the lattice theory [Tar55]:

Theorem 3.1 *Let* $\mathcal{F} : 2^S \rightarrow 2^S$ *be a monotonic functional. Then* $\mathcal{F}$ *always has a least and a greatest fixpoint, which are respectively*

$$\begin{aligned} \mu x \,.\, \mathcal{F}\ x &\equiv \bigcap \{\ x \mid \mathcal{F}\ x \subseteq x\ \} \\ \nu x \,.\, \mathcal{F}\ x &\equiv \bigcup \{\ x \mid \mathcal{F}\ x \supseteq x\ \} \end{aligned}$$

Hence, $\mu x \,.\, \mathcal{F}\ x$ is the intersection of all the sets x where $\mathcal{F}\ x \subseteq x$, and similarly, $\nu x \,.\, \mathcal{F}\ x$ is the union of all the sets x where $\mathcal{F}\ x \supseteq x$.

Note that the syntactic restriction on the μ-calculus formulas guarantees that every functional appearing in a formula is monotonic, i.e., the property given above is always applicable. The semantics of the μ-calculus follows from this property, and the semantics of the next state operators defined previously.

Below we show the μ-calculus representation of CTL properties. The logical connectives $\neg$, $\vee$, $\wedge$, and the next state operators EX and AX are included in both CTL and the μ-calculus, and hence, they do not need to be translated. The CTL temporal operators EF, AF, EG, and AG can all be expressed using the CTL temporal operators EU and AU using the equivalences given in the previous section. Hence, we only need to consider CTL formulas of the form p EU q, and p AU q for translation. Translation of these formulas to μ-calculus is given by the following property [Arn94]:

Theorem 3.2 *Truth sets of CTL formulas* p *EU* q, *and* p *AU* q *can be expressed* *as least fixpoints:*

$$\begin{aligned} p\ EU\ q &\equiv \mu x\ .\ q \ \vee\ (p\ \wedge\ EX\ x) \\ p\ AU\ q &\equiv \mu x\ .\ q \ \vee\ (p\ \wedge\ AX\ x\ \wedge\ EX\ x) \end{aligned}$$

If the underlying transition relation is total, the EX operator in the fixpoint characterization of p AU q can be dropped. However, since we allow non-total transition systems, we need the EX operator to guarantee that q is eventually satisfied.

Chapter 4

Symbolic Model Checking Using Presburger Arithmetic

In this chapter we present a symbolic model checker to analyze systems with unbounded integer domains. Our tool is based on the following concepts:

- Symbolically encoding transition relations and sets of states using affine constraints on integer variables, logical connectives and quantifiers (i.e., Presburger formulas).
- Partitioning a program's state-space via the control variables or domains of events, and using the partition classes as repositories for the model checker's symbolic computations.
- Efficiently manipulating Presburger formulas (via a fast Presburger solver called the Omega Library [KMP$^+$95, Pug92]) during fixpoint computations to derive truth sets of temporal logic formulas.

The contributions discussed in this chapter have been initially reported in [BGP96, BGP97] and [BGP98].

4.1 Model Checking

Given a reactive system $P = (V, I, E)$ with its transition system $T = (S, I, R)$ and a temporal property p, we want to verify that $P \models p$, i.e., $T \models p$. A procedure which can automatically verify this is called a **model checker**, since it is checking if the program P or the transition system T is a model for the temporal formula p.

As defined in the previous chapter, $T \models p$ if for all initial states $s \in I$, $s \models p$. In other words, using the truth set interpretation of the temporal formulas, $T \models p$ if $I \subseteq p$.

Finding a **bug** in a system is equally valuable (if not more valuable) as verifying it. Finding a bug corresponds to finding an initial state $s \in I$ which does not satisfy the temporal property, i.e., $s \in I \cap \neg p$. It is possible to generate a counter-example execution path, which demonstrates violation of property p, starting from such a state. One of the reasons for success of model checking procedures is the fact that they are able to generate counter-example behaviors [CGKM94].

Then, our goal in analyzing a system $P = (V, I, E)$ and a temporal property p is :

- Either to prove that the system P satisfies the property p by showing that $I \subseteq p$.
- Or to demonstrate a bug by finding a state $s \in I \cap \neg p$, and generating a counter-example execution path starting from s.

Assume that there exists a representation for sets of states which supports tests for equivalence, emptiness, and membership. Then, if we can represent the truth set of the temporal property p, and the set of initial states I using this representation, we can check the conditions above. If the state space is finite, explicit state enumeration would be one such representation. Note that as the state spaces of the programs grow, explicit state enumeration will become more expensive since the size of this representation is linearly related to the number of states in the set it represents. Unfortunately, state space of programs increase exponentially with the number of variables and concurrent components. This is called the **state space explosion problem**, and makes a naive implementation of the explicit state enumeration infeasible.

Another method is to use a **symbolic representation** for encoding sets of states. Symbolic representations are mathematical objects with semantics corresponding to sets of states. We can use such representations in encoding the truth sets of temporal formulas. For example Boolean logic formulas can be used as a symbolic representation for finite-state systems.

Model checking procedures compute a representation for truth sets of temporal formulas using state space exploration. Fixpoints corresponding to truth sets of temporal formulas can be computed by aggregating sets of states iteratively. Temporal properties which require more than one fixpoint computation can be computed recursively starting from the inner fixpoints and propagating the partial results to the outer fixpoints. In the following sections, we demonstrate how to implement a model checker, and discuss the functionality that a symbolic representation has to support to be included in a model checker.

4.1.1 Computing Truth Sets of Temporal Properties

Given a functional $\mathcal{F}$, $\mathcal{F}^i\ p$ is defined as:

$$\mathcal{F}^i\ p \stackrel{\text{def}}{=} \underbrace{\mathcal{F}\ (\mathcal{F} \ldots (\mathcal{F}}_{i \text{ times}}\ p)).$$

We define $\mathcal{F}^0$ as the identity relation. Then, we have the following property [Tar55]:

Theorem 4.1 *Given a monotonic functional* $\mathcal{F}\ :\ 2^S \rightarrow 2^S$*, for all* n*,*

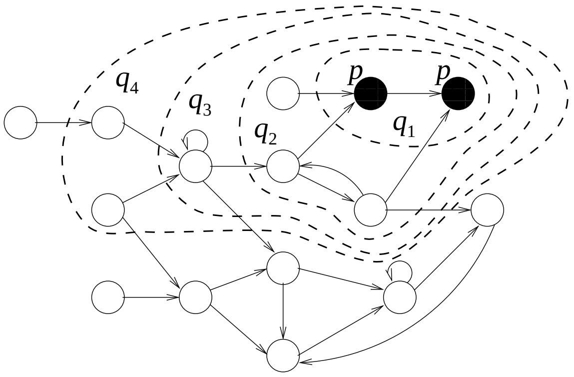
$$\begin{aligned} \mu x\ .\ \mathcal{F}\ x &\supseteq \bigcup_{i=0}^{n} \mathcal{F}^i\ \mathbf{false} \\ \nu x\ .\ \mathcal{F}\ x &\subseteq \bigcap_{i=0}^{n} \mathcal{F}^i\ \mathbf{true} \end{aligned}$$

Assume that we generate a sequence of approximations to the least fixpoint $\mu x\ .\ \mathcal{F}\ x$ of a monotonic functional $\mathcal{F}$ by generating the following sequence:

$$\mathbf{false},\ \mathcal{F}\ \mathbf{false},\ \mathcal{F}^2\ \mathbf{false},\ \ldots,\ \mathcal{F}^i\ \mathbf{false},\ \ldots$$

This sequence is monotonically increasing since **false** corresponds to the bottom element of the lattice, and the functional $\mathcal{F}$ is monotonic. If this sequence converges to a fixpoint, i.e., if we find an i where $\mathcal{F}^i\ \mathbf{false} \equiv \mathcal{F}^{i+1}\ \mathbf{false}$, then from the property above, we know that it is the least fixpoint, i.e., it is equal to $\mu x\ .\ \mathcal{F}\ x$.

Similarly, a monotonically decreasing sequence of approximations could be generated for the greatest fixpoint. In this dissertation, we only use least fixpoints. Because of the duality between the least and the greatest fixpoints, this does not restrict the set of properties we are able to analyze.



$$\mathrm{EF}\ p \equiv \underbrace{\underbrace{\underbrace{\underbrace{\underbrace{\mathbf{false}}_{q_0} \vee\ p}_{q_1} \vee\ \mathrm{EX}\ p}_{q_2} \vee\ \mathrm{EX}\,(\mathrm{EX}\ p)}_{q_3} \vee\ \mathrm{EX}\,(\mathrm{EX}\,(\mathrm{EX}\ p))}_{q_4} \vee \ldots$$

Figure 4.1: Fixpoint Computations for EF p

As an example for computing least fixpoints, consider the property EF p. A state satisfies EF p if there exists a path starting from that state which has a state that satisfies p. Based on the properties discussed in Chapter 3, we have

$$\mathrm{EF}\ p \equiv \mathbf{true}\ \mathrm{EU}\ p \equiv \mu x\ .\ p \vee (\mathbf{true} \wedge \mathrm{EX}\ x) \equiv \mu x\ .\ p \vee \mathrm{EX}\ x$$

i.e., EF p is equivalent to the least fixpoint of the functional $\mathcal{F}\ x \equiv p \vee \mathrm{EX}\ x$. We can compute the truth set of EF p by generating the following sequence:

$$\underbrace{\underbrace{\underbrace{\mathbf{false} \vee p}_{\mathcal{F}\ \mathbf{false}} \vee \mathrm{EX}\ p}_{\mathcal{F}^2\ \mathbf{false}} \vee\ \mathrm{EX}\,(\mathrm{EX}\ p)}_{\mathcal{F}^3\ \mathbf{false}} \vee \mathrm{EX}\,(\mathrm{EX}\,(\mathrm{EX}\ p)) \vee \ldots$$

as shown in Figure 4.1 on an example transition system. When this sequence converges to a fixpoint, the result will be equal to the truth set of the property EF p.

```
Procedure Evaluate($f$)
  Case
    IsEvaluated($f$)                  : Return($f$)
    $f = \neg f_1$                    : Return(Not(Evaluate($f_1$)))
    $f = f_1 \ \wedge \ f_2$          : Return(And(Evaluate($f_1$),Evaluate($f_2$)))
    $f = f_1 \ \vee \ f_2$            : Return(Or(Evaluate($f_1$),Evaluate($f_2$)))
    $f = \mathrm{EX}\ f_1$            : Return(EX(Evaluate($f_1$)))
    $f = \mu x\ .\ \mathcal{F}\ x$    : $q_0 \equiv \mathbf{false}$     $q_{i+1} \equiv$ Evaluate($\mathcal{F}\ q_i$)
                                        Return($q_n$) when $q_n \equiv q_{n+1}$
```

Figure 4.2: A Model Checking Algorithm for μ-calculus

Based on these properties we can develop a μ-calculus model checker as shown in Figure 4.2. The procedure `Evaluate`(f) computes the truth set of μ-calculus formula f. Using the equivalences AX $p \equiv \neg\mathrm{EX}\neg p$, and $\nu x\ .\ \mathcal{F}\ x \equiv \neg\mu x\ .\ \neg\mathcal{F}\ \neg x$, this procedure can compute truth set of any μ-calculus formula.

The function `IsEvaluated`(f) returns **true** if the truth set of f is already computed. Note that for a basic property $f \in BP$, `IsEvaluated`(f) always returns **true**, i.e., we assume that the truth set of basic properties are represented in the form we want before we call the procedure `Evaluate`.

As discussed in the previous section, after the truth set of f is computed, we can check if $I \subseteq f$. If this condition holds the model checker reports that the property is proved. If $I \nsubseteq f$, we compute the truth set of $\neg f$ and look for a state $s \in I \cap \neg f$, and generate a counter example. We do not discuss counter-example generation here, it is reported in [CGKM94].

4.1.2 Symbolic Representations

Assume that `Symbolic` is the data type used for encoding sets of states, and `Formula` is the data type used for representing μ-calculus formulas. Then, the signature of the procedure `Evaluate`, shown in Figure 4.2, is

`Symbolic Evaluate(Formula)`

The procedure `Evaluate` computes a representation of type `Symbolic` for the truth set of its argument formula. Note that, in Figure 4.2, the operators $\neg$, $\wedge$, $\vee$, and EX are implemented by procedures `Not`, `And`, `Or`, and `EX`. We specify these procedures below:

`Symbolic Not(Symbolic)` : Given an argument that represents a set $p \subseteq S$, it returns a representation for $S - p$.

`Symbolic And(Symbolic,Symbolic)` : Given two arguments representing two sets $p, q \subseteq S$, it returns a representation for $p \cap q$.

`Symbolic Or(Symbolic,Symbolic)` : Given two arguments representing two sets $p, q \subseteq S$, it returns a representation for $p \cup q$.

`Symbolic EX(Symbolic)` : Given an argument that represents a set $p \subseteq S$, it returns a representation for the set $\{s \mid \exists s' \,.\, s' \in p \ \wedge \ (s, s') \in R\}$.

`Boolean IsSatisfiable(Symbolic)` : Given an argument representing a set $p \subseteq S$, returns **true** if $p \equiv \emptyset$, returns **false** otherwise.

Combined with the other procedures, the last procedure is used to test for subsumption and equivalence. Note that we need this functionality to check for convergence in the fixpoint computations.

The computation of the procedure `EX` involves computing a relational image. Given a set $p \subseteq S$ and a relation $X \subseteq S \times S$ we use $X\ p$ to denote relational image of p under X, i.e., $X\ p$ is defined as restricting the domain of X to set p, and returning the range of the result. Note that we can think of relation X as a functional $X\ :\ 2^S \rightarrow 2^S$. Then, $X\ p$ denotes the application of the functional X to set p. Let R^{-1} denote the inverse of the transition relation R. Then $\mathrm{EX}\ p \equiv R^{-1}\ p$, i.e., functional EX corresponds to the inverse of the transition relation R. Hence, we can compute the procedure `EX` using a relational image computation. Most model checkers represent transition relation R in a partitioned form to make the relational image computation more efficient. We can take advantage of our disjunctive event decomposition $R \equiv \bigcup_{e \in E} R_e$ for this purpose. We discuss this in more detail in the following sections.

Using the procedures described above, given a temporal formula, the procedure `Evaluate` computes a symbolic representation for its truth set.

Any representation which can encode the set of initial states I, and the set of basic properties BP, and supports the above functionality could be used as a symbolic representation in a symbolic model checker. We call a representation L an **adequate language** [KMM$^+$97] if

- Initial states I, the transition relation R, and the set of basic properties BP can be expressed in L.
- L is effectively closed under the Boolean connectives negation, conjunction and disjunction.
- Satisfiability is decidable for assertions over L, and we have an algorithm to carry out the procedure.

- There is also an algorithm to compute binary relational images over L. That is, given X of type $L \rightarrow L$, and $p \in L$, we can compute $X\ p$, which is defined by restricting the domain of X to set p, and returning the range of the result.

For example, for finite-state systems, Boolean logic could be used as a symbolic representation. We can implement procedures for negation, conjunction and disjunction of Boolean logic formulas. We can also implement a procedure for checking satisfiability. If we can represent the transition relation R as a Boolean logic formula, then relational image computation $R^{-1}\ p$ can be computed by conjuncting the formula representing R and formula representing p, and eliminating the variables in the domain of R^{-1} using existential quantifier elimination.

Ordered binary decision diagrams (BDDs) are an efficient data structure for representing Boolean logic formulas, and they support all the functionality we described above [Bry86]. They have been successfully used for symbolic model checking [BCM+90, McM93]. However, they can not encode infinite variable domains. In the following sections, we present a symbolic representation that can be used for model checking systems with infinite integer domains.

4.2 Presburger Arithmetic Model Checking

We define **Presburger formulas** as the formulas generated by the following grammar:

$$\begin{aligned} \mathrm{AF} &::= \mathrm{AT} \leq \mathrm{AT} \mid \mathrm{AF} \ \wedge \ \mathrm{AF} \mid \neg \mathrm{AF} \mid \exists x \,.\, \mathrm{AF} \\ \mathrm{AT} &::= \mathrm{AT} + \mathrm{AT} \mid x \mid c \end{aligned}$$

where the terminal c represents an integer constant, terminal x represents an integer variable, terminals $\neg$, $\wedge$, and $\vee$ are logical connectives, terminal $\exists$ represents existential quantification, nonterminal AF represents a Presburger formula, and nonterminal AT represents an Presburger term. Using this base language, we can easily represent formulas including $<$, $=$, $\vee$, $\forall$, as well as multiplication by a constant. The set of closed formulas defined by the above grammar forms the theory of integers with addition, called **Presburger arithmetic**. An important property of Presburger arithmetic is that validity is decidable.

In general, the worst-case time bound for determining validity in Presburger arithmetic is prohibitive, with a deterministic upper bound of $2^{2^{2^{pn}}}$ (for some constant $p > 1$) [Opp78], and a nondeterministic lower bound of $2^{2^{cn}}$ (for some constant $c > 0$) [FR74], where n denotes the length of the formula. Yet we have found that the Omega Library [KMP$^+$95, Pug92] is quite efficient at solving the problems that arise in our analysis, which typically possess a small number of constraints, and do not contain multiple levels of alternating quantifiers. The Omega Library uses extensions of Fourier variable elimination to solve integer programming problems, along with a set of transformation functions and heuristics to help convert real-valued approximations into discrete-valued solutions.

Presburger formulas – and their corresponding set-theoretic interpretations – give us a convenient way to symbolically encode sets of program states. Moreover, we also use this encoding to represent the program's underlying transition relation, i.e., we represent each event with a Presburger formula. We also assume that the set of basic properties BP and the initial condition I of our input programs are expressible in Presburger arithmetic. Hence, the set of Presburger formulas forms an adequate language, since all the functionality we described in

```
PROGRAM  Unbounded Buffer
VARIABLES
  $p, c, i_1, i_2$: positive integer
  $pc : \{Idle, Send\}$
INITIAL CONDITION
  $p = c = i_1 = i_2 = 0$
EVENTS
  $e_{IS}$  :  $pc = Idle \;\wedge\; pc' = Send$
  $e_S$     :  $pc = Send \;\wedge\; p' = p + 1 \;\wedge\; (i_1' = i_1 + 1 \;\vee\; i_2' = i_2 + 1)$
  $e_{SI}$  :  $pc = Send \;\wedge\; pc' = Idle$
  $e_R$     :  $c' = c + 1 \;\wedge\; ((i_1 > 0 \;\wedge\; i_1' = i_1 - 1) \;\vee\; (i_2 > 0 \;\wedge\; i_2' = i_2 - 1))$
```

Figure 4.3: Indexing for Two Unbounded Buffers

the previous section is implementable for the Presburger formulas. Note that since Presburger formulas can encode unbounded integer variable domains, we can encode infinite transition systems using this representation.

4.2.1 An Unbounded Buffer

We explain our Presburger formula representation below, using a simple example. Consider the concurrent program (UB) shown in Figure 4.3, which handles the counters for two unbounded buffers. Note that there is a single control variable pc which is used to attenuate the producer – and it can either be idle (i.e., $pc = Idle$) or ready-to-send (i.e., $pc = Send$). Alternatively, the consumer has only one event (e_R), which can execute whenever outstanding data is sitting in either of the buffers.

Four data variables are used to keep track of the the data items, from the perspective of the producer, the consumer and the two buffers. The total number of items produced (over the program's lifetime) is kept in the integer p; similarly, the number of items consumed is counted in variable c. The other two variables – i_1 and i_2 – keep track of the outstanding items in the buffers. Note that all the data variables can increase without bound, i.e., this program is not a finite-state program.

Three properties of interest are as follows:

- (UB1) $\mathrm{AG}(p = c + i_1 + i_2)$: The total number of items produced is equal to the number of items consumed, plus the number of items sitting in the two buffers.
- (UB2) $\mathrm{AG}((pc = \mathit{Idle} \ \wedge \ i_1 + i_2 = n) \rightarrow \mathrm{AX}(i_1 + i_2 \leq n))$: If the sender is idle, the number of items in the buffers monotonically decreases.
- (UB3) $\mathrm{AG}(p \geq c)$: The number of items consumed never exceeds the number produced.

Note that the variable n in (UB2) is not a program variable; rather, it is an open integer variable. This points out one of the strengths of our Presburger representation – since n can be treated as a symbolic constant within a temporal expression. The interpretation is that n is bound by a universal quantifier (and our Presburger solver treats it as such).

Given scale of the program, one need not possess extraordinary skills to understand that the properties are indeed true. Yet, while we may know that this program is correct, its infinite state space will overcome most contemporary automated analysis engines. Hence, it is easy to see how a larger, more complex program would, in fact, be inaccessible to a both a programmer's (informal) "what-if" checks, and to an automated finite-state analyzer.

4.2.2 Presburger Arithmetic Representations

Presburger formulas give us a convenient, compact way to symbolically encode large sets of program states and transitions. For our purposes, the benefit of the Presburger encoding is often realized via the arithmetic inequality operators – which we use to implicitly describe large, nontrivial portions of a program's state space. For example, consider the following formula:

$$p - c \leq i_1 + i_2 \leq p \quad \wedge \quad p, c, i_1, i_2 \geq 0$$

In geometric terms, the constraints represent all points in an unbounded, 4-dimensional polytope. Yet in terms of our example, the shape corresponds to all of the program's reachable states (minus the program-counter).

Similar gains are realized for transitions. The events of our event-action language programs are described in terms of Presburger formulas. (This prevents us from using multiplication in a single event, and ensures that single-step image computations are decidable.) Hence, for a given event e, the transition relation of event e, R_e, is representable as a Presburger formula. So using $|E|$ Presburger formulas, one for each event, we can symbolically encode the transition relation R as

$$R \;\equiv\; \bigvee_{e \in E} R_e.$$

DNF Representation We use the Omega Library for symbolic manipulation of Presburger formulas [KMP$^+$95, Won95]. We store the Presburger formulas in the disjunctive normal form (DNF) representation provided by the Omega Library. In the DNF representation, a Presburger formula is represented as the union of a finite list of clauses. Each clause consists of a finite number of conjuncts. Each conjunct represents either an affine constraint or a stride constraint. An **affine constraint** is a linear equality or inequality constraint of the form

$$\sum_{1\leq i\leq n} a_i v_i + a_0 = 0 \quad \text{or} \quad \sum_{1\leq i\leq n} a_i v_i + a_0 \geq 0$$

respectively, where v_i's denote integer variables, and a_i's denote integer constants. A **stride constraint** is a constraint of the form

$$c \mid \sum_{1\leq i\leq n} a_i v_i + a_0$$

which is interpreted as "$\sum_{1\leq i\leq n} a_i v_i + a_0$ is divisible by c", where c is an integer constant. A stride constraint can be represented as an equality constraint using existential quantification as follows:

$$c \mid \sum_{1\leq \imath\leq n} a_i v_i + a_0 \;\equiv\; \exists y \;.\; \sum_{1\leq i\leq n} a_i v_i + a_0 = cy$$

In the Omega Library stride constraints are represented using existential quantification.

Any Presburger formula can be converted to DNF. In our symbolic model checking computations we use the DNF representation, i.e., given a set p we represent it as:

$$p \;\equiv\; \bigvee_i p_i, \qquad p_i \;\equiv\; \bigwedge_j c_{ij}$$

where each c_{ij} is a single affine constraint or a single stride constraint over the variables of the program. Transition relations of the events are represented similarly using constraints over primed and unprimed variables.

4.2.3 Partitioning Techniques

Computing fixpoints on partitioned transition systems is a common technique in symbolic model checking [BCL$^+$94]. It is analogous to the partitioning of the transition systems of the programs for fixpoint computations in data flow analysis. The idea is to decompose the state space and the transition relation of a program instead of computing fixpoints on one monolithic transition relation.

There are three main reasons for using partitioning techniques in our symbolic model checker: 1) To reduce the sizes of the Presburger formulas generated during the fixpoint computations, and to improve the efficiency of their manipulation. 2) To eliminate selected variables by partitioning the state space based on their valuations. 3) To improve the accuracy of the conservative approximation techniques.

The fundamental challenge in the Presburger formula encoding is to keep the sizes of the formulas small during the fixpoint computations. Since each image computation in a fixpoint iteration is essentially a pre- or post-condition computation, the number of disjuncts in the DNF representations may increase very quickly in the presence of if-then-else branches and loops.

Given a Presburger formula in the DNF form, our minimization procedures try to merge the disjuncts to reduce the size of the Presburger representation. Assume that we represent each fixpoint iterate using one Presburger formula. After a couple of fixpoint iterations such a representation would generate a Presburger formula with a large number of disjuncts. Trying to merge each pair of disjuncts one by one would be computationally very expensive. Instead, we use partitioning heuristics. We partition the state space of the program so that the states with similar properties are placed in the same partition class. Then, we represent each fixpoint iterate using one Presburger formula per partition class. This reduces the complexity of the minimization procedures by segregating the disjuncts which are unlikely to be merged. It also enables us to compute the relational images incrementally, using one event and one partition class at a time.

Another reason for partitioning the state space is to eliminate variables. Manipulation of Presburger formulas are computationally expensive. Hence, we would like to minimize the number of integer variables in our Presburger formula representations. We can eliminate variables with small domains (such as program counters and Boolean variables) by partitioning the state space based on their valuations. This saves us from mapping such variables to integer domains and producing constraints about them during the fixpoint computations.

We also use approximation techniques to limit the growth of fixpoint iterates. Since the temporal properties we analyze are undecidable, the fixpoint iterations are not guaranteed the converge. Using conservative approximation techniques, we increase the convergence rate of the fixpoint computations. Partitioning heuristics can increase the precision of these approximation techniques. For example one approximation technique would be to merge two disjuncts approximately using an upper or a lower bound. Such a technique is more likely to succeed if the merged clauses contain states with similar characteristics.

Below we describe our partitioning heuristics. First we describe transition relation partitioning which enables us to compute fixpoint iterations incrementally using one event at a time. Then we present our state-space partitioning strategies which enables us to store the fixpoint iterates using one Presburger formula per partition class.

Transition Relation Partitioning

The most obvious partitioning of the transition relation, R, comes immediately from the event syntax. We can represent the transition relation of a program as the union of the transition relations of its individual events instead of using one monolithic transition relation. Then, we can perform the relational image computations using one event at a time. However, when events contain multiple disjuncts (e.g., from if-then-else constructs) we can use a finer-grained technique to further decompose the transition relation as described below.

DNF Event Decomposition Given two programs $P_1 = (V, I, E_1)$ and $P_2 = (V, I, E_2)$, with the same variables and initial states, assume the following equivalence holds:

$$\bigvee_{e \in E_1} R_e \equiv \bigvee_{e \in E_2} R_e$$

Then P_1 and P_2 are semantically equivalent, since they satisfy exactly the same set of temporal formulas. However, due to their structural differences (and to the methods used in model-checking), one program may be significantly more computationally expensive to analyze than the other.

As an heuristic we use the DNF decomposition for partitioning transition relations, i.e., given a program we partition its transition relation R as:

$$R \equiv \bigvee_{e \in E} R_e, \qquad R_e \equiv \bigwedge_i c_{ei}$$

where each c_{ei} is a single affine constraint or a single stride constraint over primed and unprimed program variables. Any transition relation R can be represented in this form by converting R into DNF, and then considering each disjunct as a separate event. For example, Figure 4.4 shows the DNF decomposition of our running program from Figure 4.3. Note that the transformation simply split the separate disjuncts from the original events e_S and e_R, and renamed them as separate events.

$$e_{IS} : pc = \mathit{Idle} \ \wedge \ pc' = \mathit{Send}$$

$$e_{S1} : pc = \mathit{Send} \ \wedge \ p' = p + 1 \ \wedge \ i_1' = i_1 + 1$$

$$e_{S2} : pc = \mathit{Send} \ \wedge \ p' = p + 1 \ \wedge \ i_2' = i_2 + 1$$

$$e_{SI} : pc = \mathit{Send} \ \wedge \ pc' = \mathit{Idle}$$

$$e_{R1} : i_1 > 0 \ \wedge \ i_1' = i_1 - 1 \ \wedge \ c' = c + 1$$

$$e_{R2} : i_2 > 0 \ \wedge \ i_2' = i_2 - 1 \ \wedge \ c' = c + 1$$

Figure 4.4: DNF Event Decomposition

State Space Partitioning

Let P be a partitioning of a program's state-space S, where

$$P = \{S_1, S_2, \ldots, S_p\}, \qquad S \equiv \bigvee_{i=1}^{p} S_i, \qquad i \neq j \Rightarrow \ S_i \wedge S_j \equiv \emptyset.$$

Then if $q \subseteq S$ is a subset of program states, we can decompose q via our partitioning P of S:

$$q \ \equiv \ \bigvee_{i=1}^{p} q_i, \quad \text{where} \quad q_i \ \equiv \ q \wedge S_i.$$

Intuitively, this technique would be worthwhile if each S_i unites states with common characteristics (that is, when key formulas are true either for all the states in S_i or none of the states in S_i), and when P distinguishes between large regions of S possessing different characteristics. Below we present two heuristics which can often create such a partitioning.

We also use the state space partitioning to refine the event decomposition. Assume that we decompose the events so that each event e has a **source** partition class S_i and a **destination** partition class S_j such that **domain**$(e) \subseteq S_i$ and **range**$(e) \subseteq S_j$. This decomposition can easily be computed by conjuncting the formulas representing the events with the formulas representing the partition classes. Then, we can represent each event with a Presburger formula R_e and two indices i and j, which indicate the source and the destination partition classes of the corresponding event. This representation is useful in the incremental computation of the relational images (discussed in Section 4.2.4).

Below we present two techniques for state space partitioning. One uses the valuations of the control variables and the other uses the domains of the events to partition the state space.

Control-Point Partitioning Given a program $P = (V, I, E)$, assume we partition the variables into two classes:

$$V = V_1 \cup V_2 \quad \text{s.t.} \quad V_1 \cap V_2 = \emptyset.$$

where the variables in V_1 have finite domains. The valuations for the variables in V_1 induce a natural partitioning of S. Letting $V_1 = \{v_{11}, \ldots, v_{1m}\}$, equivalence classes are defined simply as:

$$S_{(x_{11},\ldots,x_{1m})} \equiv \bigwedge_{j=1}^{m} v_{1j} = x_{1j} \quad \text{s.t.} \quad x_{1j} \in \mathbf{domain}(v_{1j}).$$

Of course, if V_1 contains many variables, or if their domains are large, we may end up with a huge number of classes; hence the partition variables have to be chosen with care.

Fortunately, many programs yield a natural choice for V_1: the control variables (e.g., program counters). Control points – valuations of the control variables – correspond to the partition classes generated by the above technique by choosing V_1 as the set of control variables. To enable this type of partitioning, the user must specify certain variables as control variables. When our preprocessor translates event-action language code into events, the control variables are isolated as such, which then allows a default partitioning according to their values.

Eliminating the Control Variables Given a variable with a small, finite domain (such as the program counter pc in the unbounded buffer example shown in Figure 4.3) we have two choices: 1) To map it to an integer variable and encode it using our Presburger formula representations (e.g., we can map pc to an integer variable by mapping its valuations to integer values such as $Idle = 0$, and $Send = 1$). 2) To eliminate the variable by partitioning the state space based on its valuations – the value of the variable can be inferred from the indices of the partition classes. The second approach saves us from generating arithmetic constraints about the eliminated variables in our Presburger formulas. Since the manipulation of Presburger formulas is computationally expensive, this is very useful.

We eliminate the control variables from our Presburger formula representations using the control-point partitioning. Given a set $q \equiv \bigvee_{i=1}^{p} q_i$ partitioned based on the control-point partitioning $P = \{S_1, S_2, \ldots, S_p\}$, we determine the values of the control variables in each q_i from the index of the corresponding partition class. Hence, given a q_i, we replace any occurrence of a control variable v in q_i with its value in the partition class S_i.

Additionally, by decomposing the events based on the control-point partitioning, we also eliminate the control variables from the events. Given an event e represented by the Presburger formula R_e, assume that its source partition class is S_i and its destination partition class is S_j. Then, for each control variable v, we replace any occurrence of v in R_e with its value in S_i. Similarly we replace any occurrence of v' in R_e with the value of v in S_j.

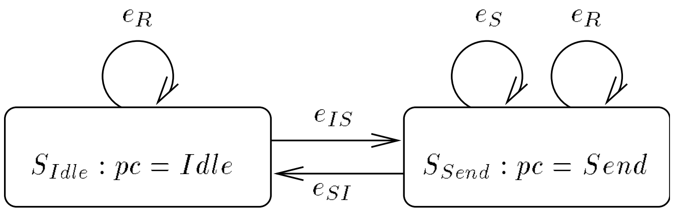


Figure 4.5: Control-Point Partitioning

Although control-point partitioning can be used without eliminating the control variables, in the experiments reported in this dissertation we eliminated the control variables every time we used the control-point partitioning.

When we apply the control-point partitioning to the program in Figure 4.3 with $V_1 = \{pc\}$ we get the following two partition classes (shown pictorially in Figure 4.5):

$$S_{Idle} \equiv pc = Idle \qquad S_{Send} \equiv pc = Send$$

and so, our partitioning is $P = \{S_{Idle}, S_{Send}\}$. Now consider a formula $q \equiv p > c$, representing all states in which more items were produced than consumed. The formula is partitioned as $P_q = \{q_{Idle}, q_{Send}\}$, where,

$$q_{Idle} \equiv pc = Idle \wedge p > c \qquad q_{Send} \equiv pc = Send \wedge p > c$$

with $q \equiv q_{Idle} \vee q_{Send}$.

Using this partitioning we can eliminate the variable pc from our state and transition representations. We can represent q_{Idle} and q_{Send} as $q_{Idle} \equiv p > c$ and $q_{Send} \equiv p > c$. The value of pc is determined by the index of the corresponding partition class. Similarly, for the event representations, the current and the next state values of the variable pc can be determined from the indices of the source and the destination partition classes. (In Figure 4.5, the source partition class of an event corresponds to the origin of the arrow representing the event,

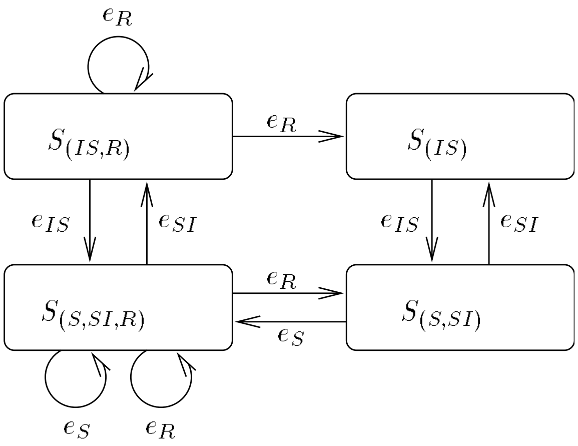


Figure 4.6: Event-Domain Partitioning

and the destination partition class corresponds to its target.) For example, we can represent the event e_{IS} simply as e_{IS} : **true** (corresponding to the identity relation) since we know that its source is S_{Idle} (i.e., $pc = Idle$), and its destination is S_{Send} (i.e., $pc' = Send$).

Event-Domain Partitioning The method above can also be generalized to the "implicit control points" inherent in the events themselves. In essence, an event's natural control point is just its enabling condition. And, using the enabling conditions as partitioning criterion can yield a much finer-grained decomposition. (This is obviously true when no control variables are isolated in a program.) In this partitioning, each class denotes a region of S in which a specific set of events are enabled to fire. So, given $P = (V, I, E)$, we define its event-domain partitioning P as:

$$P = \{\ S_i \mid \forall e \in E\ .\ (S_i \wedge \mathbf{domain}(e)) \equiv S_i\ \vee\ (S_i \wedge \neg\mathbf{domain}(e)) \equiv S_i\ \}$$

While this technique yields potentially $2^{|E|}$ partition classes, note that most of these partition classes will probably be empty. The reason is that most of the events will typically have mutually exclusive enabling conditions.

This is certainly the case when we apply the technique to our running example from Figure 4.3. Event-domain partitioning yields only 4 feasible partition classes (shown pictorially in Figure 4.6), which is significantly less than $2^{|E|} = 2^4 = 16$. These 4 partition classes are as follows (where each partition class is indexed by the events enabled in the class):

$$\begin{aligned}
S_{(IS,R)} &\equiv pc = \mathit{Idle} \ \wedge \ (i_1 > 0 \ \vee \ i_2 > 0) \\
S_{(S,SI)} &\equiv pc = \mathit{Send} \ \wedge \ i_1 = 0 \ \wedge \ i_2 = 0 \\
S_{(S,SI,R)} &\equiv pc = \mathit{Send} \ \wedge \ (i_1 > 0 \ \vee \ i_2 > 0) \\
S_{(IS)} &\equiv pc = \mathit{Idle} \ \wedge \ i_1 = 0 \ \wedge \ i_2 = 0
\end{aligned}$$

Note that if we had used the DNF event decomposition first (from Figure 4.4), we would have produced a finer partitioning with more classes.

4.2.4 A Presburger Arithmetic Model Checker for CTL

After generating our symbolic representations in terms of Presburger formulas, we use the Omega Library [KMP+95] to help symbolically compute the truth sets for the temporal properties at hand. The Omega Library includes a large collection of object classes to efficiently manipulate Presburger formulas; to date it has mainly been used in high-performance compilers, specifically for dependence analysis, program transformations, and detecting redundant synchronization [Pug92, PW94]. It implements all the functionality we need to build a symbolic model checker.

To symbolically compute the temporal operators, as we discussed in the previous chapter, we need functions for negation, conjunction, disjunction, satisfiability, and relational image computations. These functions can all be supported for Presburger formulas (as implemented in the Omega Library). We compute the functional EX incrementally, using the partitioning of the state space S and transition relation R. As discussed before, EX $q \equiv R^{-1}\ q$. Moreover, we can symbolically compute EX with respect to our program's partitioning, and maintain a formula for each partition class, as follows:

$$\begin{aligned}
\mathrm{EX}\ q &\equiv R^{-1}\ \Big(\bigvee_{S_i \in P} q \wedge S_i\Big) \\
&\equiv \bigvee_{S_i \in P} (R^{-1}\ \ (q \wedge S_i)) \\
&\equiv \bigvee_{S_i \in P} \Big(\Big(\bigvee_{e \in E} R_e\Big)^{-1}\ \ (q \wedge S_i)\Big) \\
&\equiv \bigvee_{S_i \in P} \bigvee_{e \in E} (R_e^{-1}\ \ (q \wedge S_i))
\end{aligned}$$

where P is the set of partition classes and E is the set of events. We can push the relational image computations inside the disjunctions because the existential quantification in the image computations distributes over disjunctions. By performing this computation individually for each partition class, we exploit the fact that many formulas inherently involve only small parts of the program's state space.

For example, consider the unbounded buffer example (Figure 4.3) – and specifically, the states where the producer is *Idle* and the buffers are empty:

$$q \ \equiv\ pc = \mathit{Idle}\ \wedge\ i_1 = 0\ \wedge\ i_2 = 0.$$

Using the control-point partitioning (Figure 4.5), we have $q \equiv q_{\mathit{Idle}}\ \vee\ q_{\mathit{Send}}$ where

$$\begin{aligned}
q_{\mathit{Idle}} &\equiv q\ \wedge\ S_{\mathit{Idle}} \equiv pc = \mathit{Idle}\ \wedge\ i_1 = 0\ \wedge\ i_2 = 0 \\
q_{\mathit{Send}} &\equiv q\ \wedge\ S_{\mathit{Send}} \equiv \mathbf{false}
\end{aligned}$$

and with the DNF event decomposition (Figure 4.4) we get

$$\begin{aligned}
\text{EX}\ q &\equiv R^{-1}\ (q_{Idle} \lor q_{Send}) \\
&\equiv R^{-1}\ q_{Idle} \\
&\equiv \bigvee_{e \in \{e_{R1}, e_{R2}, e_{SI}\}} (R_e^{-1}\ q_{Idle}) \\
&\equiv (R_{e_{R1}}^{-1}\ q_{Idle}) \lor (R_{e_{R2}}^{-1}\ q_{Idle}) \lor (R_{e_{SI}}^{-1}\ q_{Idle}) \\
&\equiv (\exists pc, i_1, i_2, c\ .\ i_1' > 0 \land i_1 = i_1' - 1 \land c = c' + 1 \land pc = pc' \\
&\quad \land pc = Idle \land i_1 = 0 \land i_2 = 0) \lor \\
&\quad (\exists pc, i_1, i_2, c\ .\ i_2' > 0 \land i_2 = i_2' - 1 \land c = c' + 1 \land pc = pc' \\
&\quad \land pc = Idle \land i_1 = 0 \land i_2 = 0) \lor \\
&\quad (\exists pc, i_1, i_2, c\ .\ pc' = Send \land pc = Idle \land i_1 = i_1' \land i_2 = i_2' \\
&\quad \land pc = Idle \land i_1 = 0 \land i_2 = 0) \\
&\equiv \quad pc' = Idle \land (i_1' = 1 \land i_2' = 0 \lor i_2' = 1 \land i_1' = 0) \lor \\
&\quad\quad pc' = Send \land i_1' = 0 \land i_2' = 0
\end{aligned}$$

As we discussed previously, the relational image computations correspond to existential quantifier elimination. Since we are using the inverse transition relations, the primed variables are replaced with the corresponding unprimed variables (and visa versa) in the event formulas. Although we do not write them explicitly in our event descriptions, the constraints which show that a variable preserves its value (e.g., $pc = pc'$) show up in the actual Presburger formula representations. (To simplify the above computation we did not show the constraints on the variable p.) Note that, in the last step we could replace the primed variables with the unprimed variables since the result is a set.

We want to clarify a couple of points about the implementation of the symbolic computation above. First, the result of the EX $q \equiv q'$ is also stored in a partitioned format, i.e., $q' \equiv q'_{Idle} \lor q'_{Send}$ where

$$\begin{aligned} q'_{Idle} &\equiv pc = Idle \land (i_1 = 1 \land i_2 = 0 \lor i_2 = 1 \land i_1 = 0) \\ q'_{Send} &\equiv pc = Send \land i_1 = 0 \land i_2 = 0 \end{aligned}$$

Secondly, the program counter pc can be eliminated as we discussed above – its value can be determined using the indices of the partition classes. In fact, in all the experiments reported in this dissertation we eliminated the control variables whenever we used the control-point partitioning. Finally, in our implementation we use the sources and the destinations of the events to find out which events are applicable to which partition classes. In the above example, since q only refers to the partition class S_{Idle}, we only need to consider the events with the destination S_{Idle}, i.e., events e_{R1}, e_{R2} and e_{SI}. Similarly, by looking at the sources of the events we can determine which partition class the results should be written. For example the results of the image computations of the events e_{R1} and e_{R2} are written to the partition class S_{Idle} since their source is S_{Idle}. On the other hand, the result of the image computation of the event e_{SI} is written to the partition class S_{Send} since its source is S_{Send}.

Using Presburger formula representation and the techniques described above we built a CTL model checker as shown in Figure 4.7. Given a CTL formula, this model checker computes a Presburger formula representation for its truth set. Note that the model checking procedure uses the least fixpoint semantics of the CTL operators EU and AU to compute their truth sets. Using the equivalences among CTL temporal operators this procedure can compute the truth set of any temporal property. Moreover, we can easily implement a μ-calculus model checker that uses Presburger formulas using the algorithm given in Figure 4.2.

```
Procedure Evaluate($f$)
 Case
  IsEvaluated($f$)       : Return($f$)
  $f = \neg f_1$          : Return(Not(Evaluate($f_1$)))
  $f = f_1 \wedge f_2$    : Return(And(Evaluate($f_1$),Evaluate($f_2$)))
  $f = f_1 \vee f_2$      : Return(Or(Evaluate($f_1$),Evaluate($f_2$)))
  $f = \mathrm{EX}\ f_1$  : Return(EX(Evaluate($f_1$)))
  $f = \mathrm{AX}\ f_1$  : Return(Not(EX(Not(Evaluate($f_1$)))))
  $f = f_1\ \mathrm{EU}\ f_2$ : $q_0 \equiv f_2 \quad q_{i+1} \equiv$ Evaluate($q_i \vee (f_1 \wedge \mathrm{EX}\ q_i)$)
                          Return($q_n$) when $q_n \equiv q_{n+1}$
  $f = f_1\ \mathrm{AU}\ f_2$ : $q_0 \equiv f_2 \quad q_{i+1} \equiv$ Evaluate($q_i \vee (f_1 \wedge \mathrm{EX}\ q_i \wedge \mathrm{AX}\ q_i)$)
                          Return($q_n$) when $q_n \equiv q_{n+1}$
```

Figure 4.7: A Symbolic Model Checker for CTL

4.2.5 Experiments: the Unbounded Buffer and the Bakery Algorithm

We analyzed the unbounded buffer given in Figure 4.3 using control-point partitioning and the model checking procedure given in Figure 4.7. Recall that one of the requirements for this program was $\mathrm{AG}(p = c + i_1 + i_2)$ which is equivalent to: $\neg(\textbf{true}\ \mathrm{EU}\ (p \neq c + i_1 + i_2))$ or $\neg\mathrm{EF}(p \neq c + i_1 + i_2)$. (Although our basic temporal operators are EU and AU, when possible we use their EF and AF equivalents for clarity of presentation). To compute the least fixpoint $\mathrm{EF}(p \neq c + i_1 + i_2)$, the model checker initialized the first iterate to $q_0 \equiv p \neq c + i_1 + i_2$. The fixpoint computation converged after one iteration to q, where q is partitioned as follows:

$$\begin{aligned} q_{Idle} &\equiv pc = Idle \ \wedge \ p \neq c + i_1 + i_2 \\ q_{Send} &\equiv pc = Send \ \wedge \ p \neq c + i_1 + i_2 \end{aligned}$$

Our top-level formula is $\neg \mathrm{EF}(p \neq c + i_1 + i_2)$; hence, the model checker computes $\neg q$. This yields the set of states which can never reach a violation of the assertion $p = c + i_1 + i_2$. Since the initial condition for the program is $I \equiv p = 0 \ \wedge \ c = 0 \ \wedge \ i_1 = 0 \ \wedge \ i_2 = 0$, it is easy to see that $I \subseteq \neg q$ (i.e., all of the initial states satisfy the safety property). Hence, the model checker reports that the property is proved (for a total computation time of 0.88 seconds on a Sun SPARCstation 5). The model checker also successfully proved the property $\mathrm{AG}((pcs = Idle \ \wedge \ i_1 + i_2 = n) \rightarrow \mathrm{AX}(i_1 + i_2 \leq n))$ in 0.77 seconds.

We also analyzed the bakery algorithm (Figure 3.1) with the model checking procedure given in Figure 4.7 using the control-point partitioning. The mutual-exclusion requirement for the bakery algorithm is equivalent to: $\neg \mathrm{EF}(pc_1 = C_1 \ \wedge \ pc_2 = C_2)$. To compute the least fixpoint $\mathrm{EF}(pc_1 = C_1 \ \wedge \ pc_2 = C_2)$, the model checker initialized the first iterate to $q_0 \equiv pc_1 = C_1 \ \wedge \ pc_2 = C_2$. After 4 iterations, the fixpoint computation converged to a set q, where q is partitioned as follows:

$$\begin{aligned}
q_{(T_1,T_2)} &\equiv pc_1 = T_1 \ \wedge\ pc_2 = T_2 \ \wedge\ \textbf{false} \\
q_{(T_1,W_2)} &\equiv pc_1 = T_1 \ \wedge\ pc_2 = W_2 \ \wedge\ b = 0 \\
q_{(T_1,C_2)} &\equiv pc_1 = T_1 \ \wedge\ pc_2 = C_2 \ \wedge\ b = 0 \\
q_{(W_1,T_2)} &\equiv pc_1 = W_1 \ \wedge\ pc_2 = T_2 \ \wedge\ a = 0 \\
q_{(W_1,C_2)} &\equiv pc_1 = W_1 \ \wedge\ pc_2 = C_2 \ \wedge\ (b = 0 \ \vee\ a < b) \\
q_{(C_1,T_2)} &\equiv pc_1 = C_1 \ \wedge\ pc_2 = T_2 \ \wedge\ a = 0 \\
q_{(C_1,W_2)} &\equiv pc_1 = C_1 \ \wedge\ pc_2 = W_2 \ \wedge\ (a = 0 \ \vee\ b < a) \\
q_{(C_1,C_2)} &\equiv pc_1 = C_1 \ \wedge\ pc_2 = C_2 \ \wedge\ \textbf{true} \\
q_{(W_1,W_2)} &\equiv pc_1 = W_1 \ \wedge\ pc_2 = W_2 \ \wedge \\
&\quad (a = b = 0 \ \vee\ a = 0 \ \wedge\ 1 \leq b \ \vee\ b = 0 \ \wedge\ 1 \leq a)
\end{aligned}$$

Although, the values of the control variables pc_1 and pc_2 are shown in the above representation, they are not encoded in our Presburger formula representations. These variables are eliminated using the control-point partitioning. Their values are inferred from the indices of the corresponding partition classes.

To obtain the truth set of $\neg\mathrm{EF}(pc_1 = C_1 \ \wedge\ pc_2 = C_2)$, the model checker computes $\neg q$ using the result above. This yields the set of states which can never reach a violation of the mutual-exclusion property. The set of initial states for the bakery algorithm is $I \equiv pc_1 = T_1 \ \wedge\ pc_2 = T_2 \ \wedge\ a = b = 0$, and we see that that $I \subseteq \neg q$ (i.e., all of the initial states satisfy the safety property). Hence, the model checker reports that the property is proved (for a total computation time of 2.85 seconds on a Sun SPARCstation 5).

The model checker in Figure 4.7 also proved the starvation-freedom property, $\mathrm{AG}(pc_1 = W_1 \rightarrow \mathrm{AF}(pc_1 = C_1))$, which is equivalent to $\neg\mathrm{EF}(pc_1 = W_1 \ \wedge\ \neg\mathrm{AF}(pc_1 = C_1))$. The inner (AF) and outer (EF) fixpoint computations converged in 9 and 1 iterations, respectively (with a total computation time of 7.64 seconds).

4.2.6 Related Work

As we discussed in Chapter 2, similar symbolic representations are used in analyzing hybrid systems [ACHH93, ACH+95, AHH96]. A hybrid system is a discrete program which interacts with an analog environment. It is modeled as a finite automaton with real variables. Values of these variables continuously evolve according to some dynamical laws. The semantics of hybrid systems can be defined as infinite transition systems and their properties can be expressed using real-time temporal logics [ACH+95, AHH96]. The symbolic model checker HyTech represents hybrid systems using formulas in real arithmetic with addition [AHH96]. This is the analogue of Presburger arithmetic for real domains. HyTech uses Mathematica as its symbolic manipulator [AHH96]. The fundamental difference with our techniques is that we encode sets of integers as opposed to real numbers used in hybrid systems.

In [BW94], Boigelot and Wolper propose a representation for complex periodic sets of integer values, and use it for reachability analysis. Periodic sets are sets which can be expressed using existential quantification and affine constraints. They restrict the events in their input systems to linear transformations. Hence, repeated consecutive execution of these events create periodic sets. As an example, consider the event $x' = x + 2 \ \wedge \ x < 1000$. Starting from an initial state $x = 0$, repeated execution of this event will create a reachable set $\exists k \geq 0 \ . \ x = 2k \ \wedge \ k \leq 500$. The representation discussed in [BW94] can encode such sets. Algorithms for manipulating this representation for reachability analysis are also given. Our Presburger formula representation subsumes the representation given in [BW94]. We also developed tools for model checking all CTL properties, whereas in [BW94] only reachability properties are discussed.

It is possible to encode integer sets representable by Presburger formulas using finite automata [Kla97, WB95]. The main idea is to represent integers as finite bit-strings. Then, for each Presburger formula, an automaton can be constructed that will accept the strings that correspond to integer values which make the Presburger formula true. There are algorithms for manipulating finite automata that support the functionality required for state space exploration [WB95]. Hence, it is possible to construct a model checker using this symbolic representation. We could integrate such a representation to our tool by replacing our constraint manipulator Omega Library with a library for manipulating finite automata. It would be interesting to compare two representations and investigate their effectiveness for different problem domains. Actually, in [SKR98], Shiple *et al.* do exactly this for reachability analysis of extended finite-state machines. They report that neither approach is consistently superior for the examples they analyzed.

4.2.7 Limitations of Presburger Arithmetic Model Checking

Although Presburger model checking approach enabled us to attack problems which are not analyzable by finite-state model checkers it has its own limitations:

- Presburger formula encoding is not efficient for encoding Boolean variables. Because of the high complexity of manipulating Presburger formulas, the naive approach of mapping Booleans to integers (e.g., positive means **true**, negative means **false**) does not scale.

- Model checking is undecidable for the set of infinite transition systems we analyze. Hence, fixpoint computations in our symbolic model checker are not guaranteed to converge.

We elaborate on these issues below.

Encoding Boolean and Enumerated Variables Our Presburger technique proved ill-suited for handling Boolean and (unordered) enumerated types. When all state sets are represented as Presburger constraint expressions, all Boolean variables are mapped to integers – which ends up being extremely wasteful. Since programs usually include many such variables, and since model checkers often generate large sets of states, this wastage quickly adds up to a major obstacle. Also, while both Boolean and Presburger formulas afford compact encodings with their operators, there is no natural mapping between them which preserves this compactness. Consider, for example, a set of affine constraints over n integers, which compactly describes all states lying within some n-dimensional polytope. The benefit of this encoding extends directly from the expressiveness of the arithmetic inequality operators – which are useless over 2-valued domains like Booleans, and marginally less so for small enumerated types like $\{red, green, blue\}$.

The other approach – eliminating the Boolean and the enumerated variables by partitioning the state space based on their valuations – does not scale to large numbers of variables. The number of partition classes generated with this approach increases exponentially with the number of eliminated variables. (d^n partition classes are generated where d is the size of the domains of the eliminated variables and n is the number of eliminated variables.) Since in our state representations we use one Presburger formula per partition class, if there are more than several variables this technique ends up generating too many Presburger formulas.

Undecidability of Infinite State Model Checking Note that, by restricting the expressions in our event-action language to Presburger formulas we defined a new set of transition systems. Let us call these transition systems **Presburger machines**. I.e., a Presburger machine is a transition system that can be specified in our event-action language, where the initial condition and the events are expressed as Presburger formulas. We have the following property:

Theorem 4.2 *A Presburger machine can simulate an arbitrary Turing machine.*

This property is easy to show using counter machines [HU79]. Counter machines are restricted Turing machines that consist of finite number of unbounded counters which can be incremented, decremented and checked for equivalence to zero. Counter machines can simulate arbitrary Turing machines. It is easy to see that counter machines can be simulated using Presburger machines.

Since Presburger machines can simulate Turing machines, and since we can express that a machine halts using CTL (EF *terminate*, where *terminate* is a Boolean variable true for all terminating states) we have the following property:

Theorem 4.3 *Determining if a Presburger machine satisfies a CTL formula is undecidable.*

Note that we also have the following property:

Theorem 4.4 *Computing a Presburger formula representation for the truth sets of CTL properties of Presburger machines is undecidable.*

This follows from the Theorem 4.3. If we can compute a Presburger formula representation for the truth set of an arbitrary CTL property p, then we can check if a Presburger machine satisfies p by checking if $I \subseteq p$ using the Presburger formula representation of the initial states I. Since this is undecidable, Theorem 4.4 follows. This means that the fixpoint computations for temporal operators EU and AU given in Figure 4.7 may not always converge, i.e., the procedure may keep iterating forever without reaching a fixpoint. Thus, we need conservative approximation methods, which will always converge. We define a conservative analyzer as one which always terminates and never yields a spurious result, but may not be able to produce a definite answer in certain cases.

Indeed, our exact analyzer diverged when we fed the unbounded-buffer program, along with its requirement: AG($p \geq c$). When the exact analyzer went to work on this property, it attempted to symbolically enumerate ways that p could be less than c. Since c is unbounded, this method failed to converge. But as we show in the following chapters, the same property was easily proved using a conservative approximation.

In the following two chapters we present the methods we developed for addressing the problems discussed above: Composite symbolic encodings and conservative approximation techniques.

Chapter 5

Model Checking with Composite Symbolic Representations

In this chapter we present a model checker which combines the relative strengths of both BDDs and Presburger formulas. We present a general framework for combining multiple type-specific encodings. Specifically, we use a composite model to represent a formula's valuations – which are encoded using tuples of Boolean logic and Presburger formulas, depending on the variables appearing in the subformulas. Our strategy can easily include other symbolic encodings as well – provided that the language is closed under Boolean connectives, propositions can be checked for satisfiability, and relational images can be computed, i.e., any adequate language can be included in this framework.

The key to our framework rests on some fundamental observations. Given a system whose state transformations (and requirements) exclude (1) arbitrary functions over mixed types, and (2) type-coersions (e.g., to allow using Booleans in arithmetic operators), we (a) orthogonally partition the system's state-space into sets of conjuncts, each of which contain a Boolean part and an integer part; (b) manipulate a Boolean state and its transformation independently, without interference from the integer part (and vice versa); and also (c) use semantically sound rules for handling the logical connectives that involve multiple types.

In formal terms, this partitioning allows relational image computations to distribute across the conjunction operator. We exploit this property in our implementation, which is structured in a layered class hierarchy (Appendix A). The foundation is composed of two libraries for encoding type-specific constraints, and their associated set-theoretic operations. While both share a similar API, one of them (our BDD implementation) is used exclusively for Boolean and enumerated types, while the other is the Omega Library, used for integer-valued variables and their constraints. At the next level is our composite-model library, which handles operations over mixed-type constraints (e.g., set-inclusion, intersection, etc.); in turn, these operations invoke their relevant type-specific counterparts to help carry out the desired effect. At the topmost level is the model checker's class library, which imports the composite-model operations, and exports functions for model checking and reachability analysis. Our approach to mixed constraints – and their orthogonal implementations – will hopefully allow us to expand to additional datatypes in the future.

Our preliminary results regarding the contributions discussed in this chapter have been reported in [BGL98] and [BG98].

5.1 Composite Representations

Given a concurrent system $P = (V, I, E)$, each variable $v \in V$ has a type $t_v \in T$ where T denotes the set of variable types. We assume that for each type $t \in T$, there is a symbolic assertional language L^t, which encodes formulas over t-typed variables and constants. For the sake of carrying out model checking, we assume that any type language L^t is an adequate language. Specifically, we have implemented a model checker using two adequate languages – Presburger arithmetic (implemented with the Omega Library [KMP+95]), and BDDs (implemented in our own class library).

More generally, given a set of variable types $t \in T$ and their associated adequate languages L^t, we define a **composite language** L^c to be generated by the following grammar:

$$\mathrm{CF} \ ::= \ \mathrm{CF} \ \vee \ \mathrm{CF} \mid \mathrm{CF} \ \wedge \ \mathrm{CF} \mid \neg\mathrm{CF} \mid \mathrm{form}^t$$

where $t \in T$ and terminal form^t denotes a formula in language L^t, and nonterminal CF denotes a **composite formula** in L^c.

We use this composite language to represent the basic properties BP, i.e., $f \in BP$ only if $f \in L^c$. We also assume that the set of initial states I can be represented as a composite formula, i.e., $I \in L^c$.

As with single-type symbolic model checking, we use the formulas in L^c to symbolically encode composite state sets. We convert any composite state formula $q \in L^c, q \subseteq S$ to a disjunctive form, and represent it as follows:

$$q \ \equiv \ \bigvee_{i=1}^{n_q} \ \bigwedge_{t \in T} q_i^t$$

where each $q_i^t \in L^t$, and n_q denotes the number of disjuncts needed. Such a disjunctive form can be obtained for any composite term, since we do not allow functions (or predicates) with arguments which have different types.

5.1.1 Logical Operations on Composite Representations

Assume that we have two state sets p and q represented symbolically as

$$p \equiv \bigvee_{i=1}^{n_p} \bigwedge_{t \in T} p_i^t \quad \text{and} \quad q \equiv \bigvee_{i=1}^{n_q} \bigwedge_{t \in T} q_i^t$$

where each $p_i^t, q_i^t \in L^t$. Now we show how to compute logical operators disjunction, conjunction and negation on p and q:

$$\begin{aligned} p \vee q &\equiv (\bigvee_{i=1}^{n_p} \bigwedge_{t \in T} p_i^t) \vee (\bigvee_{i=1}^{n_q} \bigwedge_{t \in T} q_i^t) \\ p \wedge q &\equiv \bigvee_{i=1,j=1}^{n_p,n_q} \bigwedge_{t \in T} p_i^t \wedge q_j^t \\ \neg q &\equiv \bigvee \bigwedge_{j=1}^{n_q} \neg q_j \;\; \text{where} \;\; q_j = q_j^t \;\; t \in T \end{aligned}$$

In other words, disjunction just requires appending the two disjuncts together; the result will be in our composite symbolic form. Conjunction is computationally more expensive. Using the distributive properties of Boolean algebra, we can compute all the pertinent disjuncts – yet we may end up with $n_p \times n_q$ disjuncts, which we have to compute by traversing the disjunctive representations of p and q.

Finally, taking a complement is more expensive; indeed, complementation of a set q with n_q disjuncts may, in fact, create $|T|^{n_q}$ disjuncts in the worst case – however it is very likely that most of these will be empty. Hence, we build $\neg q$ in an incremental manner so that we try to minimize the number of disjuncts generated. We do this by testing for emptiness on the fly, while we are computing the conjunctions.

During model checking operations, the number of disjuncts in a composite formula can easily increase. As we showed above, applying the disjunction operation is relatively cheap – yet it can still linearly increase a formula's complexity.

And this problem gets worse when applying conjunction (with quadratic growth) and even more so with negation (and its worst-case exponential growth). In practice, however, most type-specific symbolic libraries contain their own simplification procedures, which can minimize the number of formulas used to represent a set of valuations. (This is true for both the BDD and the Presburger libraries we use.) So, for composite models, the challenge lies in merging as many terms as possible into a single-type format, and still retaining the semantics of the original formula. To do this we use some simple reduction rules. Given a composite formula with two disjuncts $q \equiv (\bigwedge_{t \in T} q_1^t) \vee (\bigwedge_{t \in T} q_2^t)$ we have the following properties:

$$\begin{aligned} \forall s \in T \quad s \neq t \quad q_1^s \equiv q_2^s \quad &\rightarrow \quad q \equiv (\bigwedge_{s \in T, s \neq t} q_1^s) \wedge (q_1^t \vee q_2^t) \\ \forall t \in T \quad q_1^t \subseteq q_2^t \quad &\rightarrow \quad q \equiv \bigwedge_{t \in T} q_2^t \end{aligned}$$

Note that in both cases we can reduce the formula from two disjuncts to one. Hence, to simplify a general composite formula we (1) check all pairs for the conditions listed above, and (2) merge the appropriate disjuncts when a condition is satisfied.

5.1.2 Composite Transitions

Given a system $C = (V, I, E)$ we assume that every atomic event $e \in E$ can be conjunctively partitioned based on the variable types. Any system which does not allow type-casting or multi-typed functions would satisfy this requirement. Then, for an event e, we can represent its transition relation R_e as $R_e \equiv \bigwedge_{t \in T} R_e^t$, where each $R_e^t \in L^t$. Hence, we can symbolically encode the entire transition relation R as:

$$R \;\equiv\; \bigvee_{e\in E} R_e \;\equiv\; \bigvee_{e\in E}\bigwedge_{t\in T} R_e^t$$

Now we state the fundamental property which enables us to manipulate the type specific symbolic encodings independently:

$$\begin{aligned} R_e \;\; (\bigwedge_{t\in T} q_i^t) &\equiv (\bigwedge_{t\in T} R_e^t) \;\; (\bigwedge_{t\in T} q_i^t) \\ &\equiv \bigwedge_{t\in T}(R_e^t \;\; (\bigwedge_{t\in T} q_i^t)) \\ &\equiv \bigwedge_{t\in T}(R_e^t \;\; q_i^t) \end{aligned}$$

The first step follows from the fact that each R_e^t only references variables of type t, i.e., we can push the existential quantification for the image computation inside the first conjunction. The second step follows from the fact that each q_i^t only references variables of type t, i.e., we can push the existential quantification for the image computation inside the second conjunction. Taken as a whole, the property shows that the image computation for different variable types are orthogonal, and hence they can be computed separately.

The Boolean operations we discussed in the previous section and the image computations we discussed in this section show that the composite formulas form an adequate language. Hence, we can build a symbolic model checker using composite representations.

5.2 Composite Model Checker

We built a composite model checker for CTL using the ideas developed in the previous sections. The model checker is exactly the same as the CTL model checker given in Figure 4.7, except it uses composite representations composed of BDDs and Presburger formulas as its symbolic representation instead of only Presburger formulas. Again, it is possible to implement a μ-calculus model checker using the same ideas.

The functional EX is implemented with respect to the system's event decomposition, and the composite representation of $q \equiv \bigvee_{i=1}^{n_q} \bigwedge_{t \in T} q_i^t$:

$$\begin{aligned}
\text{EX } q &\equiv \bigvee_{e \in E} R_e^{-1}\ q \\
&\equiv \bigvee_{e \in E} ((\bigwedge_{t \in T} (R_e^t)^{-1})\ \ (\bigvee_{i=1}^{n_q} \bigwedge_{t \in T} q_i^t)) \\
&\equiv \bigvee_{e \in E} \bigvee_{i=1}^{n_q} \bigwedge_{t \in T} ((R_e^t)^{-1}\ \ q_i^t)
\end{aligned}$$

which follows from results developed in the previous section.

If all types are finite, the composite CTL model checker will always converge. However, if integers (or other unbounded types) are involved, this procedure may well be a partial function, as it is with our prototype tool – which uses BDDs and Presburger arithmetic over unbounded transition systems. Hence, we often appeal to conservative approximation techniques, which can help when exact results are unobtainable.

We demonstrate the effectiveness of our composite model checker on two example systems, a requirements specification for a nuclear reactor's water pressure system, and a transport protocol. These experiments are discussed in Chapter 8. Because of the high number of integer and Boolean variables, our Presburger arithmetic model checker ran out of memory when we used it to analyze the temporal properties of these systems. However, our composite model checker analyzed them successfully.

5.3 Related Work

We do not know any other work where different symbolic representations are used together in one model checker, but in recent years various symbolic encodings have been proposed which are efficient for certain variable types. We summarize some of these representations below.

As we discussed before, HyTech a symbolic model checker for hybrid systems encodes real domains using affine constraints on real variables [ACHH93, ACH$^+$95]. This encoding is very useful for analyzing real time systems and real time temporal logic properties.

Queue content Decision Diagrams (QDDs) are proposed to encode sets of queue-configurations [BG96, BGWW97], and are used to carry out reachability queries on communicating state machines. Queue configurations are modeled via deterministic finite-state automata structures, where the language accepted by the automata is equal to the set of queue-contents. QBDDs extend QDDs, by combining QDD representation with BDD encoding [GL96]. QBDDs have limited expressiveness for infinite sets. They are more appropriate for encoding bounded queues.

The MONA tool is a library for manipulating automata-based representations of formulas in WS1S (Weak Second order theory of One Successor) [BK95, BKR96, HJJ$^+$95, Kla97]. WS1S corresponds to regular languages (languages accepted by finite automata) and subsumes a fragment of arithmetic (including Presburger arithmetic). The name MONA comes from monadic second order logic on finite strings, which can also be represented by the MONA tool using a different semantics for the automata representation. The tool is based on some efficient algorithms for minimizing finite-state automata that uses BDDs to represent transition functions in compressed form. QBDD representation corresponds to a special case of the automata-based representation used in MONA [BKR96].

Several symbolic representations have been proposed for modeling functions over Boolean variables with integer ranges, including Multi-Terminal Binary Decision Diagrams (MTBDDs) [CFZ95], Binary Moment Diagrams (BMDs) [BC95], and their generalization Hybrid Decision Diagrams (HDDs) [CFZ95, CZ95]. These are especially useful for datapath circuit verification, since they can encode functions that map Boolean vectors to integers [CZ95].

Each of the symbolic representations described above can provide efficient encodings for certain variable types. We believe our techniques for combining different symbolic encodings could, with little effort, be extended to all abovementioned representations – with the result being a general-purpose model checker.

Chapter 6

Conservative Approximations

Model checking is undecidable for the set of infinite-state systems we analyze. This implies that the fixpoint computations in our symbolic model checker are not guaranteed to converge. In this chapter, we show that we do not always need to compute the fixpoints exactly to prove or disprove temporal properties. We use conservative approximation techniques. In conservative analysis, given a temporal property, we either 1) prove the property, or 2) disprove the property, or 3) report that the analysis is inconclusive. Note that approximation techniques refer to approximations of the fixpoints. Since our techniques are conservative, the proofs and the counter-examples are always exact.

We reported our initial results about conservative analysis in [BG98, BGP96, BGP97] and [BGP98].

6.1 Conservative Analysis

If we cannot directly compute the truth set of a property f for a transition system $T = (S, I, R)$, the next best thing is to generate a **lower bound** for f, denoted f^-, such that $f^- \subseteq f$. Then if we determine that the set of initial

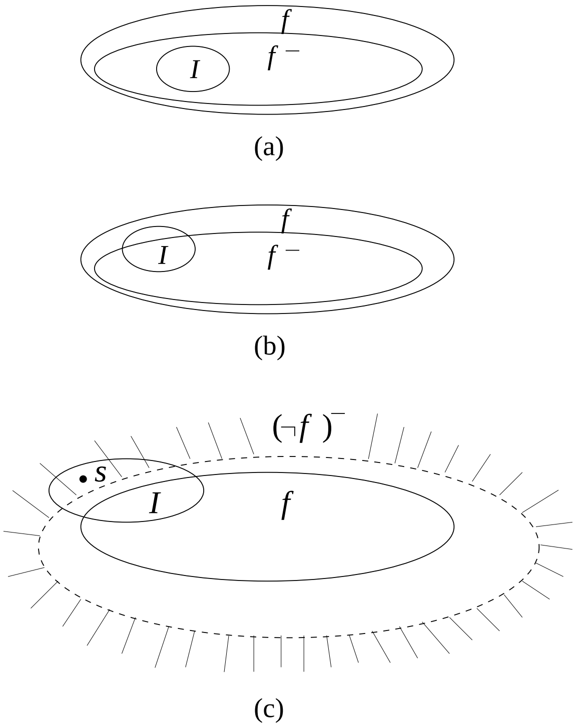


Figure 6.1: Lower Bounds for the Truth Set of a Temporal Property and Its Negation

states is included in this lower bound (i.e., $I \subseteq f^-$), we have also achieved our objective – that $I \subseteq f$, i.e., we proved that $T \models f$ (Figure 6.1 (a)). However, if $I \not\subseteq f^-$, we cannot conclude anything because it can be a **false negative** (Figure 6.1 (b)). In that case we can compute a lower bound for the negated property: $(\neg f)^-$. If we can find a state s such that $s \in I \cap (\neg f)^-$, then we can generate a counter example which would be a **true negative** (Figure 6.1 (c)). If both cases fail, i.e., both $I \not\subseteq f^-$ and $I \cap (\neg f)^- \equiv \emptyset$, then the analyzer cannot report a definite answer.

Since we seek to carry out our analysis in a recursive manner (as in the exact analyzers in Figure 4.7 and Figure 4.2), we have to compute an approximation to a formula by first computing approximations for its subformulas. Hence, to compute a lower bound to a property like $g \equiv \neg h$, we first need to compute an **upper bound** h^+ for the subformula h, and then let $g^- \equiv S - h^+$. This follows directly from set theory, since $(\neg f)^- \equiv \neg(f^+)$ and $(\neg f)^+ \equiv \neg(f^-)$. Thus, we need algorithms to compute both lower and upper bounds of temporal formulas.

When analyzing a negation-free formula, the compositionality of an approximation follows directly from the fact that all operators other than "$\neg$" are monotonic. This means that any lower/upper approximation for a negation free formula can be computed using the corresponding lower/upper approximation for its subformulas. As for handling arbitrary levels of negation, we can easily generalize the above mentioned method for outermost negation operators. That is, to approximate a temporal formula f, the following procedure determines which of f's subformulas require an upper bound, and which require a lower bound.

1. Mark the root of the parse tree for formula f with a minus sign ("$-$") if a lower bound is desired, and with a plus sign ("$+$") if an upper bound is desired.

2. Using a preorder tree traversal, visit each node in the tree, mark each node with the mark of its parent, unless its parent is a $\neg$ operator. In that case, mark the node with the opposite bound.

6.2 Computing Upper Bounds Using Presburger Representations

In this section we discuss computing upper bounds for least fixpoints using Presburger formula representations. When the procedure in Figure 4.7 attempts to compute fixpoints for EU and AU, it may generate sequences of increasing lower bounds which never converge. From Theorem 3.1, we know that a least fixpoint exists – but it may simply not be computable. Hence, our goal is to accelerate the computation, and "leap-frog" over multiple members of the chain – perhaps at the risk of over-shooting the exact least fixpoint. As long as the result is larger than the exact fixpoint, we have an upper approximation.

The way we go about this is as follows. If the **exact** iteration sequence generated by the procedure in Figure 4.7 is $q_0, q_1, q_2, \ldots$, then we find a **majorizing** sequence $\hat{q}_0, \hat{q}_1, \hat{q}_2, \ldots$, such that (1) for each i, $q_i \subseteq \hat{q}_i$, and (2) the $\hat{q}_i$ sequence reaches a fixpoint after finitely many iterates. Thus the fixpoint of the $\hat{q}_i$'s is an upper **approximation** to the fixpoint of the q_i's.

To generate the $\hat{q}_i$'s, we currently adopt a method developed by Cousot and Cousot, within the framework of abstract interpretation [CC77]. That is, we define an operator called **widening** (represented as $\bigtriangledown$), which majorizes the union computation:

$$\text{Given a pair of sets } p,\ p' \text{ such that } p \subseteq p' \quad p \cup p' \ \subseteq\ p \bigtriangledown p'.$$

Recall that, the fixpoint iteration sequence generated by the procedure in Figure 4.7 to compute the truth set of f_1 EU f_2 is:

$$q_i \equiv \begin{cases} f_2 & \text{if} \quad i = 0 \\ q_{i-1} \ \vee \ (f_1 \ \wedge \ (\text{EX } q_{i-1})) & \text{if} \quad i > 0 \end{cases}$$

$$f_1 \text{ EU } f_2 \equiv q_n \text{ when } \hat{q}_n \equiv \hat{q}_{n+1}$$

We call this sequence the exact fixpoint computation for f_1 EU f_2. Using a suitable widening operator, we can compute an upper bound for f_1 EU f_2 as:

$$\hat{q}_i \equiv \begin{cases} q_i & \text{if} \quad 0 \le i \le s \\ \hat{q}_{i-1} \bigtriangledown (\hat{q}_{i-1} \vee (f_1 \quad \wedge \quad (\text{EX } \hat{q}_{i-1}))) & \text{if} \quad i > s \end{cases}$$

$$(f_1 \text{ EU } f_2)^+ \equiv \hat{q}_n \text{ when } \hat{q}_n \equiv \hat{q}_{n+1}$$

where $q_0, q_1, q_2, \ldots$, is the sequence generated by the exact fixpoint computation shown above, and s is the **seed** of the widening sequence. The first s iterations of the approximation sequence are equal to the first s iterations of the exact sequence. The iterate s is the first iterate in the sequence (seed) that the widening operator is applied to. From the monotonicity of the EX operator, one can easily show by induction that iterates of this sequence do indeed majorize the q_i's computed by the exact fixpoint computation. And when this sequence terminates, the final iterate is an upper bound for f_1 EU f_2.

For the property f_1 AU f_2, the exact fixpoint computation implemented in the procedure in Figure 4.7 generates the following sequence:

$$q_i \equiv \begin{cases} f_2 & \text{if} \quad i = 0 \\ q_{i-1} \ \vee \ (f_1 \ \wedge \ (\text{EX } q_{i-1}) \ \wedge \ (\text{AX } q_{i-1})) & \text{if} \quad i > 0 \end{cases}$$

$$f_1 \text{ AU } f_2 \equiv q_n \text{ when } \hat{q}_n \equiv \hat{q}_{n+1}$$

We can generate a majorizing sequence for f_1 AU f_2 as:

$$\hat{q}_i \equiv \begin{cases} q_i & \text{if} \quad 0 \le i \le s \\ \hat{q}_{i-1} \bigtriangledown (\hat{q}_{i-1} \vee (f_1 \quad \wedge \quad (\text{EX } \hat{q}_{i-1}) \ \wedge \ (\text{AX } \hat{q}_{i-1}))) & \text{if} \quad i > s \end{cases}$$

$$(f_1 \text{ AU } f_2)^+ \equiv \hat{q}_n \text{ when } \hat{q}_n \equiv \hat{q}_{n+1}$$

where $q_0, q_1, q_2, \ldots$, is the sequence generated by the exact fixpoint computation given above.

Our goal is to find a widening operator which (1) yields a suitable (i.e., reasonably tight) upper bound for union, and (2) forces the $\hat{q}_i$ sequences to converge. In defining our widening operator, we generalized a technique used by Cousot and Halbwachs in [CH78]. The idea is to guess the direction of growth in the model checker's q_i iterates, and to extend the successive iterates in these directions.

6.2.1 A Widening Operator for Convex Sets

Cousot and Halbwachs defined a **convex widening** operator (represented as $\bigtriangledown_C$) for convex polyhedra – i.e., regions formed by conjuncting affine constraints [CH78]. If both p and p' are convex, then $p \bigtriangledown_C p'$ is defined by the constraints in p which are also satisfied by p'. For example,

$$x - 1 \leq y \ \wedge \ y \leq x \ \ \bigtriangledown_C \ \ x - 2 \leq y \ \wedge \ y \leq x \ \equiv \ y \leq x$$

Intuitively, if a constraint of p is not satisfied by p' this means that the iterates are increasing in that direction. By removing that constraint we extend the iterates in the direction of growth as much as possible without violating other constraints. Since $p \bigtriangledown_C p'$ is built by simply removing constraints from p and since we cannot remove infinitely many constraints, the finiteness property is satisfied.

We demonstrate how widening can be used in the context of our event-action language and with the Presburger formula representations. Consider the following program, which consists of only one event:

```
VARIABLES
  $x, y$: positive integer
EVENTS
  $e \ : \ x > 0 \ \wedge \ x' = x - 1 \ \wedge \ y' = y + 1$
```

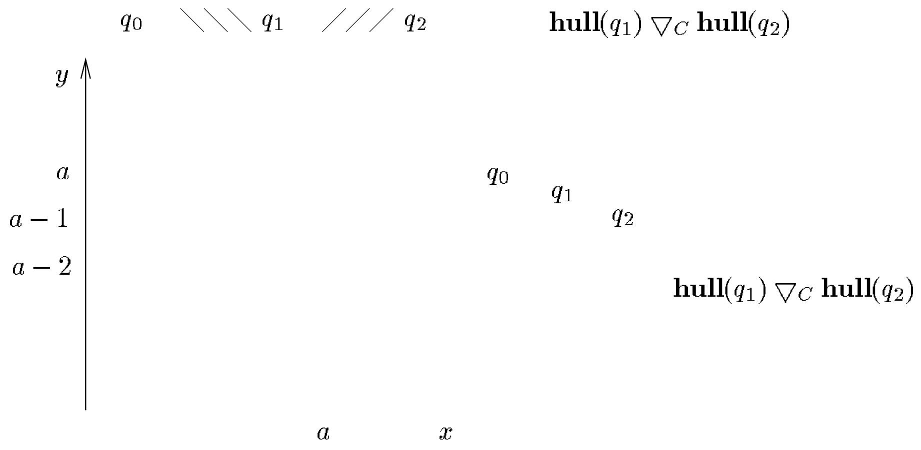


Figure 6.2: A Simple Example Demonstrating How the Widening Operator $\bigtriangledown_C$ Works

Assume that we wish to check the property AG$(y \neq a)$, where a is a positive constant. Our symbolic model checker will convert this property to $\neg($**true** EU $y = a)$, and first try to compute an exact fixpoint for **true** EU $y = a$. The sets q_0, q_1, q_2 generated by the first iterations of the exact fixpoint algorithm are (Figure 6.2)

$$\begin{aligned}
q_0 &\equiv y = a \\
q_1 &\equiv (\text{EX}\ \ y = a) \ \vee \ y = a \\
q_1 &\equiv (\exists x', y' \,.\, x > 0 \ \wedge \ x' = x - 1 \ \wedge \ y' = y + 1 \ \wedge \ y' = a) \ \vee \ y = a \\
&\equiv (y = a - 1 \ \wedge \ x \geq 1 \ \wedge \ a \geq 1) \ \vee \ y = a \\
q_2 &\equiv (\exists x', y' \,.\, x > 0 \ \wedge \ x' = x - 1 \ \wedge \ y' = y + 1 \ \wedge \\
&\quad ((y = a - 1 \ \wedge \ x \geq 1 \ \wedge \ a \geq 1) \ \vee \ y = a)) \ \vee \\
&\quad ((y = a - 1 \ \wedge \ x \geq 1 \ \wedge \ a \geq 1) \ \vee \ y = a) \\
&\equiv (y = a - 2 \ \wedge \ x \geq 2 \ \wedge \ a \geq 2) \ \vee \ (y = a - 1 \ \wedge \ x \geq 1 \ \wedge \ a \geq 1) \ \vee \\
&\quad y = a
\end{aligned}$$

At this point, we can see that the sequence will diverge. Actually, the i'th iteration of the fixpoint computation will be in the form

$$q_i \;\equiv\; q_{i-1} \;\vee\; (y = a - i \;\wedge\; x \geq i \;\wedge\; a \geq i)$$

Since for all i $q_i \not\equiv q_{i+1}$, fixpoint iterations will not converge.

Now, we demonstrate how the widening technique can be applied to this example. Assume that we apply the widening sequence with seed $s = 1$. Then the first two iterates of the widening sequence are same as the first two iterates of the exact sequence, i.e., $\hat{q}_0 \equiv q_0$ and $\hat{q}_1 \equiv q_1$. The second iterate of the exact sequence is $q_2 \equiv \hat{q}_1 \vee (\mathrm{EX}\ \hat{q}_1)$.

We want to compute the second iterate of the widening sequence $\hat{q}_2$ by applying the widening operator as $\hat{q}_2 \equiv q_1 \nabla_C q_2$, however we cannot do that since the representations computed above for q_1 and q_2 are not convex representations. To apply the convex widening operator ∇_C to the sets q_1 and q_2, we have to express them as conjunctions of affine constraints. However, our symbolic computations return the above representations for q_1 and q_2, where both sets are expressed in a disjunctive form as unions of convex regions. To convert them into convex polyhedra, we use the **convex hull** operator implemented by the Omega Library. Given a set p represented as a disjunction of convex regions, let us define **hull**(p) as the least convex region that includes every point in p (i.e., convex hull of p). If **hull**(p) $\equiv p$, then we can use **hull**(p) as a convex representation of p. If we use the **hull** operator on two disjuncts representing the set q_1 we get:

$$\begin{aligned}&\mathbf{hull}(\; y = a - 1 \;\wedge\; x \geq 1 \;\wedge\; a \geq 1 \;\vee\; y = a \;)\\ &\equiv\; a \leq x + y \;\wedge\; a - 1 \leq y \;\wedge\; y \leq a\end{aligned}$$

Similarly we compute the following convex representation for q_2 using the **hull** operator:

$$q_2 \;\equiv\; a \leq x+y \;\wedge\; a-2 \leq y \;\wedge\; y \leq a$$

We then obtain $\hat{q}_2$ by computing $q_1 \bigtriangledown_C q_2$ as:

$$\begin{aligned} q_1 \bigtriangledown_C q_2 \;&\equiv\; (a \leq x+y \;\;\wedge\;\; a-1 \leq y \;\wedge\; y \leq a) \\ &\qquad \bigtriangledown_C \; (a \leq x+y \;\wedge\; a-2 \leq y \;\wedge\; y \leq a) \\ &\equiv\; a \leq x+y \;\wedge\; y \leq a \end{aligned}$$

The iterations converge, since this formula is also generated for $\hat{q}_3$. When we negate the result, we get $x+y < a \;\vee\; a < y$. In other words, if our initialization of x and y satisfies this condition, then the invariant will indeed hold.

There are several points that should be clarified. First, note that the widening operator works on the syntax of the formula. Hence, different representations of the same formula may give different results when fed into the widening operator. Secondly, in the simple example discussed above, if we start with $s = 0$, we end up computing $\hat{q}_1$ via $q_0 \bigtriangledown_C q_1$, where

$$q_0 \equiv y = a \qquad \text{and} \qquad q_1 \equiv a \leq x+y \;\;\wedge\;\; a-1 \leq y \leq a.$$

Then the result is $q_0 \bigtriangledown_C q_1 \equiv$ **true** which is obviously a bad approximation. And indeed, in our experiments, we found that some widening sequences require higher seeds to compute good approximations. By using a higher seed, we gain more exact information about the program before making any approximations – and hence, growth directions can be predicted better. In our implementation of the widening technique, we start with setting the seed to zero ($s = 0$). If the computed approximation is too coarse (not strong enough to prove or disprove the property), we increase the seed's value, and generate a new widening

```
Input $q \equiv \bigvee_{j=1}^{m} q_j$, $r \equiv \bigvee_{i=1}^{n} r_i$  s.t.  $q_j, r_i$ are all convex polytopes, and $q \subseteq r$
Output $p \equiv q \bigtriangledown r$
Repeat
    $r' \leftarrow$ false; $mark[1..n] \leftarrow$ false; $deleted \leftarrow 0$;
    For  $1 \leq i < j \leq n$  Do
        If  $mark[i] \equiv mark[j] \equiv$ false $\wedge$ hull$(r_i \vee r_j) \equiv r_i \vee r_j$
        Then  $r' \leftarrow r' \vee$ hull$(r_i \vee r_j)$; $mark[i], mark[j] \leftarrow$ true; $deleted++$;
    For  $1 \leq i \leq n$  Do
        If  $mark[i] \equiv$ false  Then  $r' \leftarrow r' \vee r_i$;
    $r \leftarrow r'$; $n \leftarrow n - deleted$;
Until $deleted \equiv 0$
$p \leftarrow$ false; $mark[1..n] \leftarrow$ false;
For  $1 \leq i \leq n \wedge 1 \leq j \leq m$  Do
    If  $q_j \subseteq r_i$  Then  $p \leftarrow p \vee q_j \bigtriangledown_C r_i$ ; $mark[i] \leftarrow$ true;
For  $1 \leq i \leq n$  Do
    If  $mark[i] \equiv$ false  Then  $p \leftarrow p \vee r_i$;
Return p
```

Figure 6.3: Algorithm for Multi-Polyhedra Widening

sequence. In this way, successively tighter approximations can be obtained. We also allow selective bounding of the seed because there may be cases where a property cannot be proved or disproved

Another way to prevent over approximations is to avoid using the widening operator when two polytopes have different dimensions. (We did not implement this feature in our prototype analyzer.) Note that for the example given above q_0 is a line (i.e., a one-dimensional polytope) whereas q_1 is a plane (i.e., a two-dimensional polytope). By checking the dimensions of these sets we can prevent the generation of a bad approximation.

6.2.2 Multi-Polyhedra Widening

The widening operator ∇_C is sufficient if we always have convex sets. However, a program's state space is not always convex; in fact, most (exact) fixpoint computations are composed of a potentially large number of disjuncts, each defining a convex polytope. Since the widening operator ∇_C folds all arguments into a single convex region, a direct application of this method failed to work for the examples we analyzed. The reason is that on most of our examples to date, iterates of the fixpoint computations were composed of a list of disjuncts which were not representable as a single convex polytope. To accommodate this we generalized the widening technique to handle multiple polyhedra.

Assume that we have two sets q and r, where $q \subseteq r$ and q and r are represented as:

$$q \equiv q_1 \ \vee \ q_2 \ \vee \ \ldots \ \vee \ q_m \quad \text{and} \quad r \equiv r_1 \ \vee \ r_2 \ \vee \ \ldots \ \vee \ r_m \ \vee \ \ldots \ \vee \ r_n$$

where all the q_i's and r_i's are convex polytopes. In Figure 6.3 we present how our multi-polyhedra widening operator is computed with such an input. There are two phases to the algorithm: 1) The **merging phase** where the number of disjuncts in the representation of r is reduced (lines 1 to 9). 2) The **widening phase** where the convex widening operator is used on matching convex pairs of regions from q and r (lines 10-14). Note that these phases are generalizations of the convex hull computation followed by the widening operation for the simple example discussed in the previous section.

The until loop (lines 1 to 9) reduces the number of disjuncts in r by merging adjacent convex polytopes. Two convex polytopes are replaced by their hull if their hull is equal to their union. We continue doing this until no more reduction is possible. Note that the resulting representation of r depends on the order of execution of the first for loop (lines 3 to 5). Traversing the list of polytopes in different orders may lead to different results.

After reducing the number of disjuncts in r, we look for convex polytopes (i.e., disjuncts) in q and r such that $q_j \subseteq r_i$. When we find such a pair, we use convex widening operator to compute $q_j \bigtriangledown_C r_i$. This new convex polytope is appended to the output Presburger set p. Finally we copy the disjuncts from r which are not widened to p, and return p as the result. Note that the order of the for loop in lines 11 to 12 can again affect the result.

The final result may include too many disjuncts. In the algorithm given in Figure 6.3, if after the merging phase q is represented by m disjuncts and r is represented by n disjuncts, in the worst case the result $(q \bigtriangledown r)$ can contain $m \times n$ disjuncts. To ensure convergence, we also assign an upper bound to the number of disjoint convex regions we wish to represent. When we reach this bound we force-merge disjoint regions by replacing them with their convex hull – even if that loses precision (which is valid since we are computing upper bounds). In the experiments we conducted so far we never had to resort to this technique.

As we stated above the algorithm in Figure 6.3 is separated in two phases: 1) merging (lines 1 to 9) and 2) widening (lines 10 to 14). Both phases are crucial for the performance of the algorithm, and in both phases two regions are matched with each other either for merging or widening.

Note that in the algorithm given in Figure 6.3 we can pair up regions which may not be "related", i.e., two regions which do not indicate any type of repetitive (or iterative) behavior. The intuition in the widening technique is that the regions which are matched demonstrate a behavior that will be repeated. Hence, the guess we make based on their behavior will be valid for the following iterations of the fixpoint computations. In the algorithm given in Figure 6.3, if there are more than one candidates for matching, then one is chosen arbitrarily. As we discussed above the traversal order of the loops in lines 11 to 12 and lines 3 to 5 can affect the decision of which regions are matched. It would be interesting to develop heuristics which take into account information from the program that is being analyzed in making these decisions. Actually, partitioning techniques (section 4.2.3) do exactly this as we discuss below.

Also we would like to mention that we did not discuss the stride constraints above. Hence our discussions are restricted to sets expressible as a finite union of convex polyhedra. This does not cover all possible Presburger formula representations. The technique described above was sufficient for analyzing the examples discussed in this dissertation. It would be valuable to extend the above technique to more general representations by considering stride constraints.

6.2.3 Widening on Partitioned Transition Systems

As explained in Section 4.2.3, our symbolic model checker implements various partitioning heuristics. In the discussion above, we did not consider the fixpoint computations on the partitioned transition systems. As we present below, the generalization of the widening technique to the partitioned transition systems is straightforward.

Assume that the two sets that we want to apply the widening operator are partitioned as

$$q \equiv \bigvee_{i=1}^{p} q \wedge S_i \quad \text{and} \quad r \equiv \bigvee_{i=1}^{p} r \wedge S_i$$

using the partitioning $P = \{S_1, S_2, \ldots, S_p\}$. We apply the widening operator to the partitioned representations by applying it individually for each partition class:

$$q \bigtriangledown r \equiv \bigvee_{i=1}^{p} (q \wedge S_i) \bigtriangledown (r \wedge S_i)$$

as discussed in [Hal93].

Now we can explain how partitioning techniques could improve the accuracy of the conservative approximation techniques as we mentioned in section 4.2.3. As discussed above, in the widening technique the regions from consecutive fixpoint iterations are matched for merging and widening. If we use partitioning, two regions could be matched only if they are from the same partition class, i.e., the regions from $q \wedge S_i$ and $r \wedge S_j$ cannot be matched unless $i = j$. Hence, we can consider partitioning techniques as heuristics that synthesize information from the structure of the input program to restrict the set of regions that could be matched for merging and widening. If the partitioning used is able to group the parts of the program with similar characteristics and segregate the ones which are different, then it would improve the accuracy of the widening technique.

6.3 Computing Upper Bounds with Composite Representations

We compute upper bounds using composite representations by generalizing the widening technique over multiple types. Assume that q_1^t and q_2^t are two symbolic sets formed over a single type $t \in T$, and that $\bigtriangledown^t$ is a type-specific widening operator such that:

$$q_1^t \cup q_2^t \subseteq q_1^t \bigtriangledown^t q_2^t$$

i.e., $q_1^t \bigtriangledown^t q_2^t$ is an upper bound for the union computation.

Obviously there are many choices for operators which majorize binary union. However, if $\bigtriangledown^t$ is to be useful, it should – in many settings – be able to guess the direction of growth in the fixpoint iterates, and to extend the successive iterates in these directions. This will accelerate the fixpoint computation by generating a majorizing sequence to the exact fixpoint iterates. In the previous section we showed how this is done for integer domains, over multiple convex regions.

Here, we extend the widening idea to composite models. Assume that we have two composite representations

$$p \equiv \bigvee_{i=1}^{n_p} p_i \equiv \bigvee_{i=1}^{n_p} \bigwedge_{t \in T} p_i^t \quad \text{and} \quad q \equiv \bigvee_{i=1}^{n_q} q_i \equiv \bigvee_{i=1}^{n_q} \bigwedge_{t \in T} q_i^t$$

such that $p \subseteq q$. Let p' denote the following subset of disjuncts forming p: For each $p_i \in p'$, there exists some q_j such that $p_i \subseteq q_j$. Likewise, let q' denote the set of disjuncts in q for which $q_j \in q'$ implies that there is a p_i such that $p_i \subseteq q_j$. Then we define $p \bigtriangledown q$ as

$$p \bigtriangledown q \equiv \bigvee_{p_i \notin p'} p_i \;\vee\; \bigvee_{q_i \notin q'} q_i \;\vee\; \bigvee_{p_i \subseteq q_j} \bigwedge_{t \in T} (p_i^t \bigtriangledown^t q_j^t)$$

Using this operator we can generate a majorizing sequence for our least fixpoint computations for EU and AU operators as discussed above.

6.4 Computing Lower Bounds

Recall that each iteration of an exact fixpoint computation will yield a lower a bound for f_1 EU f_2 and f_1 AU f_2. So to obtain a lower approximation for the purposes of analysis, we need only stop after a finite number of iterations; in this manner we are guaranteed to have a conservative approximation. Of course the question is: when do we stop?

Our verifier uses the following rules: if it is handling the outermost formula, then after each iteration it checks whether the initial states are included in the current lower bound. If so, it stops, since the property is proved; if not, it keeps going. Obviously, there will be cases where this method fails to converge, and if this happens the tool will not be able to prove or disprove the property. However, the user is able to interact with the analyzer, and periodically monitor its progress; thus, the user can optionally "pull the plug" on waiting for a response.

If the fixpoint we are computing is a subformula of another computation, the analyzer sets a user specified time limit to stop generating an approximation – after which it is used in the next-higher formula. But if the analyzer is unable to prove or disprove the outermost formula, the user may optionally return and improve the lower bound by continuing the fixpoint sequence.

6.5 Experiments

In this section we present some example systems we analyzed using our conservative approximation techniques.

6.5.1 Approximate Analysis of the Unbounded Queue

We analyzed the running example given in Figure 4.3 and its requirement AG($p \geq c$). Using the negation-labeling algorithm, the requirement is rendered as $(\neg(\mathrm{EF}(p \geq c)^{+})^{+})^{-}$. The temporal operator EF is marked with "+" which means that we need an upper bound for the set of states violating the requirement. The symbolic model checker computes the upper bound using the multi-polyhedra widening technique, and it converges after 2 iterations. The

PROGRAM The Ticket Algorithm

VARIABLES

a, b, t, s: integer

$pc_1 : \{T_1, W_1, C_1\}$, $pc_2 : \{T_2, W_2, C_2\}$

INITIAL CONDITION

$t = s \ \wedge \ pc_1 = T_1 \ \wedge \ pc_2 = T_2$

EVENTS

$$\begin{array}{lcl}
e_{T_1} & : & pc_1 = T_1 \ \wedge \ pc_1' = W_1 \ \wedge \ a' = t \ \wedge \ t' = t+1 \\
e_{W_1} & : & pc_1 = W_1 \ \wedge \ a \leq s \ \wedge \ pc_1' = C_1 \\
e_{C_1} & : & pc_1 = C_1 \ \wedge \ pc_1' = T_1 \ \wedge \ s' = s+1 \\
e_{T_2} & : & pc_2 = T_2 \ \wedge \ pc_2' = W_2 \ \wedge \ b' = t \ \wedge \ t' = t+1 \\
e_{W_2} & : & pc_2 = W_2 \ \wedge \ b \leq s \ \wedge \ pc_2' = C_2 \\
e_{C_2} & : & pc_2 = C_2 \ \wedge \ pc_2' = T_2 \ \wedge \ s' = s+1
\end{array}$$

Figure 6.4: The Ticket Mutual-Exclusion Algorithm

result is the set $\hat{q}$ which is partitioned as

$$\begin{array}{lcl}
\hat{q}_{Idle} & \equiv & pcs = Idle \ \wedge \ p < c + i_1 + i_2 \\
\hat{q}_{Send} & \equiv & pcs = Send \ \wedge \ p < c + i_1 + i_2
\end{array}$$

However, since we are actually computing $\neg \mathrm{EF}(p \geq c)$, the model checker computes $\neg \hat{q}$, which gives a lower approximation for the states which satisfy the property $\mathrm{AG}(p \geq c)$. Recall that the set of initial states of the running example is $I \equiv p = 0 \ \wedge \ c = 0 \ \wedge \ i_1 = 0 \ \wedge \ i_2 = 0$, and observe that $I \subseteq \neg \hat{q}$. Hence, the model checker reports that the property is proved (with a CPU time of 1.74 seconds).

6.5.2 The Ticket Algorithm

In Figure 6.4, we present the **ticket algorithm** [And91]. In particular, note its similarity to the bakery algorithm. The difference is that the value of the next available ticket is stored in the global variable t, while another global variable s holds the highest ticket value served thus far. New tickets are obtained by executing a fetch-and-add on t. A customer can enter the critical section when the last-used ticket s catches up to its local ticket number.

Again, the mutual-exclusion property is (T1) $\neg \mathrm{EF}(pc_1 = C_1 \ \wedge \ pc_2 = C_2)$, which asserts that two processes can not be in the critical section at the same time. We first tried to check the mutual-exclusion property of the ticket algorithm with the exact analyzer using the control-point partitioning. (We eliminated the control variables pc_1 and pc_2 from the Presburger formula representations using this partitioning.) To compute the exact fixpoint, the analyzer started symbolically enumerating ways that both a and b could be less than s. Since s and t are unbounded, this computation does not terminate. Next, we applied our conservative approximation technique. Using the negation-labeling algorithm, the mutual-exclusion property of the ticket algorithm is rendered as $(\neg(\mathrm{EF}(pc_1 = C_1 \ \wedge \ pc_2 = C_2)^{+})^{+})^{-}$. The temporal operator EF is marked with "+" which means that we need an upper bound for the set of states violating mutual-exclusion. The symbolic model checker computes the upper bound using the multi-polyhedra widening technique, and it converges after 9 iterations. The result is the set $\hat{q}$ which is partitioned as

$$\begin{aligned}
\hat{q}_{(T_1,T_2)} &\equiv pc_1 = T_1 \ \wedge \ pc_2 = T_2 \ \wedge \ t < s \\
\hat{q}_{(T_1,W_2)} &\equiv pc_1 = T_1 \ \wedge \ pc_2 = W_2 \ \wedge \ t \leq s \\
\hat{q}_{(T_1,C_2)} &\equiv pc_1 = T_1 \ \wedge \ pc_2 = C_2 \ \wedge \ t \leq s \\
\hat{q}_{(W_1,T_2)} &\equiv pc_1 = W_1 \ \wedge \ pc_2 = T_2 \ \wedge \ t \leq s \\
\hat{q}_{(C_1,T_2)} &\equiv pc_1 = C_1 \ \wedge \ pc_2 = T_2 \ \wedge \ t \leq s \\
\hat{q}_{(C_1,C_2)} &\equiv pc_1 = C_1 \ \wedge \ pc_2 = C_2 \ \wedge \ \mathbf{true} \\
\hat{q}_{(W_1,C_2)} &\equiv pc_1 = W_1 \ \wedge \ pc_2 = C_2 \ \wedge \ (a \leq s \ \vee \ t \leq s+1) \\
\hat{q}_{(C_1,W_2)} &\equiv pc_1 = C_1 \ \wedge \ pc_2 = W_2 \ \wedge \ (b \leq s \ \vee \ t \leq s+1) \\
\hat{q}_{(W_1,W_2)} &\equiv pc_1 = W_1 \ \wedge \ pc_2 = W_2 \ \wedge \ (b \leq s \ \wedge \ a \leq s \ \vee \ t \leq s+1 \ \wedge \\
&\qquad b \leq s \ \vee \ t \leq s+1 \ \wedge \ a \leq s)
\end{aligned}$$

However, since we are actually computing $\neg\mathrm{EF}(pc_1 = C_1 \ \wedge \ pc_2 = C_2)$, the model checker computes $\neg\hat{q}$, which gives a lower approximation for the states which respect mutual-exclusion. Recall that the set of initial states of the ticket algorithm is $I \equiv pc_1 = T_1 \ \wedge \ pc_2 = T_2 \ \wedge \ t = s$, and observe that $I \subseteq \neg\hat{q}$. Hence, the model checker reports that the property is proved (with a CPU time of 7.32 seconds).

We also wish to prove starvation-freedom. Negation-labeling converts process 1's relevant formula to:

$$\text{(T2)} \quad (\neg(\mathrm{EF}((pc_1 = W_1)^+ \ \wedge \ (\neg(\mathrm{AF}(pc_1 = C_1)^-)^-)^+)^+)^+)^-.$$

Note that because of the double negation, the inner fixpoint (AF) is marked with "−" (i.e., a lower bound), whereas the outer fixpoint (EF) is marked with "+." The checker computes the AF property exactly, in 5 fixpoint iterations; hence the lower bound turns out to be exact. Then it computes an upper bound for the EF property in 7 iterations, by using the widening technique. After the lower bound for the whole formula is computed, the model checker reports that all the initial states do indeed satisfy the liveness property (for a total CPU time of 27.03 seconds).

6.6 Related Work

Recently, there has been other work that used widening operator for computing fixpoints on polyhedral representations. In [Hal93], Halbwachs investigates application of the widening operator $\bigtriangledown_C$ to delay analysis in synchronous programs. In [HH95] and [HRP94], conservative approximation techniques are used for analyzing temporal properties of linear hybrid systems. Fixpoint iterations are represented as unions of finite convex regions on real domains and the widening operator $\bigtriangledown_C$ is used to compute upper approximations. In [HH95], Henzinger and Ho, use both forward and backward fixpoint computations and iteratively refined conservative approximations. They present an extrapolation operator which gives better approximations (i.e., tighter bounds) then the widening operator $\bigtriangledown_C$ for the example analyzed in that paper.

In [Hal93], [HH95], and [HRP94], approximation operators are used on convex regions. Our multi-polyhedra widening operator generalizes this approach. Also by combining upper approximations with lower approximations we are able to analyze any CTL property conservatively. Actually, our methods easily generalize to full μ-calculus.

In [DP98], the properties of the ticket algorithm (and some other systems we analyzed) are verified using a model checker based on constraint logic programming. The fixpoints reported in [DP98] are equivalent to our results. The symbolic representation used in [DP98] is constraints over reals as in [AHH96].

Our conservative approximation framework relies on the same principles used in [KDG95] and later in [DGG97]. While we approximate the fixpoint iterations, Kelb *et al.* use the conservative approximation framework to generate upper and lower bounds for individual next state computations EX and AX. This could be done by producing an over approximation R^+ and under approximation R^- of the transition relation R. Note that, if we use R^- instead of R and compute EX p, we will get a lower approximation (EX $p)^-$. If we use R^+ instead, we will get an upper approximation (EX $p)^+$. Using this approach while computing the fixpoint iterates, we can generate lower and upper bounds for fixpoint iterations. While our approach uses the exact transition relation and approximates the fixpoints on that transition relation, Kelb *et al.* use approximate transition relations and compute exact fixpoints on those transition relations. Of course, these two approaches could easily be combined to generate a wider variety of heuristics for approximating fixpoints.

Chapter 7

Reachability Analysis

In the previous chapter we discussed state-based approximations for general model-checking decision problems, where we have to verify a general CTL formula over infinite state spaces. However, we also make use of two special-purpose techniques, which fall into the generic category of reachability analysis. One variant of this is **state-based**, and it computes an upper bound for the program's reachable state-space. The other method is **transition-based**, it produces approximations for the transitive closure of the transition system itself. Both of these techniques have proved most successful when used in conjunction with our symbolic model-checker – both for exact and conservative analysis.

Results discussed in this chapter were initially reported in [BGP96] and [BGP98].

7.1 State-Based Reachability Analysis

The fixpoint algorithms described thus far are **backward** techniques, in that they start with a property f, and then use EX to determine which states can reach f. The last step is to determine whether the initial condition I is included in the derived set. Alternatively, it may be useful to start with I, compute an upper approximation RS^+ to the reachable state-space RS, and then use RS^+ to help in the model-checking process. We can accomplish practically this by altering the symbolic model checker to restrict its computations to states in RS^+.

To generate the upper bound RS^+, we define a previous state operator. The existential previous state operator EP is defined as follows, EP q is true in a state if there exists a previous state of that state where q is true. Hence, given a set of states, the functional EP : $2^S \rightarrow 2^S$ returns the states reachable from this set in one step; i.e., EP $q \overset{\text{def}}{=} \{\, s \mid \exists s' \; : \; s' \in q \;\wedge\; (s', s) \in R\}$. In other words EP $q \equiv R\ q$. Note that EP is the forward analogue of the EX operator. We claim that the (exact) reachable state space of a program is the least fixpoint of the functional $\lambda y \,.\, I \;\vee\; \mathrm{EP}\ y$, i.e.,

$$RS \equiv \mu y \,.\, I \;\vee\; \mathrm{EP}\ y$$

and RS can be computed using the techniques we previously developed for EU and AU. Moreover, we can use the widening method to compute an upper bound for RS as well.

After computing RS^+, we restrict the result of every operation in the model checker to RS^+. For example, when a fixpoint iterate q_i is produced, it is replaced by $q_i \wedge RS^+$. Most importantly, we also can use this technique when we compute approximate fixpoints, as defined above.

7.2 Transition-Based Reachability Analysis

We also developed a transition-based reachability analysis technique, in which multiple execution paths get collapsed into single relations. We achieve this by adding new events to the original program; these events summarize repeated executions of self-loops, while preserving the underlying state-space. These transformations make the fixpoint computations for reachability properties converge faster without changing their truth sets.

Given a program $P = (V, I, E)$, we define a set of programs $C(P)$ as follows:

$$C(P) = \{\ (V, I, E') \mid \bigvee_{e \in E} R_e \subseteq \bigvee_{e \in E'} R_e \subseteq (\bigvee_{e \in E} R_e)^* \equiv R^*\ \}$$

where R^* denotes the **reflexive, transitive closure** of the transition relation R.

The programs in $C(P)$ have the following property:

Theorem 7.1 *If $P' \in C(P)$, then (1) it includes at least all the single step executions in P, and (2) for every finite execution path in P' there exists a finite execution path in P such that the first and the last states of the two execution paths are the same.*

Using definition of $C(P)$ and Theorem 7.1 we can show the following:

Theorem 7.2 *If $P' \in C(P)$ then*

1. *Truth sets of basic properties $f \in BP$ for P and P' are identical.*
2. *The set of reachable states RS for P and P' are identical.*
3. *If the truth set of the formula f for P' and P are the same then the truth sets of formulas EFf and AGf are the same for P' and P.*
4. *If the truth sets of the formulas f and g for P' are supersets of the truth sets of the corresponding formulas for P, then the truth sets of formulas $f\ EU\ g$ and EXf for P' give upper bounds for the truth sets of the corresponding formulas for P.*

5. *If truth sets of the formulas f and g for P' are subsets of the truth sets of the corresponding formulas for P, then the truth sets of formulas $f\ AU\ g$ and AXf for P' give lower bounds for the truth sets of the corresponding formulas for P.*

Theoretically, if we could compute R^* exactly, we would have a single relation which would summarize all reachable executions of the original program. That is, R^* would take any initial program state s as input, and instantly produce all states reachable from s. Likewise, $(R^*)^{-1}$ could handle any reverse reachable-state query. In other words, we could compute EFf and RS in a single step. However, in most instances, computing transitive closures of Presburger relations is not practical. Actually, in general, constructing transitive closure of a Presburger relation is not computable.

However, akin to our state-based techniques, there are approximation techniques for obtaining upper and lower bounds for R^* [KPRS94]. In turn, these can be used to compute our desired lower and upper bounds for RS and EFf. However, there's a fundamental trade-off involved in this: On one hand, if R is a simple convex relation in a few dimensions, it may be easy to compute R^* exactly, or a reasonable approximation thereof. Yet for such simple programs, we rarely need automated analysis. On the other hand, if a transition relation R is relatively complex (i.e., with many variables and concavities), the only tractable upper bound for R^* may be a trivial one – e.g., $S \times S$. Similarly, the computed lower bound may just turn out to be the identity relation.

Hence, we use these approximation techniques on selected parts of R – and never on the whole program at once. Specifically, we compute transitive closures on selected self-loops, which are stable with respect to our partition classes.

We apply this strategy automatically as follows. Given a program $P = (V, I, E)$, with a state space partitioning $P = \{S_1, S_2, \dots S_p\}$, we compute the maximal set of self-loops, where each loop l_{ei} is nonempty, and satisfies the following property:

$$l_{ei} = \{\ (s, s')\ |\ (s, s') \in R_e \wedge s \in S_i \wedge s' \in S_i\ \}$$

In other words, the loop l_{ei} represents stability – when it starts on a state in partition class S_i, it returns to the partition class S_i.

Next, for all such loops, we attempt to compute their reflexive, transitive closures l^*_{ei}. If obtaining the exact l^*_{ei} proves impossible, our analyzer gets a lower-bound approximation. (Note that this is often sufficient.) Then we generate a new program $P' = (V, I, E')$, by adding all events l^*_{ei} (or their lower bounds) to E, and deleting redundant transitions. Since

$$\bigvee_{e \in E} R^*_e \subseteq (\bigvee_{e \in E} R_e)^* \quad \text{and} \quad \bigvee_{e \in E'} R_e \subseteq \bigvee_{e \in E} R^*_e$$

we have

$$\bigvee_{e \in E} R_e \subseteq \bigvee_{e \in E'} R_e \subseteq (\bigvee_{e \in E} R_e)^*.$$

Therefore, $P' \in C(P)$ and satisfies the conditions in Theorem 7.1 and Theorem 7.2.

If we apply this technique to our unbounded buffer example from Figure 4.3, along with the event-domain partitioning, we get the program shown in Figure 7.1 where the three new events $l^*_{R,1}$, $l^*_{R,2}$ and l^*_S are self-loop closures which are computed as explained above.

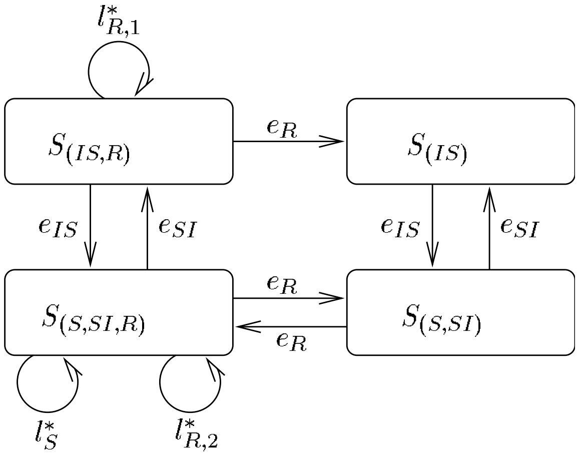


Figure 7.1: Unbounded Queue with Event-Domain Partitioning and Loop Closures

$$\begin{aligned}
l^*_{R,1} \equiv\ & (pc = \mathit{Idle} \wedge pc' = \mathit{Idle}) \wedge (c' = q_1 + q_2 + c - q'_1 - q'_2) \wedge (0 \leq q'_1 \leq q_1) \wedge \\
& (0 \leq q'_2 \leq q_2) \wedge ((1 \leq q'_1) \vee (1 \leq q'_2)) \\
l^*_{R,2} \equiv\ & (pc = \mathit{Send} \wedge pc' = \mathit{Send}) \wedge (c' = q_1 + q_2 + c - q'_1 - q'_2) \wedge (0 \leq q'_1 \leq q_1) \wedge \\
& (0 \leq q'_2 \leq q_2) \wedge ((1 \leq q'_1) \vee (1 \leq q'_2)) \\
l^*_S \equiv\ & (pc = \mathit{Send} \wedge pc' = \mathit{Send}) \wedge (p' = p - q_1 - q_2 + q'_1 + q'_2) \wedge (0 \leq q_1 \leq q'_1) \wedge \\
& (0 \leq q_2 \leq q'_2) \wedge ((1 \leq q_1) \vee (1 \leq q_2))
\end{aligned}$$

7.3 Experiments

In this section we discuss some examples we analyzed using the reachability analysis.

7.3.1 The Ticket Algorithm Revisited

We analyzed the ticket algorithm given in Figure 6.4 using reachability analysis. The analyzer computed the following upper bound for the set of reachable states RS^+ of the ticket algorithm (which turns out to be exact):

$$\begin{aligned}
RS^+_{(T_1,T_2)} &\equiv pc_1 = T_1 \ \wedge \ pc_2 = T_2 \ \wedge \ t = s \\
RS^+_{(T_1,W_2)} &\equiv pc_1 = T_1 \ \wedge \ pc_2 = W_2 \ \wedge \ t = s+1 \ \wedge \ b = s \\
RS^+_{(T_1,C_2)} &\equiv pc_1 = T_1 \ \wedge \ pc_2 = C_2 \ \wedge \ t = s+1 \ \wedge \ b = s \\
RS^+_{(W_1,T_2)} &\equiv pc_1 = W_1 \ \wedge \ pc_2 = T_2 \ \wedge \ t = s+1 \ \wedge \ a = s \\
RS^+_{(C_1,T_2)} &\equiv pc_1 = C_1 \ \wedge \ pc_2 = T_2 \ \wedge \ t = s+1 \ \wedge \ a = s \\
RS^+_{(C_1,C_2)} &\equiv pc_1 = C_1 \ \wedge \ pc_2 = C_2 \ \wedge \ \mathbf{false} \\
RS^+_{(W_1,C_2)} &\equiv pc_1 = W_1 \ \wedge \ pc_2 = C_2 \ \wedge \ t = s+2 \ \wedge \ a = s+1 \ \wedge \ b = s \\
RS^+_{(C_1,W_2)} &\equiv pc_1 = C_1 \ \wedge \ pc_2 = W_2 \ \wedge \ b = s+1 \ \wedge \ a = s \ \wedge \ t = s+2 \\
RS^+_{(W_1,W_2)} &\equiv pc_1 = W_1 \ \wedge \ pc_2 = W_2 \ \wedge \ b+a = 2s+1 \ \wedge \ t = 2+s \ \wedge \ b-1 \leq s \leq b
\end{aligned}$$

When the state space of the ticket algorithm was restricted to this set the model checker was able to prove both the mutual-exclusion and the starvation-freedom properties of the ticket algorithm without using the conservative approximation techniques.

7.3.2 A Bounded Buffer

In Figure 7.2, we present a bounded-buffer producer-consumer problem adapted from an example in [Sha93]. This program implements an instance of the problem with two producers and two consumers. We wish to prove that (BB) $\mathrm{AG}(0 \leq p_1 + p_2 - (c_1 + c_2) \leq s)$ holds, i.e., that the bounded buffer properties are satisfied. When we translate this into existential form, we get $\neg\mathrm{EF}(\neg(0 \leq p_1 + p_2 - (c_1 + c_2) \leq s))$. The exact model checker diverged when we fed it the producer-consumer program with this property. But when we tried

PROGRAM A Bounded Buffer

CONSTANTS

s: integer, $s \geq 1$

VARIABLES

a, p_1, p_2, c_1, c_2: positive integer

INITIAL CONDITION

$p_1 = p_2 = c_1 = c_2 = 0 \wedge a = s$

EVENTS

$e_{P_1} \; : \; a > 0 \wedge p_1' = p_1 + 1 \wedge a' = a - 1$

$e_{P_2} \; : \; a > 0 \wedge p_2' = p_2 + 1 \wedge a' = a - 1$

$e_{C_1} \; : \; a < s \wedge c_1' = c_1 + 1 \wedge a' = a + 1$

$e_{C_2} \; : \; a < s \wedge c_2' = c_2 + 1 \wedge a' = a + 1$

Figure 7.2: A Bounded Buffer

the widening technique, the model checker successfully verified the property. We were also able to prove the same property using reachability analysis in conjunction with exact fixpoint computations.

7.3.3 A Circular Queue

The circular queue program (Figure 7.3) consists of one producer component and one consumer component. The producer and consumer execute concurrently. Figure 7.4 shows some possible configurations of the queue during the execution of the program. Crossed locations indicate occupied cells. Note that, (a) shows the initial configuration and (e) shows a configuration where the queue is full.

Variables h and t represent the **head** and the **tail** of the queue, respectively; h points to the location of the item that will be consumed next (if the queue is not empty), and t points to the location where the next produced item will

PROGRAM Circular Queue

CONSTANTS

s: integer, $s \geq 1$

VARIABLES

h, t, p, c: integer

INITIAL CONDITION

$h = t = p = c = 0$

EVENTS

$e_P \;:\; (t = s \rightarrow h > 0) \wedge (t < s \rightarrow t + 1 \neq h) \wedge$
$\quad ((t = s \wedge t' = 0) \vee (t < s \wedge t' = t + 1)) \wedge p' = p + 1$

$e_C \;:\; (t \neq h) \wedge ((h = s \wedge h' = 0) \vee (h < s \wedge h' = h + 1)) \wedge c' = c + 1$

Figure 7.3: A Circular Queue

be placed (if the queue is not full). Constant s denotes the size of the queue. Although there are $s+1$ locations in the queue, maximum number of items that can be stored is s. Variables p and c show the number of items produced and consumed, respectively.

There are several interesting properties we may want to prove about the circular queue program:

$$\text{(CQ1)} \quad \text{AG}(h \leq s \;\wedge\; t \leq s)$$

$$\text{(CQ2)} \quad \text{AG}(t \geq h \rightarrow p - c = t - h)$$

$$\text{(CQ3)} \quad \text{AG}(t < h \rightarrow p - c = s - (h - t) + 1)$$

$$\text{(CQ4)} \quad \text{AG}(0 \leq p - c \leq s)$$

When we decompose the events of the circular queue program to DNF-decomposed events, we get a program with 6 events as shown in Figure 7.5. In this program the producer and the consumer components are represented by

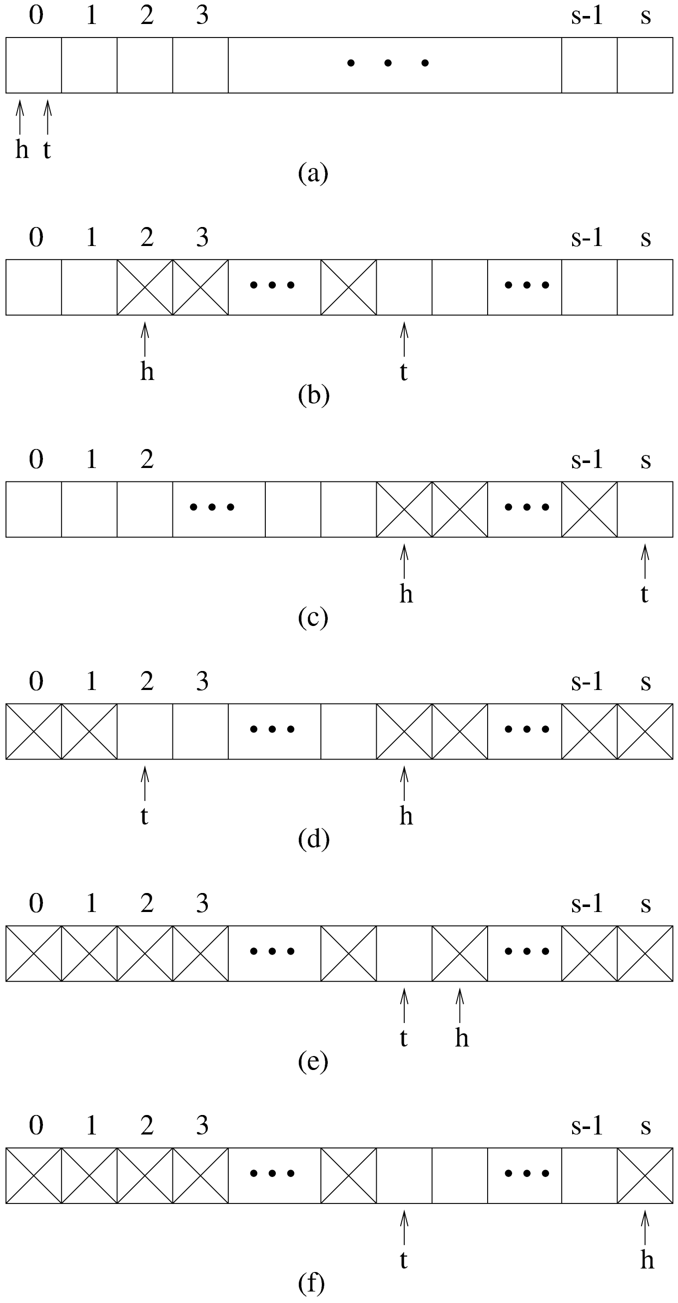


Figure 7.4: Possible Configurations of the Circular Queue

EVENTS

e_{P_1} : $t < s \wedge h < t + 1 \wedge t' = t + 1 \wedge p' = p + 1$

e_{P_2} : $t < s \wedge h > t + 1 \wedge t' = t + 1 \wedge p' = p + 1$

e_{P_3} : $t = s \wedge h > 0 \wedge t = 0 \wedge p' = p + 1$

e_{C_1} : $h < t \wedge h' = h + 1 \wedge c' = c + 1$

e_{C_2} : $h < s \wedge h > t \wedge h' = h + 1 \wedge c' = c + 1$

e_{C_3} : $h = s \wedge h > t \wedge h' = 0 \wedge c' = c + 1$

Figure 7.5: Circular Queue Program with DNF Event Decomposition

three events each. We observe that at any time only one producer event and one consumer event is enabled. The enabling conditions for events e_{P_1}, e_{P_2} and e_{P_3} are mutually exclusive (i.e., when one of them is enabled the rest are disabled). This is also true for the three consumer events. This means that there is some inherent sequential behavior in the producer and consumer components of the program. Based on this new event set, the event-domain partitioning algorithm generates a partitioning with 10 classes which is significantly less than the worst case, and for each partition class there are at most two events enabled.

After computing the event-domain partitioning based on DNF decomposition, we computed self-loop closures of the program. We first transform the program via a reachable-transition computation. After this transformation, the symbolic model checker in Figure 4.7 successfully verified all the safety properties of the circular queue on the transformed program. Hence, we can conclude that the original program satisfied all of the safety properties.

7.3.4 Summary of the Experiments on Concurrent Algorithms

A summary of our experiments is shown in Tables 7.1, 7.2 and 7.3. We have used five programs and several temporal properties to test our analyzer. The properties are listed in Table 7.1. The results of the tests are shown in Tables 7.2 and 7.3. In Tables 7.2 and Table 7.3 DNF denotes DNF partitioning of events (Section 4.2.3), SP denotes state partitioning (Section 4.2.3), TR denotes transition-based reachability analysis (Section 7.2), SR denotes state-based reachability analysis (Section 7.1), and $\uparrow$ denotes that the model checker diverged.

The approximate fixpoint computations were generated using the multi-polyhedra widening technique presented in Section 6.2.2. For the approximation sequences, we used a starting seed of $s = 0$, and increased it incrementally if the computed approximation was not strong enough to prove the property. During the generation of approximation sequences with higher seed values we started the computation from scratch, i.e., we did not use any information from the previous approximation sequences. The seed values reported in the Tables 7.2 and 7.3 are the ones which generated approximations that were good enough for verifying the properties. The execution times reported for the approximate fixpoint computations are the overall computation times, including the time for computing the approximations which failed to verify the property. Hence, the properties that were verified with higher seeds (such as CQ2 and CQ4) take significantly longer times to compute then the ones which were verified with lower seeds.

Program	Property	Label
Unbounded Buffer	$\mathrm{AG}(p = c + i_1 + i_2)$	UB1
Unbounded Buffer	$\mathrm{AG}((pc_S = \mathit{Idle} \ \wedge \ i_1 + i_2 = i) \rightarrow \mathrm{AX}(i_1 + i_2 \leq i))$	UB2
Unbounded Buffer	$\mathrm{AG}(p \geq c)$	UB3
Bakery Algorithm	$\mathrm{AG}(\neg(pc_1 = C_1 \ \wedge \ pc_2 = C_2))$	B1
Bakery Algorithm	$\mathrm{AG}(pc_1 = W_1 \rightarrow \mathrm{AF}(pc_1 = C_1))$	B2
Ticket Algorithm	$\mathrm{AG}(\neg(pc_1 = C_1 \ \wedge \ pc_2 = C_2))$	T1
Ticket Algorithm	$\mathrm{AG}(pc_1 = W_1 \rightarrow \mathrm{AF}(pc_1 = C_1))$	T2
Bounded Buffer	$\mathrm{AG}(0 \leq p - (c_1 + c_2) \leq s)$	BB
Circular Queue	$\mathrm{AG}(h \leq s \ \wedge \ t \leq s)$	CQ1
Circular Queue	$\mathrm{AG}(t \geq h \rightarrow p - c = t - h)$	CQ2
Circular Queue	$\mathrm{AG}(t < h \rightarrow p - c = s - (h - t) + 1)$	CQ3
Circular Queue	$\mathrm{AG}(0 \leq p - c \leq s)$	CQ4

Table 7.1: List of the Properties Used in the Experiments

Problem Instance	DNF	SP	TR	SR	Fixpoint Computations	Widening Seed	Execution Time
UB1					Exact		0.73 sec.
UB2					Exact		0.67 sec.
UB3					Exact		↑
UB3					Approximate	0	1.37 sec.
UB3			√		Exact	0	4.18 sec.
B1		Control Point			Exact		2.85 sec.
B1		Control Point			Approximate	0	3.21 sec.
B2		Control Point			Exact		7.64 sec.
B2		Control Point			Approximate	0	4.72 sec.
T1		Control Point			Exact		↑
T1		Control Point		√	Exact		1.44 sec.
T1		Control Point			Approximate	0	7.32 sec.
T2		Control Point			Exact		↑
T2		Control Point		√	Exact		4.34 sec.
T2		Control Point			Approximate	0	27.03 sec.

Table 7.2: Summary of the Experiments on Concurrent Systems

Problem Instance	DNF	SP	TR	SR	Fixpoint Computations	Widening Seed	Execution Time
BB					Exact		↑
BB					Approximate	0	5.34 sec.
BB			√		Exact	0	4.25 sec.
BB		Event-Domain	√		Exact	0	37.47 sec.
CQ1					Exact		0.54 sec.
CQ1					Approximate	0	0.6 sec.
CQ1	√	Event-Domain	√		Exact		11.15 sec.
CQ2					Exact		↑
CQ2	√	Event-Domain			Exact		↑
CQ2	√	Event-Domain			Approximate	4	420.84 sec.
CQ2	√	Event-Domain	√		Exact		18.91 sec.
CQ3					Exact		↑
CQ3	√	Event-Domain			Approximate	0	37.06 sec.
CQ3	√	Event-Domain	√		Exact		25.53 sec.
CQ4					Exact		↑
CQ4	√	Event-Domain			Approximate	3	231.94 sec.
CQ4	√	Event-Domain		√	Exact		↑
CQ4	√	Event-Domain	√		Exact		119.44 sec.

Table 7.3: Summary of the Experiments on Concurrent Systems (Continued)

7.4 Related Work

The idea of computing an approximation to the set of reachable states by a forward fixpoint computation, and then using this result to prune the iterates of the backward fixpoint computations is used by other researchers in verification of hardware [CCQ94b, CCQ94a] and hybrid systems [HH95].

In [BW94], to compute reachable states of a system, repeated executions of loops are computed before starting the symbolic state space exploration. After these transitions, called **meta-transitions**, are added to the system, fixpoint computations are computed on this new transition system. In [BW94], this approach is used for symbolic reachability analysis using periodic sets. Later, in [BG96] and [BGWW97], this approach is adopted for symbolic reachability analysis with QDDs, and dubbed **loop-first search**.

Our approach of computing loop closures is very similar to the approach reported in [BW94]. However, our results generalize the techniques described in [BW94], [BG96] and [BGWW97] to a wider set of properties. Also, we present a more flexible framework by observing that any transition relation R^+ which satisfies $R \subseteq R^+ \subseteq R^*$ can be used for computing reachability properties.

Chapter 8

Two Case Studies

In this chapter we present two case studies: Analysis of (1) a requirements specification for a nuclear reactor's water pressure system [BGL98], and (2) a transport protocol [BG98].

8.1 Analyzing Requirements Specifications

In this section, we demonstrate results of our composite model checking technique using an enhanced version of a known SCR specification, which states the requirements for a nuclear reactor's water pressure system [BH97, CP93, HJL96]. The underlying model contains a good mixture of Booleans, unbounded integers and enumerated types, each of which retain their exact semantic interpretation in our composite model checker. Specifically, this means that during automated analysis checks, important integer values get propagated through the system's transitions – along with the relevant (bounded) values for modes and conditions. This allows us to fully interpret integer-valued functions and predicates appearing in an SCR specification; hence, the model checker automatically deduces inferences such as:

$$(\mathrm{temp} > \mathrm{High} \longleftrightarrow \mathrm{Alarm}) \quad \rightarrow (\exists\, \mathrm{x} < 0 :: \mathrm{temp} + \mathrm{x} = \mathrm{High} \rightarrow \mathrm{Alarm}).$$

With this capability, we need not map SCR integer predicates to Boolean literals (e.g., as in [ABB$^+$96, AB96, AG93, CABN97]); moreover, for our model checker this type of inference comes at no additional cost. Finally, our framework allows us to test mixed integer-Boolean environmental hypotheses. These tests involve queries on possible feedback relationships between values conveyed to actuators (i.e., "controlled variables") and subsequent samples on sensors (i.e., "monitored variables").

8.1.1 Model Checking Requirements Specifications

There has been significant work in using model checking to verify tabular-style SCR requirements. In [AG93] Atlee and Gannon mapped queries about SCR mode-transition tables to the MCB model checker, which uses explicit state enumeration as its underlying representation. Later, Atlee and Buckley improved this process, by using SMV for the same purpose [AB96]. Since SMV [McM93] is a symbolic, BDD-based model checker, it generates more efficient encodings of the SCR state space, which makes it possible to check larger systems. The same tool was also used to analyze parts of the RSML specification of the TCAS II system [ABB$^+$96]. The main difficulty in using SMV for checking requirements specifications seems to be that every variable gets represented in the same symbolic format, namely BDDs. This can easily result in inefficient encodings of totally-ordered variables; moreover, the resulting number of BDD nodes (not to mention their inherent finiteness) often makes verification untenable, at least without human-guided abstractions.

In [CABN97], Chan *et al.* report that representing integers using bitwise BDD representations is not efficient when the input system contains non-linear constraints. They present a technique in which (both linear and non-linear) constraints are mapped to BDD variables (similar representations were also used in [AB96, AG93]). These constraints are used for specifying guarding conditions of transitions. A constraint solver is used during model checking computations (in conjunction with SMV) to prune infeasible combinations of these constraints. Although this technique is capable of handling non-linear constraints, it is restricted to systems where transitions are either data-memoryless (i.e., next state value of a data variable does not depend on its current state value), or data-invariant (i.e., data variables remain unchanged). Hence, even a transition which increments a variable (i.e., $x' = x + 1$) is ruled out. It is reported in [CABN97] that this restriction is partly motivated by the semantics of RSML, and it allows modeling of a significant portion of TCAS II system.

In [BH97] Bharadwaj and Heitmeyer used the SPIN model checker [Hol91, Hol97] to analyze behaviors of SCR specifications as a whole, including all conditions and events, as well as mode transition tables. But since SPIN relies on a finite-state model (like SMV), it cannot check systems with unbounded variables. And indeed, in most of the abovementioned work, abstraction techniques were used to simplify the systems being analyzed. Although we believe that abstraction is bound to play a key role in any automated analysis technique, in some cases it can be avoided by using symbolic representations which can capture the inherent properties of the underlying types. Using arithmetic constraints provides an opportunity to investigate general properties of integer variables, without abstracting their behavior or bounding their domains.

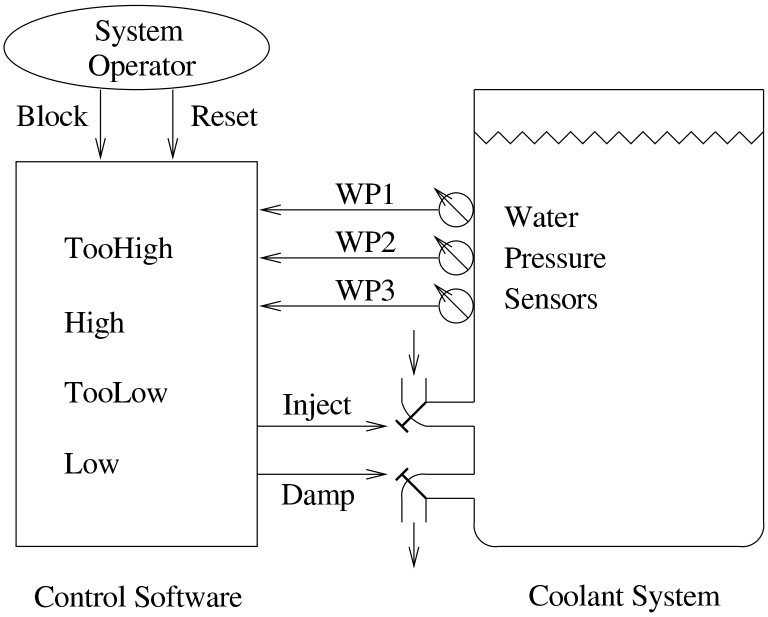


Figure 8.1: A Nuclear Reactor's Water Coolant System

The remainder of this chapter is organized as follows. First, we introduce our motivating SCR example, and explain how we enhanced its behavior to make the verification problem more challenging. Then, we discuss our composite model representation, and show how we translate the SCR requirements into our underlying transition language. Next, we show how our symbolic model checker works in general, and also the results it achieved for the SCR specification. We conclude with some remarks on the results, and outline our future research directions.

8.1.2 Requirements Specifications for the Safety Injection System

In this section we present the requirements for a reactor's cooling system; this example was adapted from previous specifications in [BH97, CP93, HJL96] –

and in fact, we take a superset of these requirements, as well as add a few of our own. The target application is called an "Engineered Safety Feature Actuation System," for a PWR Nuclear Power Plant. It basically functions as a feedback loop: its sensors monitor the coolant system's water pressure via three redundant channels (Figure 8.1). When the pressure is determined to fall below a certain threshold, the processor conveys signals to pressure-control actuators with the objective of increasing it. (This is called "safety injection".) However, the operator may use a manual control (called the "block" switch) to override safety injection during normal start-up or cool-down phases. Moreover (as in [BH97]), the operator also has a "reset switch," which results in the controller program clearing any state triggered by outstanding "block" signals.

In [CP93] and [BH97] this specification is rendered using the SCR tabular notation – although the system given in [BH97] uses only one pressure sensor. Here, we adapt the original three-sensor system to the style of requirements in [BH97], assuming that an action is taken when two of three sensors agree on a condition.

Additionally, we have complicated the system somewhat, by adding an actuator called "damp," which is similar to "safety injection" – but conveys the opposite meaning. When a "damp" signal is sent, it means that incoming water pressure is getting too high, and that the coolant system should start reducing it. Again, the "block" signal can disable this condition (which the reset button can also clear).

The SCR requirements notation is used to specify plant-controller systems; it gives the choice of specifying a state-change explicitly (via events), or implicitly (via current valuations of of other conditions). In SCR, a system's environment

Constants: `min`, `low`, `high`, `toohigh`, `max` : Integer

Monitored Variables: `WP1, WP2, WP3`: Integer; `Block, Reset` : { `On, Off` }

Controlled variables: `Inject, Damp` : { `On Off` }

Terms: `Overridden` : Boolean

Mode Class: `Pressure` : { `TooLow`, `Low`, `High`, `TooHigh` };

Initial Conditions: `Block, Reset, Inject, Damp` = `Off`;

`Overridden` = `False`; `Pressure` = `Low`;

$\mathtt{low} < \mathtt{WP1}, \mathtt{WP2}, \mathtt{WP3} < \mathtt{high}$;

$\mathtt{min} < \mathtt{low} < \mathtt{high} < \mathtt{toohigh} < \mathtt{max}$;

$$\begin{aligned}
\text{CTLow} &\overset{\text{def}}{=} (\mathtt{WP1}, \mathtt{WP2} < \mathtt{low}) \text{ OR } (\mathtt{WP1}, \mathtt{WP3} < \mathtt{low}) \text{ OR } (\mathtt{WP2}, \mathtt{WP3} < \mathtt{low})\\
\text{CLow} &\overset{\text{def}}{=} (\mathtt{low} \leq \mathtt{WP1}, \mathtt{WP2} < \mathtt{high}) \text{ OR } (\mathtt{low} \leq \mathtt{WP1}, \mathtt{WP3} < \mathtt{high}) \text{ OR }\\
&\phantom{\overset{\text{def}}{=}} (\mathtt{low} \leq \mathtt{WP2}, \mathtt{WP3} < \mathtt{high})\\
\text{CHigh} &\overset{\text{def}}{=} (\mathtt{high} \leq \mathtt{WP1}, \mathtt{WP2} < \mathtt{toohigh}) \text{ OR } (\mathtt{high} \leq \mathtt{WP1}, \mathtt{WP3} < \mathtt{toohigh}) \text{ OR }\\
&\phantom{\overset{\text{def}}{=}} (\mathtt{high} \leq \mathtt{WP2}, \mathtt{WP3} < \mathtt{toohigh})\\
\text{CTHigh} &\overset{\text{def}}{=} (\mathtt{toohigh} \leq \mathtt{WP1}, \mathtt{WP2}) \text{ OR } (\mathtt{toohigh} \leq \mathtt{WP1}, \mathtt{WP3}) \text{ OR }\\
&\phantom{\overset{\text{def}}{=}} (\mathtt{toohigh} \leq \mathtt{WP2}, \mathtt{WP3})
\end{aligned}$$

Figure 8.2: SCR Specification of the Safety Injection System (Declarations)

Old Mode	**Event**	**New Mode**
–	`@T(CTLow)`	`TooLow`
–	`@T(CLow)`	`Low`
–	`@T(CHigh)`	`High`
–	`@T(CTHigh)`	`TooHigh`

Mode	**Events**	
`High`	False	`@T(InMode)`
`TooLow, Low, TooHigh`	`@T(Block=On) WHEN Reset=Off`	`@T(InMode)` OR `@T(Reset=On)`
Overridden	True	False

Mode	**Conditions**	
`Low, High, TooHigh`	True	False
`TooLow`	`Overridden`	NOT `Overridden`
`Inject`	`Off`	`On`

Mode	**Conditions**	
`TooLow, Low, High`	True	False
`TooHigh`	`Overridden`	NOT `Overridden`
`Damp`	`Off`	`On`

Figure 8.3: SCR Specification of the Safety Injection System (Tables)

is abstracted as a set of monitored and controlled state variables, corresponding to the sensors and actuators in a control-theoretic setting. Based on the values conveyed on the monitored variables, the system can change its internal state – and send out signals on the controlled variables.

SCR uses three basic constructs to represent software behavior: modes, terms and conditions. A **mode class** is just an enumerated type, denoting an internal state of the controller. Each class has an exclusivity relation between its values – but many classes can be active simultaneously. Modes are usually described behaviorally, in a table denoting current-state/next-state transitions, labeled by conditions that trigger the state-change. A **term** is any function of modes, variables, constants, operators, etc. – i.e., a function built up over the other symbols in an SCR specification. A **condition** is just a term which evaluates to True or False, based on the present state's valuation.

The SCR specification of our safety injection system is given in Figure 8.2 and Figure 8.3. The monitored variables `WP1, WP2, WP3` model the readings from three water pressure sensors. The mode class `Pressure` denotes the controller's state, dependent on the other conditions of the system. Note that it ranges over `TooLow, Low, High, TooHigh` – since we also include an "unsafe" mode for overly-high pressure.

Sensor values change between two constants `min` and `max`. Other constants – `low`, `high`, `toohigh` – indicate the critical pressure levels to which the system reacts. Water pressure is assumed to be dangerously low when it is between `min` and `low`, and too high when it ranges between `toohigh` and `max`. The objective is to maintain the pressure between `low` and `high`.

If at least two of three sensors detect a drop in the water pressure below the constant `low`, this causes the system to enter the mode `TooLow` – and to start safety injection (if it is not overridden). The analogous logic holds for when pressure gets too high.

We show the explicit transition tables for mode `Pressure` and term `Overridden`. As for the controlled variables `Inject` and `Damp`, their state-changes are stated implicitly, via condition tables. In these tables, `@T(InMode)` denotes that the system enters the corresponding mode (i.e., the mode shown on the left-hand-side of the corresponding column).

Note that the system is considered `Overridden` when it is blocked in particular modes (as well as not reset). As for `Inject`, the condition table states that it can only be actuated when the system is in mode `TooLow` and it is not overridden. The conditions for `Damp` are analogous.

The semantics of SCR is defined in [HJL96]. An important restriction of the model is the One Input Assumption, which states that only one monitored variable can change at a time. For the SCR specification given in Figure 8.2 and Figure 8.3 we assume that at any given time, either `Block` or `Reset` can toggle, or the water pressure readings change. However, we assume the pressure sensors are read in on a vector – hence, we treat them as one input.

We also place environmental constraints on the fluctuations of the pressure readings, as in [BH97]. Here, we ensure that readings can change within a certain range, ± `bound`.

Note that in the specification given in Figure 8.2 and Figure 8.3 values of the constants `min`, `max`, `low`, `high`, `toohigh`, and `bound` are unspecified. These constants can take any integer value as long as they satisfy the ordering `min` < `low` < `high` < `toohigh` < `max`. Our representation of the system, which is described below, leaves these constants as unspecified. Hence, any property we verify is valid for any possible interpretation of these constants.

Figures 8.4, 8.5, and 8.6 give the representation of the safety injection system in our event-action language. Using the formal semantics of SCR requirements specifications given in [HJL96], any specification in SCR notation can be automatically converted to our event-action language – and any theorems we prove valid in our model will be true for the original SCR requirements.

We use Boolean variables to encode unordered enumerated SCR variables. Note that we could actually encode `Pressure` using two Boolean variables, but for clarity of presentation we use four. We also define several formulas as abbreviations of complicated expressions – for example, to change conditions, to evaluate voting of sensors, etc. Note that formulas $FInject$ and $FDamp$ define the semantics of condition tables for `Inject` and `Damp`, respectively; similarly $FOver$ defines the semantics of the event table for `Overridden`.

As in the SCR requirements we use the One Input Assumption, which yields 6 events specifying the behavior of the system. At any time only one of the following can occur: *Block* or *Reset* may toggle; or values of $wp1, wp2, wp3$ may change within a range $\pm bound$. (Note however that this range is not specified in advance.) Again, we assume that all three pressure readings are updated at the same time.

In event e_{TLow}, variables $wp1, wp2, wp3$ may change values, and this may cause a change in system state. Specifically, if the previous voting outcome did not detect a pressure reading of `TooLow` – and now it does – then the event can fire, causing changes in other variables too. This corresponds to the action of SCR event `@T(CTLow)`; as in the original specification, the value of the mode class

PROGRAM Safety Injection System

CONSTANTS

$min,\ max,\ low,\ high,\ toohigh, bound$: integer $min < low < high < toohigh < max$

VARIABLES

$wp1, wp2, wp3$: integer

$Block, Reset, Inject, Damp, Over, TLow, Low, High, THigh$: boolean

INITIAL CONDITION

$low \leq wp1, wp2, wp3 < high \ \wedge \ \neg Block \ \wedge \ \neg Reset \ \wedge \ \neg Inject \ \wedge \ \neg Damp \ \wedge \ \neg Over$

EVENTS

e_{TLow} : $RTLow \ \wedge \ TLow' \ \wedge \ \neg Low' \ \wedge \ \neg High' \ \wedge \ \neg THigh' \ \wedge \ Feasible \ \wedge$
$FOver \ \wedge \ FInject \ \wedge \ FInject' \ \wedge \ FDamp \ \wedge \ FDamp'$

e_{Low}: $RLow \ \wedge \ Low' \ \wedge \ \neg TLow' \ \wedge \ \neg High' \ \wedge \ \neg THigh' \ \wedge \ Feasible \ \wedge$
$FOver \ \wedge \ FInject \ \wedge \ FInject' \ \wedge \ FDamp \ \wedge \ FDamp'$

e_{High} : $RHigh \ \wedge \ High' \ \wedge \ \neg TLow' \ \wedge \ \neg Low' \ \wedge \ \neg THigh' \ \wedge \ Feasible \ \wedge$
$FOver \ \wedge \ FInject \ \wedge \ FInject' \ \wedge \ FDamp \ \wedge \ FDamp'$

e_{THigh} : $RTHigh \ \wedge \ THigh' \ \wedge \ \neg TLow' \ \wedge \ \neg Low' \ \wedge \ \neg High' \ \wedge \ Feasible \ \wedge$
$FOver \ \wedge \ FInject \ \wedge \ FInject' \ \wedge \ FDamp \ \wedge \ FDamp'$

e_{Same} : $((CTLow' \ \wedge \ CTLow) \ \vee \ (CLow' \ \wedge \ CLow) \ \vee \ (CHigh' \ \wedge \ CHigh) \ \vee$
$(CTHigh' \ \wedge \ CTHigh)) \ \wedge \ Feasible \ \wedge \ FOver \ \wedge \ FInject \ \wedge$
$FInject' \ \wedge \ FDamp \ \wedge \ FDamp'$

e_{BOrR} : $((Block' = \neg Block \ \wedge \ Reset' = Reset) \ \vee$
$(Reset' = \neg Reset \ \wedge \ Block' = Block)) \ \wedge$
$FOver \ \wedge \ FInject \ \wedge \ FInject' \ \wedge \ FDamp \ \wedge \ FDamp'$

Figure 8.4: Event-Action Language Representation of the Safety Injection System Requirements Specifications

$$CTLow \overset{\text{def}}{=} wp1, wp2 < low \ \lor \ wp1, wp3 < low \ \lor \ wp2, wp3 < low$$

$$CLow \overset{\text{def}}{=} low \leq wp1, wp2 < high \ \lor \ low \leq wp1, wp3 < high \ \lor \ low \leq wp2, wp3 < high$$

$$\begin{aligned} CHigh \overset{\text{def}}{=} {} & high \leq wp1, wp2 < toohigh \ \lor \ high \leq wp1, wp3 < toohigh \ \lor \\ & high \leq wp2, wp3 < toohigh \end{aligned}$$

$$CTHigh \overset{\text{def}}{=} toohigh \leq wp1, wp2 \ \lor \ toohigh \leq wp1, wp3 \ \lor \ toohigh \leq wp2, wp3$$

$$CTLow' \overset{\text{def}}{=} wp1', wp2' < low \ \lor \ wp1', wp3' < low \ \lor \ wp2', wp3' < low$$

$$\begin{aligned} CLow' \overset{\text{def}}{=} {} & low \leq wp1', wp2' < high \ \lor \ low \leq wp1', wp3' < high \ \lor \\ & low \leq wp2', wp3' < high \end{aligned}$$

$$\begin{aligned} CHigh' \overset{\text{def}}{=} {} & high \leq wp1', wp2' < toohigh \ \lor \ high \leq wp1', wp3' < toohigh \ \lor \\ & high \leq wp2', wp3' < toohigh \end{aligned}$$

$$CTHigh' \overset{\text{def}}{=} toohigh \leq wp1', wp2' \ \lor \ toohigh \leq wp1', wp3' \ \lor \ toohigh \leq wp2', wp3'$$

$$RTLow \overset{\text{def}}{=} CTLow' \ \land \ \neg CTLow \qquad RLow \overset{\text{def}}{=} CLow' \ \land \ \neg CLow$$

$$RHigh \overset{\text{def}}{=} CHigh' \ \land \ \neg CHigh \qquad RTHigh \overset{\text{def}}{=} CTHigh' \ \land \ \neg CTHigh$$

$$\begin{aligned} Feasible \overset{\text{def}}{=} {} & (wp1 - bound \leq wp1' \leq wp1 + bound) \ \land \\ & (wp2 - bound \leq wp2' \leq wp2 + bound) \ \land \\ & (wp3 - bound \leq wp3' \leq wp3 + bound) \ \land \ (min \leq wp1', wp2', wp3' \leq max) \end{aligned}$$

Figure 8.5: Abbreviations for Conditions on Water Pressure Sensor Readings of the Safety Injection System

$$
\begin{array}{lcl}
\mathit{FOver} & \stackrel{\text{def}}{=} & (\mathit{Over}' \wedge \mathit{Block}' \wedge \mathit{Block} \wedge \neg \mathit{Reset} \wedge (\mathit{TLow} \vee \mathit{Low} \vee \mathit{THigh})) \\
 & & \vee\, (\neg \mathit{Over}' \wedge (\mathit{Reset}' \wedge \mathit{Reset} \vee \mathit{TLow}' \wedge \mathit{TLow} \vee \\
 & & \mathit{Low}' \wedge \neg \mathit{Low} \vee \mathit{High}' \wedge \neg \mathit{High} \vee \mathit{TooHigh}' \wedge \mathit{TooHigh})) \vee \\
 & & (\mathit{Over}' = \mathit{Over} \wedge (\neg(\mathit{Block}' \wedge \mathit{Block} \wedge \neg \mathit{Reset} \wedge \\
 & & (\mathit{TLow} \vee \mathit{Low} \vee \mathit{High} \vee \mathit{THigh})) \vee \\
 & & \neg(\mathit{Reset}' \wedge \mathit{Reset} \vee \mathit{TLow}' \wedge \mathit{TLow} \vee \mathit{Low}' \wedge \neg \mathit{Low} \vee \\
 & & \mathit{High}' \wedge \neg \mathit{High} \vee \mathit{THigh}' \wedge \mathit{THigh}))) \\
\mathit{FInject} & \stackrel{\text{def}}{=} & (\neg \mathit{Inject} \wedge (\mathit{TLow} \wedge \mathit{Over} \vee \mathit{Low} \vee \mathit{High} \vee \mathit{THigh})) \vee \\
 & & (\mathit{Inject} \wedge \mathit{TLow} \wedge \neg \mathit{Over}) \\
\mathit{FDamp} & \stackrel{\text{def}}{=} & (\neg \mathit{Damp} \wedge (\mathit{TLow} \vee \mathit{Low} \vee \mathit{High} \vee \mathit{THigh} \wedge \neg \mathit{Over})) \vee \\
 & & (\mathit{Damp} \wedge \mathit{THigh} \wedge \neg \mathit{Over})
\end{array}
$$

Figure 8.6: Abbreviations for Conditions on Boolean Variables of the Safety Injection System

`Pressure` changes from `Low` to `TooLow`. Events $e_{Low} - e_{THigh}$ behave similarly, whereas e_{Same} represents the transitions where water pressure conditions remain static. Finally, event e_{BOrR} defines the changes in the system entities when one of the variables *Block* or *Reset* changes.

8.1.3 Automated Analysis of the Safety Injection System

We analyzed the safety injection system using the composite model checking technique discussed in Chapter 5. We only used exact fixpoint computations in this analysis, and we did not partition the state space (i.e., each fixpoint iterate was stored as a single composite representation). We disjunctively partitioned the transition relation using the event syntax (Section 4.2.3).

Two properties of the safety injection system verified in [BH97] are:

$$\text{(SIS1)} \quad \text{AG}((\mathit{Reset} \ \wedge \ \neg \mathit{High}) \rightarrow \neg \mathit{Over})$$

$$\text{(SIS2)} \quad \text{AG}((\mathit{Reset} \ \wedge \ \mathit{TLow}) \rightarrow \mathit{Inject})$$

We verified these properties on the system model presented in Figures 8.4, 8.5, and 8.6. (P1) and (P2) required 2.71 seconds and 2.58 seconds, respectively, as run on a Sun Ultra. Note that, our safety injection system is significantly more complicated than that in [BH97]; in fact, it models the three-way voting scheme as originally specified in [CP93]. This complicates the transition system considerably, since the actions are taken by a majority vote on three different readings – which can range over the entire space of integers. Indeed, note that the system we check (Figure 8.4) is unbounded in most dimensions, since the limit constants $\{\mathit{min}, \mathit{low}, \mathit{high}, \mathit{toohigh}, \mathit{max}\}$ remain unspecified. Hence, any property we check is proved for any concrete values, provided they satisfy the ordering $\mathit{min} < \mathit{low} < \mathit{high} < \mathit{toohigh} < \mathit{max}$. One cannot check such a system with a finite-state model checker like SMV or SPIN, without using some abstraction techniques.

We also wanted to determine whether mode class `Pressure` is a correct abstraction of the water pressure readings. This can be shown by checking the following four properties:

$$\text{(SIS3)} \quad \text{AG}(\mathit{CTLow} \iff \mathit{TLow})$$

$$\text{(SIS4)} \quad \text{AG}(\mathit{CLow} \iff \mathit{Low})$$

$$\text{(SIS5)} \quad \text{AG}(\mathit{CHigh} \iff \mathit{High})$$

$$\text{(SIS6)} \quad \text{AG}(\mathit{CTHigh} \iff \mathit{THigh})$$

where the voting technique introduces the following constraints:

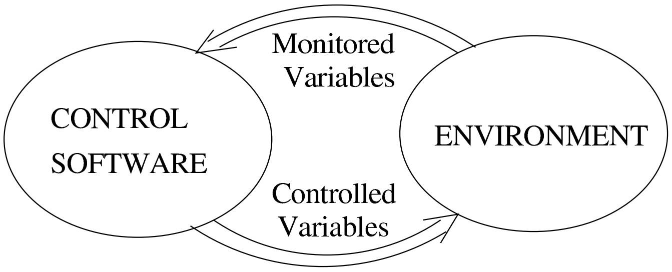


Figure 8.7: Feedback Between the Control Software and the Environment

$$
\begin{aligned}
CTLow &\overset{\text{def}}{=} wp1, wp2 < low \;\vee\; wp1, wp3 < low \;\vee\; wp2, wp3 < low\\
CLow &\overset{\text{def}}{=} low \leq wp1, wp2 < high \;\vee\; low \leq wp1, wp3 < high \;\vee\\
&\quad\; low \leq wp2, wp3 < high\\
CHigh &\overset{\text{def}}{=} high \leq wp1, wp2 < toohigh \;\vee\; high \leq wp1, wp3 < toohigh \;\vee\\
&\quad\; high \leq wp2, wp3 < toohigh\\
CTHigh &\overset{\text{def}}{=} toohigh \leq wp1, wp2 \;\vee\; toohigh \leq wp1, wp3 \;\vee\\
&\quad\; toohigh \leq wp2, wp3
\end{aligned}
$$

The model checker verified these (P3)-(P6) in 16.22, 35.54, 35.39 and 15.94 seconds, respectively. Some other properties we tried were

$$\text{(SIS7)} \quad \mathrm{AG}(Inject \rightarrow TLow)$$

$$\text{(SIS8)} \quad \mathrm{AG}(Damp \rightarrow THigh)$$

which were successfully verified in 1.52 and 1.60 seconds, respectively.

Finally, we wanted to check a liveness property, a property which would state that if certain conditions hold system will eventually go to a good state. For example we may want to check if the system will eventually leave the `TooLow` mode when `Inject` stays `On`. This property can be stated in CTL as:

$$\text{(SIS9)} \quad \mathrm{AG}(Inject \ \mathrm{AU} \ \neg TLow)$$

But this property does not hold for the system given in Figure 8.4. The reason is, there is no causality between safety injection process and water pressure readings. I.e., there is no feedback between the control software and the environment (Figure 8.7). What we would like to do is, then, to change the environment constraints to reflect the effect of the safety injection process, and check the liveness property under the new environment constraints.

Using the notation in Figure 8.4 and Figure 8.5, we can do this quite easily. We just have to change the constraint $Feasible$. Consider the following redefinition of $Feasible$:

$$\begin{aligned}
&Feasible \overset{\text{def}}{=} (\neg Inject \ \wedge \ wp1 - bound \leq wp1' \leq wp1 + bound \ \wedge \\
&\quad wp2 - bound \leq wp2' \leq wp2 + bound \ \wedge \ wp3 - bound \leq wp3' \leq wp3 + bound \ \wedge \\
&\quad min \leq wp1', wp2', wp3' \leq max) \ \vee \ (Inject \ \wedge \ wp1 < wp1' \leq wp1 + bound \ \wedge \\
&\quad wp2 < wp2' \leq wp2 + bound \ \wedge \ wp3 < wp3' \leq wp3 + bound \ \wedge \\
&\quad min \leq wp1', wp2', wp3' \leq max)
\end{aligned}$$

This constraint basically states that water pressure increases when the safety injection is on. Now, we can try to verify our liveness property using the new $Feasible$ constraint. But, when we run our model checker on this system it does not converge. The reason is that since all the constants in the system are uninterpreted, there is no fix number of steps after which the system will reach $\neg TLow$. Hence, the fixpoint computations do not converge after fixed number of steps. This problem maybe solved using conservative approximation techniques such as widening. Unfortunately, a standard application of the widening technique would not work since we need a lower bound for a least fixpoint. However, an approach which combines automated theorem proving with widening technique to guess and verify least fixpoints may be possible [HR98]. Another solution would be to force water pressure to increase with a fixed amount when safety injection is on. For example we can redefine $Feasible$ as

$$\begin{aligned}
&Feasible \stackrel{\text{def}}{=} (\neg Inject \ \wedge \ wp1 - bound \leq wp1' \leq wp1 + bound \wedge \\
&\quad wp2 - bound \leq wp2' \leq wp2 + bound \ \wedge \ wp3 - bound \leq wp3' \leq wp3 + bound \wedge \\
&\quad min \leq wp1', wp2', wp3' \leq max) \ \vee \ (Inject \ \wedge \ wp1' = wp1 + (low - min) \wedge \\
&\quad wp2' = wp2 + (low - min) \ \wedge \ wp3' = wp3 + (low - min) \wedge \\
&\quad min \leq wp1', wp2', wp3' \leq max)
\end{aligned}$$

This will force the system to get out of `TooLow` mode after one execution step. Using this constraint we tried to check the liveness property (SIS9) again. This time the liveness property was verified in 32.11 seconds.

8.2 Analyzing a Transport Protocol

In this section we will demonstrate the effectiveness of our methods by analyzing a protocol. The protocol we will present is a one-directional transport protocol, abstracted from the TCP specification (RFC 793). Although abstractions and simplifications have been made, much of TCP's complexity remains. We analyze parts of the protocol, specifically: handshaking for call setup and take-down, as well as reliable data-transfer using sliding-window protocol [Ste76] over an unreliable channel, which can delay, duplicate, lose or reorder messages to an unlimited extent (this is modeled using existential quantification). Moreover, the protocol's send and receive window sizes are not specified in advance; rather, they are represented as symbolic constants. There are several points to the exercise: (1) the properties examined rely on multiple state changes, triggered by conditions over both Boolean and integer-valued variables; (2) we verify these properties without bounding window sizes or sequence numbers; hence, correctness is assured for any implementation of the protocol; (3) automated verification was only possible using composite models, as well as approximation techniques based on their operators.

The protocol system contains a mixture of integer and Boolean-valued variables. Hence, it was not amenable to verification by a BDD-checker alone; moreover, its size (and the number of Boolean variables) made pure-Presburger checking infeasible. In this chapter, we show how the system was automatically verified using our composite model checking approach, in concert with a conservative approximation technique.

In [Kai97], a different version of a sliding-window protocol [Ste76] was verified, using a compositional preorder approach. In certain respects, this previous study was more broad, e.g., it handled liveness properties for arbitrary channel lengths. In other respects, it was more constrained, e.g., it assumed a fixed window size, whereas we verify the problem for any window size. But essentially, while examining a similar problem, the work in [Kai97] and our approach illustrate two different (but related) techniques. In [Kai97], verification is carried out in a semi-automated fashion, where some key user-generated abstractions were required. Here, the protocol is verified automatically, via composite model checking. In the future, we would expect to see many of these concepts used together, toward solving more complex problems.

8.2.1 A Transport Protocol

Figure 8.8 shows how synchronization and data-transfer are implemented in our transport protocol, which involves one sender process and one receiver process. A sender can be in one of the following states: `closed`, `syn_sent`, `established`, and `fin_wait`, whereas the receiver process will either be in `listen`, or `established` states.

After the connection is established via hand-shaking, and both parties are in the `established` state, they use a sliding-window protocol to send and receive data, where each message has a unique sequence number. Here, it is sufficient to model the sequence numbers, as well as the send and receive pointers, and to abstract away the message contents.

As specified in TCP, a sender has two integer pointers snd_una (oldest unacknowledged sequence number), and snd_nxt (next sequence number to be sent). On the other hand, the receiver maintains a single integer pointer rcv_nxt (next sequence number to be received). Again, these variables can be unbounded;

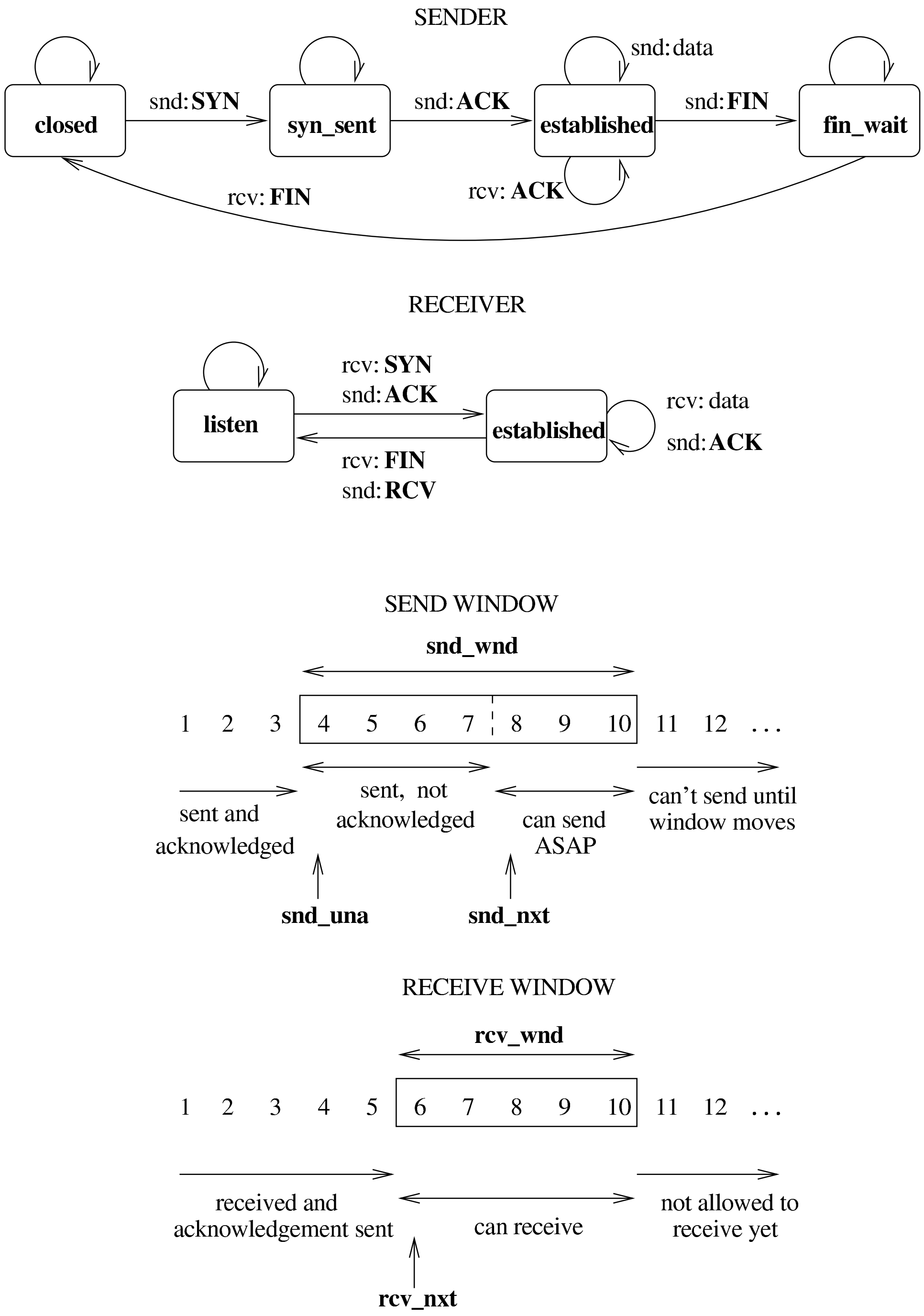


Figure 8.8: A Transport Protocol

```
typedef enum { closed, syn_sent, established, fin_wait } sender_state;
typedef enum { listen, established } receiver_state;
typedef enum { syn, ack, snt, fin } signal;
typedef enum { send, receive_ack, finish } event;

int snd_una, snd_nxt, rcv_nxt;
int snd_wnd, rcv_wnd;
int seg_seq, seg_len, seg_ack;
sender_state sender;
receiver_state receiver;

signal signal_from_net();
int seq_num_from_net();
int seg_len_from_net();
to_net( signal sent_signal, int seq_num, int seg_len );

event wakeup_event();
```

Figure 8.9: A Transport Protocol Implementation (Declarations)

hence, it is not hard to see how the underlying transition system is an infinite one, and why it is not amenable to known automated techniques.

The integers *snd_wnd* and *rcv_wnd* denote the sizes of sender and receiver windows, respectively. These are unspecified constants; hence, a property verified for the protocol will hold for any interpretation of these constants. As usual, the windows are used to avoid buffer overflow on the sender and receiver sides. The sender can have *snd_wnd* messages outstanding – a situation maintained by ensuring that $snd\_nxt - snd\_una \leq snd\_wnd$. Also, the receiver can only queue up *rcv_wnd* messages, and additional messages will be lost.

Figures 8.9, 8.10 and 8.11 demonstrate a possible implementation of the protocol.

```
sender_process()
{
  switch ( sender )
  {
    case closed:
      to_net( syn, dummy, dummy );
      sender = syn_sent;
      break;
    case syn_sent:
      if ( signal_from_net() == ack )
      {
        sender = established;
        snd_una = snd_nxt = 0;
      }
      break;
    case established:
      switch  ( wakeup_event() )
      {
        case send:
          if ( snd_nxt + seg_len <= snd_una + snd_wnd )
          {
            to_net( snt, snd_nxt, seg_len );
            snd_nxt = snd_nxt + seg_len;
          }
          else
            print_error( "message length exceeds send window" );
          break;
        case receive_ack:
          seg_ack = seq_num_from_net();
          snd_una = seg_ack;
          break;
        case finish:
          to_net( fin, dummy, dummy );
          sender = fin_wait;
      }
      break;
    case fin_wait:
      if ( signal_from_net() == fin )
        sender = closed;
      break;
  }
}
```

Figure 8.10: A Transport Protocol Implementation (Sender Process)

```
receiver_process()
{
  switch ( receiver )
  {
    case listen:
      if ( signal_from_net() == syn )
      {
        receiver = established;
        rcv_nxt = 0;
        to_net( ack, dummy, dummy );
      }
      break;
    case established:
      if ( signal_from_net() == snt )
      {
        seg_seq = seq_num_from_net();
        seg_len = seg_len_from_net();

        if ( (rcv_nxt <= seg_seq < rcv_nxt + rcv_wd) ||
             (rcv_nxt <= seg_seq + seg_len - 1 < rcv_nxt + rcv_wnd) )
        {
          if ( seg_seq + seg_len > rcv_nxt + rcv_wnd )
            rcv_nxt = rcv_nxt + rcv_wnd;
          if ( seg_seq + seg_len <= rcv_nxt + rcv_wnd )
            rcv_nxt = seg_seq + seg_len;
          to_net( ack, rcv_nxt, dummy );
        }
        else
          print_error( "message is outside of the receive window" );
      }
      else if ( signal_from_net() == fin )
      {
        receiver = listen;
      }
      break;
  }
}
```

Figure 8.11: A Transport Protocol Implementation (Receiver Process)

```
PROGRAM   A Transport Protocol
CONSTANTS
  snd_wnd: positive integer   // sender window
  rcv_wnd: positive integer   // receiver window
VARIABLES
  sender : {closed, syn_sent, established, fin_wait}   // sender state
  receiver : {listen, established}   // receiver state
  snd_una: positive integer   // oldest unacknowledged sequence number
  snd_nxt: positive integer   // next sequence number to be sent
  rcv_nxt: positive integer   // next sequence number to be received
  syn: boolean   // if true, then sender has sent a syn
  ack: boolean   // if true, then receiver has sent an ack
  snt: boolean   // if true, then receiver has sent data
  finS: boolean   // if true, then sender has sent a fin
  finR: boolean   // if true, then receiver has sent a fin
INITIAL CONDITION
  snd_una = snd_nxt = rcv_nxt = 0  ∧  sender = closed  ∧  receiver = listen
  ∧ syn = ack = snt = finS = finR = false
```

Figure 8.12: Variable Declarations for the Transport Protocol

8.2.2 Event-Action Language Specification of the Transport Protocol

Figures 8.12, 8.13 and 8.14 show our event-action language description of the protocol. This is the model we analyze with our composite model checker. Below we describe the behavior of the protocol by explaining the events individually.

Upon start-up, the sender forwards a *syn* message to the receiver, and then it goes into the `syn_sent` state (event e_{S1}). When the receiver gets the *syn*,

EVENTS

// SENDER EVENTS

e_{S1} : // sender goes from closed to syn_sent

$sender = \texttt{closed} \ \wedge\ sender' = \texttt{syn\_sent} \ \wedge\ syn' = \textbf{true} \ \wedge\ finS' = \textbf{false}$

e_{S2} : // sender receives ack and goes from syn_sent to established

$ack = \textbf{true} \ \wedge\ sender = \texttt{syn\_sent} \ \wedge\ sender' = \texttt{established} \ \wedge\ syn' = \textbf{false}$

$\wedge\ snd\_una' = snd\_nxt' = 0$

e_{S3} : // sender sends data in established

$sender = \texttt{established} \ \wedge\ snt' = \textbf{true} \wedge$

$(\exists seg\_len \ : \ 0 < seg\_len \ \wedge\ snd\_nxt + seg\_len \leq snd\_una + snd\_wnd$

$\wedge\ snd\_nxt' = snd\_nxt + seg\_len) \ \vee\ snd\_nxt' = snd\_nxt$

e_{S4} : // sender receives ack in established

$sender = \texttt{established} \wedge$

$(\exists seg\_ack : 0 \leq seg\_ack \ \wedge\ seg\_ack \leq rcv\_nxt \ \wedge\ snd\_una < seg\_ack \leq snd\_nxt$

$\wedge\ snd\_una' = seg\_ack) \ \vee\ snd\_una' = snd\_una$

e_{S5} : // sender goes from established to fin_wait

$sender = \texttt{established} \ \wedge\ sender' = \texttt{fin\_wait} \ \wedge\ finS' = \textbf{true}$

e_{S6} : // sender goes from fin_wait to closed after receiving fin

$finR = \textbf{true} \ \wedge\ sender = \texttt{fin\_wait} \ \wedge\ sender' = \texttt{closed} \wedge$

$syn' = finS' = snt' = \textbf{false}$

Figure 8.13: Sender Events for the Transport Protocol

EVENTS

// RECEIVER EVENTS

e_{R1} : // receiver goes from listen to established after receiving syn

$\mathit{syn} = \mathbf{true} \ \land \ \mathit{receiver} = \texttt{listen} \ \land \ \mathit{receiver}' = \texttt{established} \land$

$\mathit{ack}' = \mathbf{true} \ \land \ \mathit{finR}' = \mathbf{false} \ \land \ \mathit{rcv\_nxt}' = 0$

e_{R2} : // receiver receives data in established

$\mathit{snt} = \mathbf{true} \ \land \ \mathit{receiver} = \texttt{established} \land$

$(\exists \mathit{seg\_seq}, \mathit{seg\_len} \ : \ 0 \leq \mathit{seg\_seq} \ \land \ 0 < \mathit{seg\_len} \ \land \ \mathit{rcv\_wnd} > 0 \ \land$

$(\mathit{seg\_seq} + \mathit{seg\_len} \leq \mathit{snd\_nxt}) \ \land \ (\mathit{rcv\_nxt} \leq \mathit{seg\_seq} < \mathit{rcv\_nxt} + \mathit{rcv\_wnd} \ \lor$

$\mathit{rcv\_nxt} \leq \mathit{seg\_seq} + \mathit{seg\_len} - 1 < \mathit{rcv\_nxt} + \mathit{rcv\_wnd}) \ \land$

$((\mathit{seg\_seq} + \mathit{seg\_len} > \mathit{rcv\_nxt} + \mathit{rcv\_wnd} \ \land \ \mathit{rcv\_nxt}' = \mathit{rcv\_nxt} + \mathit{rcv\_wnd}) \ \lor$

$(\mathit{seg\_seq} + \mathit{seg\_len} \leq \mathit{rcv\_nxt} + \mathit{rcv\_wnd} \ \land \ \mathit{rcv\_nxt}' = \mathit{seg\_seq} + \mathit{seg\_len}))) \ \lor$

$\lor \ \mathit{rcv\_nxt}' = \mathit{rcv\_nxt}$

e_{R3} : // receiver goes from established to listen after receiving fin

$\mathit{finS} = \mathbf{true} \ \land \ \mathit{receiver} = \texttt{established} \ \land \ \mathit{receiver}' = \texttt{listen} \land$

$\mathit{finR}' = \mathbf{true} \ \land \ \mathit{ack}' = \mathbf{false}$

// IDLE EVENT

e_I : // receiver and sender events can be delayed indefinitely

true

Figure 8.14: Receiver and Idle Events for the Transport Protocol

it acknowledges via an *ack*; then it goes into the `established` state (event e_{R1}). When the send party receives the *ack*, it likewise transitions into the `established` state (e_{S2}). Note that both parties initialize their sequence number pointers to 0 before starting data-transfer, which works in a "stop-and-wait" manner: The sender repeatedly sends messages within its current window (event e_{S3}), and updates *snd_una* when acknowledgments are received (e_{S4}). The receiver, on the other hand, attempts to receive new messages within its own window (event e_{R2}); when the input queue bound is exceeded, the excess is dropped.

To close a connection, the sender sends a $finS$ message, and goes to the `fin_wait` state (event e_{S5}). When receiver receives this signal, it responds with a $finR$, and then it goes back to `listen`. Finally, when sender receives this signal, it goes back to `closed` state, and the hand-shake is complete.

We assume that the communication channel is unreliable, i.e., it can delay, lose, duplicate or reorder outstanding messages. To model this, we use shared variables and existential quantification, which include all the possible behaviors such a channel would generate. Also, note the presence of an idle event – which is used to delay messages for an arbitrary number of transitions before being received (if ever). We do assume, however, that there are no delayed duplicates in the network – i.e., when a new session of the protocol is instantiated, there are no previous syn, fin, snt, and ack signals hanging around in the network.

8.2.3 Automated Analysis of the Protocol

We analyzed the transport protocol presented above using the composite model checking technique discussed in Chapter 5. We used both exact and approxi-

PROP.	FIXPOINT	CONVERGED IN
(TP1)	Exact	4 iterations – 3.11 sec.
	Approximate	4 iterations – 3.42 sec.
(TP2)	Exact	did not converge
	Approximate	6 iterations – 14.13 sec.
(TP3)	Exact	1 iterations – 1.74 sec.
	Approximate	1 iterations – 1.93 sec.
(TP4)	Exact	1 iterations – 1.73 sec.
	Approximate	1 iterations – 1.71 sec.
(TP5)	Exact	7 iterations – 91.97 sec.
	Approximate	4 iterations – 11.52 sec.
(TP6)	Exact	did not converge
	Approximate	10 iterations – 377.44 sec.

Table 8.1: Verified Properties for the Transport Protocol

mate fixpoint computations. We applied the widening technique to composite representations as described in Section 6.3. We disjunctively partitioned the transition relation using the event syntax, but we did not use any state space partitioning technique (Section 4.2.3).

Some interesting properties of this data transfer protocol are as follows:

- (TP1) $\mathrm{AG}(\mathit{sender} = \texttt{established} \rightarrow \mathit{receiver} = \texttt{established})$:

  If the sender establishes a connection, then so does the receiver.

- (TP2) $\mathrm{AG}(\mathit{receiver} = \texttt{established} \rightarrow \neg(\mathit{sender} = \texttt{closed}))$: If the receiver establishes a connection, then the sender is not closed.

- (TP3) $\mathrm{AG}(sender = \texttt{established} \rightarrow snd\_una \leq snd\_nxt \leq snd\_una + snd\_wnd)$: The sender does not overload the send buffer.
- (TP4) $\mathrm{AG}((receiver = \texttt{established} \wedge rcv\_nxt = i) \rightarrow \mathrm{AX}(rcv\_nxt \leq i + rcv\_wnd))$: The receiver does not overload the receive buffer.
- (TP5) $\mathrm{AG}(sender = receiver = \texttt{established} \rightarrow snd\_una \leq rcv\_nxt)$: All the acknowledged messages are received by the receiver.
- (TP6) $\mathrm{AG}(sender = receiver = \texttt{established} \rightarrow rcv\_nxt \leq snd\_nxt)$: All messages received were actually sent by the sender.

Figure 8.1 shows the results of our experiments. Property (TP1) converged in 4 iterations, using both exact and approximate fixpoint computations; and due to additional cost of widening, the approximate fixpoint actually used a bit more time. However, property (TP2) does not converge at all using the exact method – even though the property does not use any integer variables. However, this is not too surprising, since within the protocol, almost all state-changes are due to a combination of interacting variables, both Booleans and integers. We were able to prove property (TP2) using the approximate fixpoint computations.

On the other hand, properties (TP3) and (TP4) are both built up exclusively using integer variables – yet, both converge exactly in one iteration; indeed they follow from the event specifications of the sender and the receiver.

Property (TP5) establishes the relationship between *snd_una* and *rcv_nxt*. Since *snd_una* is updated by the sender, and since *rcv_nxt* is updated by the receiver, verification of this property involves considering all possible concurrent executions. The model-checker verifies the property in 7 iterations using exact fixpoint computations, whereas the approximate fixpoint computations converges much faster in 4 iterations. This shows that conservative approximations are not only useful for approximating divergent fixpoint computations; they can also be used to get quicker results in general.

Property (TP6) is another property which involves both the sender and the receiver processes, and it shows the relationship between variables *rcv_nxt* and *snd_nxt*. Again *rcv_nxt* is updated by the receiver, and *snd_nxt* is a variable updated by the sender. While exact analysis did not converge for this property, the approximate fixpoint computations converged in 10 iterations.

Chapter 9

Conclusions

In this dissertation we presented several methods to improve the effectiveness of model checking in analyzing software systems. First, we presented a Presburger arithmetic model checker for analyzing programs with unbounded integer variables. Some features of our Presburger model checker are as follows: (1) it symbolically encodes transition relations and state sets using Presburger formulas, (2) it partitions a program's state-space via the control variables or domains of events, and uses the partition classes as repositories for the model checker's symbolic computations; and (3) it uses a fast constraint manipulator called the Omega Library [KMP$^+$95, Pug92] for propagating Presburger formulas during the fixpoint computations.

We also developed a method for combining different symbolic representations in a single composite model checker. BDDs have been very successful in encoding Boolean domains, whereas our Presburger encoding enables us to analyze systems with unbounded integer variables. Our composite model checker exploits strengths of both BDD and Presburger representations by using type-specific encodings. To do this, we conjunctively partition the atomic events of the system based on the underlying variable types, and then compute each type's relational image separately. Our method also forms a general framework for combining multiple type-specific symbolic representations in a single model checker.

To address the issue of undecidability of model checking for infinite-state systems, we investigated conservatively approximating temporal properties. Given a temporal property conservative approximation techniques either 1) prove the property, or 2) disprove the property by providing a counter-example, or 3) report that the analysis is inconclusive. Using this framework we were able to integrate abstract interpretation techniques to our model checker. Conservative approximations also enable us to analyze large systems incrementally.

We also developed new techniques for reachability analysis. Using these techniques, a set of temporal properties which we named reachability properties, can be computed faster.

Finally, we built a prototype analysis tool which implements these methods. We used this tool to automatically analyze some subtle, nontrivial infinite-state systems – the type of systems usually analyzed by hand or via on-line testing. Particularly, we presented two case studies: Analysis of (1) a requirements specification for a nuclear reactor's water pressure system, and (2) a transport protocol.

9.1 Future Work

We are interested in expanding our composite model checker by adding other symbolic representations such as automata based representations [BG96, BGWW97, BK95, BKR96, GL96, HJJ$^+$95, Kla97], real arithmetic [ACHH93, ACH$^+$95], etc. This will allow us to encode variable types like reals and queues. It would be interesting to investigate the scalability of our approach to several data types.

Also our conservative approximation techniques can be extended. There are two basic ways of approximating truth sets of temporal formulas: (1) Using approximate fixpoint computations (this is the approach we adopted), and (2) using approximate models which over- or under-estimate behaviors of a system [DGG97, KDG95]. These two techniques can be combined to form a model checker that would be capable of both compositional and incremental analysis. We currently compute the fixpoints over the Cartesian-product of all variable domains. Using the conservative approximation approach, we could integrate compositional strategies developed for finite-state systems [BFG96, CLM89] in our analysis.

Finally, we would like to test presented techniques on larger systems, and determine its feasibility for checking "industrial-strength" examples. We think that using various abstraction techniques – in conjunction with some human-guided interaction – we should be able to attack these significantly larger systems.

Appendix A

The Prototype Tool

We developed a prototype analysis tool in C++ that implements the methods described in this dissertation. The software architecture of our tool is shown in Figure A.1. Its layered structure is designed to facilitate future extensions.

The front end compiler translates the programs written in a simple event-action language to symbolic representations: Boolean logic and Presburger arithmetic formulas. The user can enter temporal queries about the input program using the temporal logic CTL.

In the lower level, we have forward, backward, exact, and approximate fixpoint computations for computing truth sets of temporal properties. These fixpoint computations can be computed over the partitioned state space of the program. We also have reachability techniques which can increase the convergence rate of the fixpoint computations. The user can combine these techniques in various ways to prove a single property.

The lowest level of our layered architecture consists of libraries implementing our symbolic representations. Our tool uses two class libraries for encoding type-specific constraints. While both share the same API, one of them (a BDD implementation) is used for Boolean and enumerated types, and the other (the

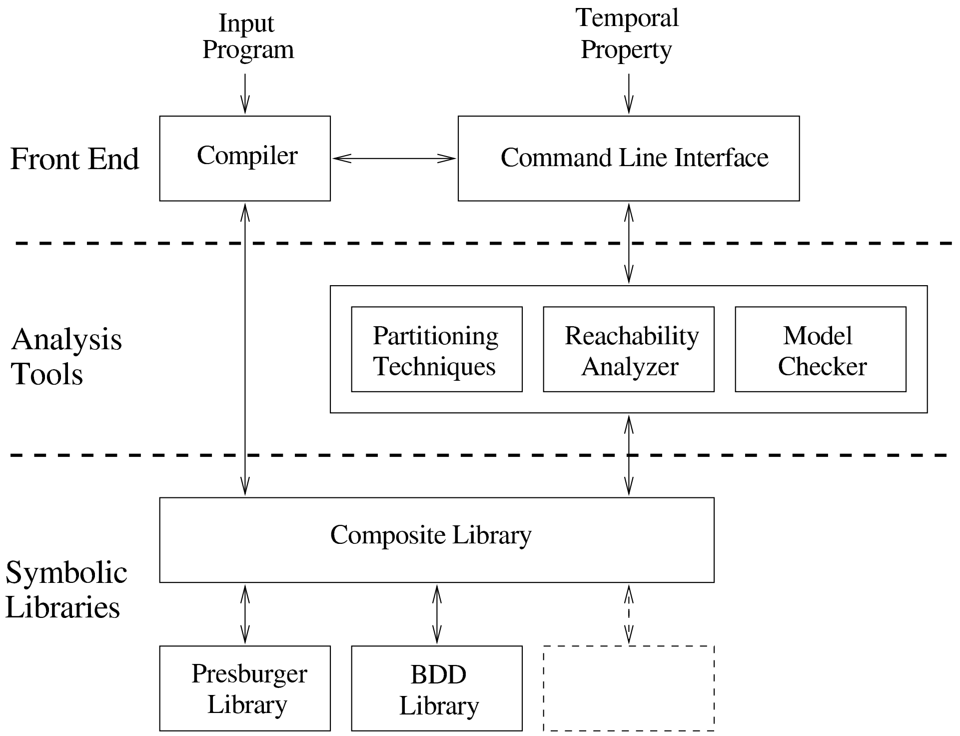


Figure A.1: Software Architecture of the Prototype Tool

Omega Library) is used for integer variables. Then a composite-model library handles operations over mixed-type constraints (e.g., set-inclusion, intersection, etc.); in turn, these operations invoke their relevant type-specific counterparts to help carry out the desired effect.

Our approach to composite representations – and their orthogonal implementations – will allow expanding to additional datatypes in the future. Note that given an adequate symbolic representation, we can easily integrate its implementation to our software architecture (the dashed box in Figure A.1).

Bibliography

[AB96] J. M. Atlee and M. A. Buckley. A logic-model semantics for SCR software requirements. In *Proceedings of the 1996 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 280–292, January 1996.

[ABB+96] R. J. Anderson, P. Beame, S. Burns, W. Chan, F. Modugno, D. Notkin, and J. D. Reese. Model checking large software specifications. In *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 156–166, October 1996.

[ACH+95] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, X. Nicollin P. H. Ho, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1):3–34, 1995.

[ACHH93] R. Alur, C. Courcoubetis, T. A. Henzinger, and P. H. Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In R. L. Grossman, A. Nerode, A. P. Ravn, and H. Richel, editors, *Hybrid Systems*, volume 736 of *Lec-*

ture Notes in Computer Science, pages 209–229. Springer-Verlag, 1993.

[AG93] J. M. Atlee and J. Gannon. State-based model checking of event-driven system requirements. *IEEE Transactions on Software Engineering*, 19(1):24–40, January 1993.

[AHH96] R. Alur, T. A. Henzinger, and P. Ho. Automatic symbolic verification of embedded systems. *IEEE Transactions on Software Engineering*, 22(3):181–201, March 1996.

[And91] G. R. Andrews. *Concurrent Programming: Principles and Practice.* The Benjamin/Cummings Publishing Company, Redwood City, California, 1991.

[Arn94] A. Arnold. *Finite Transition Systems: Semantics of Communicating Systems.* Prentice Hall, Englewood Cliffs, New Jersey, 1994.

[BC95] R. E. Bryant and Y. Chen. Verification of arithmetic functions with binary moment diagrams. In *Proceedings of the 32nd ACM/IEEE Design Automation Conference*, June 1995.

[BCL$^+$94] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4):401–424, April 1994.

[BCM$^+$90] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. H. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Pro-*

ceedings of the 5th Annual IEEE Symposium on Logic in Computer Science, pages 428–439, January 1990.

[BFG96] T. Bultan, J. Fischer, and R. Gerber. Compositional verification by model checking for counter-examples. In *Proceedings of the 1996 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 224–238, January 1996.

[BG96] B. Boigelot and P. Godefroid. Symbolic verification of communication protocols with infinite state spaces using QDDs. In *Proceedings of the 8th International Conference on Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 1–12. Springer-Verlag, August 1996.

[BG98] T. Bultan and R. Gerber. Composite model checking with type specific symbolic encodings. Technical Report CS-TR-3871, Department of Computer Science, University of Maryland, College Park, February 1998.

[BGL98] T. Bultan, R. Gerber, and C. League. Verifying systems with integer constraints and boolean predicates: A composite approach. In *Proceedings of the 1998 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 113–123, March 1998.

[BGP96] T. Bultan, R. Gerber, and W. Pugh. Symbolic model checking of infinite state programs using presburger arithmetic. Technical Report CS-TR-3687, Department of Computer Science, University of Maryland, College Park, September 1996.

[BGP97] T. Bultan, R. Gerber, and W. Pugh. Symbolic model checking of infinite state systems using presburger arithmetic. In O. Grumberg, editor, *Proceedings of the 9th International Conference on Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 400–411. Springer, June 1997.

[BGP98] T. Bultan, R. Gerber, and W. Pugh. Model checking concurrent systems with unbounded integer variables: Symbolic representations, approximations and experimental results. Technical Report CS-TR-3870, Department of Computer Science, University of Maryland, College Park, February 1998.

[BGWW97] B. Boigelot, P. Godefroid, B. Williams, and P. Wolper. The power of QDDs. In *Proceedings of the Fourth Static Analysis Symposium*, September 1997.

[BH97] R. Bharadwaj and C. Heitmeyer. Verifying SCR requirements specifications using state exploration. In *Proceedings of First ACM SIGPLAN Workshop on Automatic Analysis of Software*, January 1997.

[BK95] D. A. Basin and N. Klarlund. Hardware verification using monadic second-order logic. In *Proceedings of the 7th International Conference on Computer Aided Verification*, volume 939 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[BKR96] M. Biehl, N. Klarlund, and T. Rauhe. Mona: decidable arithmetic in practice. In *Proceedings of Formal Techniques in Real-Time and Fault-Tolerant Systems, 4th International Symposium*, volume 1135 of *Lecture Notes in Computer Science*. Springer, 1996.

[BM79] R. S. Boyer and J. S. Moore. *A computational logic*. Academic Press, New York, 1979.

[BM88] R. S. Boyer and J. S. Moore. *A computational logic handbook*. Academic Press, New York, 1988.

[Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[BW94] B. Boigelot and P. Wolper. Symbolic verification with periodic sets. In *Proceedings of the 6th International Conference on Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 55–67. Springer-Verlag, June 1994.

[BY96] R. Boyer and Y. Yu. Automated proofs of object code for a widely used microprocessor. *Journal of the ACM*, 43(1):166–192, January 1996.

[CABN97] W. Chan, R. Anderson, P. Beame, and D. Notkin. Combining constraint solving and symbolic model checking for a class of systems with non-linear constraints. In O. Grumberg, editor, *Proceedings of the 9th International Conference on Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 316–327. Springer, June 1997.

[CBM89] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. In J. Sifakis, editor, *Automatic Verification Methods for Finite State*

Systems, volume 407 of *Lecture Notes in Computer Science*, pages 365–373. Springer-Verlag, June 1989.

[CC76] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proceedings of the 2nd International Symposium on Programming*, pages 106–130, 1976.

[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 ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[CC92a] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of logic programming*, 13:103–179, 1992.

[CC92b] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.

[CCQ94a] G. Cabodi, P. Camurati, and S. Quer. Efficient state space pruning in symbolic backward traversal. In *Proceedings 1994 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, October 1994.

[CCQ94b] G. Cabodi, P. Camurati, and S. Quer. Symbolic exploration of large circuits with enhanced forward/backward traversals. In *Proceedings: EURO-DAC'94 with EURO-VHDL'94*, pages 22–27, September 1994.

[CE81] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In D. Kozen, editor, *Logic of Programs: Workshop*, volume 131 of *Lecture Notes in Computer Science*. Springer-Verlag, May 1981.

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

[CFZ95] E. Clarke, M. Fujita, and X. Zhao. Applications of multi-terminal binary decision diagrams. Technical Report CMU-CS-95-160, School of Computer Science, Carnegie Mellon University, April 1995.

[CGKM94] E. M. Clarke, O. Grumberg, and X. Zhao K. McMillan. Efficient generation of counterexamples and witnesses in symbolic model checking. Technical Report CMU-CS-94-204, School of Computer Science, Carnegie Mellon University, October 1994.

[CGL92] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. In *Proceedings of the 18th Annual ACM Symposium on Principles of Programming Languages*, pages 343–354, 1992.

[CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th Annual ACM Symposium on Principles of Programming*, pages 84–97, 1978.

[CLM89] E. M. Clarke, D. E. Long, , and K. L. McMillan. Compositional model checking. In *Proceedings of the 4th Annual IEEE Symposium on Logic in Computer Science*, pages 464–475, June 1989.

[CMB90] O. Coudert, J. C. Madre, and C. Berthet. Verifying temporal properties of sequential machines without building their state diagrams. In *Proceedings of the 2nd International Conference on Computer Aided Verification*, volume 531 of *Lecture Notes in Computer Science*, pages 23–32. Springer, June 1990.

[Coo71] D. C. Cooper. Programs for mechanical program verification. In *Machine Intelligence 6*, pages 43–59, New York, 1971. American Elsevier.

[CP93] P. J. Courtois and D. L. Parnas. Documentation for safety critical software. In *Proceedings of the 15th International Conference on Software Engineering*, pages 315–323, May 1993.

[CW96] E. M. Clarke and J. M. Wing. Formal methods: State of the art and the future. *ACM Computing Surveys*, 28(4):626–643, December 1996.

[CZ95] E. Clarke and X. Zhao. Word level symbolic model checking: A new approach for verifying arithmetic circuits. Technical Report CMU-CS-95-161, School of Computer Science, Carnegie Mellon University, May 1995.

[Dam94] M. Dam. CTL* and ECTL* as fragments of the modal μ-calculus. *Theoretical Computer Science*, 126:77–96, 1994.

[DF95] J. Dingel and T. Filkorn. Model checking for infinite state systems using data abstraction. In *Proceedings of the 7th International Conference on Computer Aided Verification*, volume 939 of *Lecture Notes in Computer Science*, pages 54–69. Springer-Verlag, 1995.

[DGG97] D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems*, 19(2):253–291, March 1997.

[Dij75] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[Dil90] L. K. Dillon. Using symbolic execution for verification of ada tasking programs. *ACM Transactions on Programming Languages and Systems*, 12(4):643–669, October 1990.

[DP98] G. Delzanno and A. Podelski. Model checking infinite-state systems in CLP. Technical Report MPI-I-98-2-012, Max Planck Institut fuer Informatik, July 1998.

[Eme90] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*. Elsevier Science Publishers, 1990.

[Flo67] R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Proceedings of the Symposium in Applied Mathematics 19*, pages 19–32, 1967.

[FR74] M. J. Fischer and M. O. Rabin. Super-exponential complexity of presburger arithmetic. In *SIAM-AMS Proceedings*, volume 7, pages 27–41, 1974.

[GL94] O. Grumberg and D. E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, May 1994.

[GL96] P. Godefroid and D. Long. Symbolic protocol verification with queue BDDs. In *Proceedings of the 11th Symposium on Logic in Computer Science*, pages 198–206, July 1996.

[God94] P. Godefroid. *Partial-order methods for the verification of concurrent systems: An approach to the state-explosion problem.* PhD thesis, Universite De Liege, 1994.

[Hal93] N. Halbwachs. Delay analysis in synchronous programs. In C. Courcoubetis, editor, *Proceedings of computer aided verification*, volume 697 of *Lecture Notes in Computer Science*, pages 333–346. Springer-Verlag, 1993.

[HH95] T. A. Henzinger and P. Ho. A note on abstract-interpretation strategies for hybrid automata. In P. Antsaklis, W. Kohn, A Nerode, and S. Sastry, editors, *Hybrid Systems II*, volume 999 of *Lecture Notes in Computer Science*, pages 252–264. Springer-Verlag, 1995.

[HJJ+95] J. G. Henriksen, J. Jensen, M. Jorgensen, N. Klarlund, R. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems, First International Workshop*, volume 1019 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[HJL96] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, July 1996.

[HK76] S. L. Hantler and J. C. King. An introduction to proving the correctness of programs. *ACM Computing Surveys*, 8(3):331–353, September 1976.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[Hol91] G. J. Holzmann. *Design and validation of computer protocols*. Prentice Hall, New Jersey, 1991.

[Hol97] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[HR98] T. A. Henzinger and V. Rusu. Reachability verification for hybrid systems. In *Proceedings of the International Workshop on Hybrid Systems: Computation and Control*, volume 1386 of *Lecture Notes in Computer Science*, pages 190–204. Springer-Verlag, April 1998.

[HRP94] N. Halbwachs, P. Raymond, and Y. Proy. Verification of linear hybrid systems by means of convex approximations. In B. LeCharlier, editor, *Proceedings of International Symposium on Static Analysis*, volume 864 of *Lecture Notes in Computer Science*. Springer-Verlag, September 1994.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to automata theory, languages, and computation.* Addison-Wesley, Reading, Massachusetts, 1979.

[Kai97] R. Kaivola. Using compositional preorders in the verification of sliding window protocol. In O. Grumberg, editor, *Proceedings of the 9th International Conference on Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 400–411. Springer, June 1997.

[KDG95] P. Kelb, D. Dams, and R. Gerth. Practical symbolic model checking of the full μ-calculus using compositional abstractions. Technical Report 95-31, Department of Computer Science, Eindhoven University of Technology, 1995.

[Kla97] N. Klarlund. Mona & fido: The logic-automaton connection in practice. In *Proceedings of Computer Science Logic*, 1997.

[KM97] M. Kaufmann and J. S. Moore. An industrial strength theorem prover for a logic based on common lisp. *IEEE Transactions on Software Engineering*, 23(4):203–213, April 1997.

[KMM$^+$97] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. In O. Grumberg, editor, *Proceedings of the 9th International Conference on*

Computer Aided Verification, volume 1254 of *Lecture Notes in Computer Science*, pages 400–411. Springer, June 1997.

[KMP+95] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. The omega library interface guide. Technical Report CS-TR-3445, Department of Computer Science, University of Maryland, College Park, March 1995.

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

[KPRS94] W. Kelly, W. Pugh, E. Rosser, and T. Shpeisman. Transitive closure of infinite graphs and its applications. Technical Report CS-TR-3457, UMIACS-TR-95-48, Department of Computer Science, University of Maryland, College Park, April 1994.

[Lio96] J. L. Lions. Ariane 5 flight 501 failure: Report by the inquiry board. Technical report, European Space Agency, July 1996.

[McM93] K. L. McMillan. *Symbolic model checking.* Kluwer Academic Publishers, Massachusetts, 1993.

[Nau66] P. Naur. Proof of algorithms by general snapshots. *BIT*, 6:310–316, 1966.

[Opp78] D. C. Oppen. A $2^{2^{2^{pn}}}$ upper bound on the complexity of presburger arithmetic. *Journal of Computer and System Sciences*, 16:323–332, 1978.

[ORR+96] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In

Proceedings of the 8th International Conference on Computer Aided Verification, volume 1102 of *Lecture Notes in Computer Science*. Springer-Verlag, August 1996.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In *Proceedings of the 11th International Conference on Automated Deduction*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer-Verlag, June 1992.

[Pnu77] A. Pnueli. The temporal semantics of concurrent programs. In *Proceedings of the 18th Symposium on Foundations of Computer Science*, 1977.

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

[Pug92] W. Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 8:102–104, August 1992.

[PW94] W. Pugh and D. Wonnacott. Static analysis of upper and lower bounds on dependences and parallelism. *ACM Transactions on Programming Languages and Systems*, 14(3):1248–1278, July 1994.

[QS82] J. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the Fifth International Symposium in Programming*, 1982.

[Sch98] D. A. Schmidt. Data flow analysis is model checking of abstract interpretations. In *Proceedings of the 24th Annual ACM Symposium on Principles of Programming Languages*, pages 38–48, 1998.

[Sha93] A. U. Shankar. An introduction to assertional reasoning for concurrent systems. *ACM Computing Surveys*, 25(3):225–262, September 1993.

[SKR98] T. R. Shiple, J. H. Kukula, and R. K. Ranjan. A comparison of presburger engines for EFSM reachability. In *Proceedings of the 10th International Conference on Computer Aided Verification*, 1998.

[Ste76] N. V. Stenning. A data transfer protocol. *Computer Networks*, 1:99–110, 1976.

[Tar55] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[VW86] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of First IEEE Symposium on Logic in Computer Science*, pages 322–331, 1986.

[WB95] P. Wolper and B. Boigelot. An automata-theoretic approach to presburger arithmetic constraints. In *Proceedings of the Static Analysis Symposium*, September 1995.

[Wes78] C. H. West. Generalized technique for communication protocol validation. *IBM Journal of Research and Development*, 22:393–404, 1978.

[Wil97] D. Wilner. Vx-files: What really happened on mars (keynote address). In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1997.

[Won95] D. G. Wonnacott. *Constraint-Based Array Dependence Analysis.* PhD thesis, University of Maryland, College Park, August 1995.

[WVF95] J. M. Wing and M. Vaziri-Farahani. Model checking software systems: A case study. In *Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 128–139, 1995.