A Counterexample Guided Abstraction Refinement Framework for Verifying Concurrent C Programs

Sagar J. Chaki

CMU-CS-05-102
May 24, 2005

School of Computer Science
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

This research was sponsored by the Department of Defense under Naval Research Laboratory grant no. N000140110796, the National Science Foundation under grant nos. CCR-0098072 and CCR-9803774, and the Microsoft Corporation through a graduate fellowship.

Keywords: Software verification, Concurrency, Compositionality, C programs, Process algebra, Simulation, State/Event formalism, Deadlock, Temporal logic, Model checking, Predicate abstraction, Abstraction refinement, Computers, Science.

মা, বাবা, বিটু ও মামনের জন্য ...

Abstract

This dissertation presents a framework for verifying concurrent message-passing C programs in an automated manner. The methodology relies on several key ideas. **First**, programs are modeled as finite state machines whose states are labeled with data and whose transitions are labeled with events. We refer to such state machines as labeled Kripke structures (LKSs). Our state/event-based approach enables us to *succinctly express* and *efficiently verify* properties which involve simultaneously both the *static* (data-based) and the *dynamic* (reactive or event-based) aspects of any software system. **Second**, the framework supports a wide range of specification mechanisms and notions of conformance. For instance, *complete* system specifications can be expressed as LKSs and simulation conformance verified between such specifications and any C implementation. For *partial* specifications, the framework supports (in addition to LKSs) a state/event-based linear temporal logic capable of expressing complex safety as well as *liveness* properties. Finally, the framework enables us to check for *deadlocks* in concurrent message-passing programs. **Third**, for each notion of conformance, we present a completely automated and *compositional* verification procedure based on the counterexample guided abstraction refinement (CEGAR) paradigm. Like other CEGAR-based approaches, these verification procedures consist of an iterative application of model construction, model checking, counterexample validation and model refinement steps. However, they are uniquely distinguished by their *compositionality*. More precisely, in each of our conformance checking procedures, the algorithms for model construction, counterexample validation and model refinement are applied *component-wise*. The state-space size of the models are controlled via a two-pronged strategy: (i) using two complementary abstraction techniques based on the static (predicate abstraction) and dynamic (action-guided abstraction) aspects of the program, and (ii) minimizing the number of predicates required for predicate abstraction. The proposed framework has been implemented in the MAGIC tool. We present experimental evaluation in support of the effectiveness of our framework in verifying non-trivial concurrent C programs against a rich class of specifications in an automated manner.

Contents

List of Figures

List of Algorithms

List of Tables

Glossary of Terms

Note: Several symbols are used in multiple contexts and their meaning depends on both the symbol and the type of the subscript. For instance, the symbol S is used to denote the set of states of labeled Kripke structures, Büchi automata, as well as Kripke structures. Therefore, S_M denotes the set of states of a labeled Kripke structure or Kripke structure M, while S_B denotes the set of states of a Büchi automaton B. Similarly, AP_M denotes the set of atomic propositions of a labeled Kripke structure M, while AP_γ denotes the set of atomic propositions specified by a context γ. Some other symbols with multiple connotations are $Init$, Σ, L, T (which are used for both labeled Kripke structures and Büchi automata) and $\downharpoonright$ (which is used for both labeled Kripke structures and traces).

Chapter 2 : Preliminaries

AP	denumerable set of atomic propositions.
S_M	set of states of LKS M.
$Init_M$	set of initial states of LKS M.
AP_M	set of atomic propositions of LKS M.
L_M	propositional labeling of the states of LKS M.
Σ_M	alphabet of LKS M.
T_M	transition relation of LKS M.
$s \xrightarrow{\alpha}_M s'$	transition in LKS M from state s to s' labeled by α.
$Succ_M(s, \alpha)$	successors of state s of LKS M on action α.
$PSucc_M(s, \alpha, P)$	successors of state s of LKS M on action α labeled with set of propositions P.
$ObsAct \quad SilAct$	set of observable and silent actions respectively.
$\mathcal{LKS}$	set of LKSs.

$\preccurlyeq \quad \precsim$	simulation and weak simulation preorder over $\mathcal{LKS}$ respectively.

Chapter 3 : C Programs

$Var \quad Field$	set of C variables and fields respectively.
$\mathbb{D}$	domain of C variable and address values.
Type	set of types of C variables and structures.
$Address$	address, a mapping from variables and structure fields to addresses.
Store	set of all stores. a store is a mapping from addresses to values.
$Expr \quad LValue$	set of C expressions and lvalues respectively.
$Add(\sigma, e)$	address of lvalue e under store σ.
$Val(\sigma, e)$	value of expression e under store σ.
$\sigma \models e$	store σ satisfies expression e, i.e., $Val(\sigma, e) \neq 0$.
$Concrete$	concretization bijection between propositions and expressions.
$\mathbb{T}$	set of statement types $\{\mathsf{ASGN}, \mathsf{CALL}, \mathsf{BRAN}, \mathsf{EXIT}\}$.
$Stmt_{\mathcal{C}}$	set of statements of component $\mathcal{C}$.
$Type_{\mathcal{C}}$	map from statements of component $\mathcal{C}$ to set of types $\mathbb{T}$.
$entry_{\mathcal{C}}$	initial or entry statement of component $\mathcal{C}$.
$Cond_{\mathcal{C}}$	map from branch statements of component $\mathcal{C}$ to branch conditions.
$LHS_{\mathcal{C}}$	maps assignment statements of component $\mathcal{C}$ to left-hand-sides.
$RHS_{\mathcal{C}}$	maps assignment statements of component $\mathcal{C}$ to right-hand-sides.
$Then_{\mathcal{C}}$	map from statements of component $\mathcal{C}$ to **then**-successors.
$Else_{\mathcal{C}}$	map from statements of component $\mathcal{C}$ to **else**-successors.
$InitCond_{\gamma}$	expression qualifying initial states specified by context γ.
AP_{γ}	set of propositions specified by context γ.
Σ_{γ}	alphabet specified by context γ.

$Silent_\gamma$	silent action specified by context γ.
FSM_γ	map from call statements to EFSMs specified by context γ.
$[\![\mathcal{C}]\!]_\gamma$	semantic LKS of component $\mathcal{C}$ under context γ.
$Restrict(\mathcal{S}, P)$	set of states in $\mathcal{S}$ labeled by set of propositions P.
$PreImage(\mathcal{S}, \alpha)$	set of predecessors of states in $\mathcal{S}$ on action α.
$[\![\mathcal{P}]\!]_\Gamma$	semantics of program $\mathcal{P}$ under program context Γ.

Chapter 4 : Abstraction

$\mathbb{B}$	set of Boolean values $\{\text{TRUE}, \text{FALSE}\}$.
$PropVal(AP)$	set of valuations of the set of propositions AP.
$Concrete(V)$	expression for the concretization of valuation V.
$V \Vdash e$	valuation V and expression e are admissible.
$V \Vdash V'$	valuations V and V' are admissible.
$\mathcal{WP}[a](e)$	weakest precondition of expression e with respect to assignment a.
$[\![\mathcal{C}]\!]_\gamma^\Pi$	predicate abstraction of component $\mathcal{C}$ under context γ and with respect to predicate mapping Π.
$[\![\mathcal{P}]\!]_\Gamma^\Pi$	predicate abstraction of program $\mathcal{P}$ under program context Γ and with respect to program predicate mapping Π.
$\mathcal{B}_\mathcal{C}$	set of branch statements of component $\mathcal{C}$.

Chapter 5 : Simulation

$\mathbf{Game}(s_{Im}, s_{Sp})$	simulation game with (s_{Im}, s_{Sp}) as the initial position.
Counterexample Tree Counterexample Witness	counterexample tree and witness LKS respectively for simulation.
Pos	set of all simulation game positions.
$Response(c)$	set of game positions that can result after the specification has responded to challenge c.
$Child(n)$	set of of children of Counterexample Tree node n.
$\overline{WP}(s_{Im}, s_{Sp})$	(s_{Im}, s_{Sp}) is not a winning position in a simulation game.

$\mathcal{HG}$	hypergraph corresponding to N-HORNSAT formula ϕ.
$\mathcal{N}_b$	node of $\mathcal{HG}$ corresponding to Boolean variable b.
$\mathcal{N}_{\text{TRUE}}$ $\mathcal{N}_{\text{FALSE}}$	special nodes of $\mathcal{HG}$ corresponding to TRUE and FALSE respectively.

Chapter 6 : Refinement

$M \downharpoonleft \Sigma'$	projection of LKS M on alphabet Σ'.

Chapter 8 : State-Event Temporal Logic

KS BA	Kripke Structure and Büchi Automaton respectively.
$\models$	entailment between a path, KS or LKS and SE-LTL or LTL formula.
S_M S_B	set of states of KS M and BA B respectively.
$Init_M$ $Init_B$	set of initial states of KS M and BA B respectively.
AP_M AP_B	set of atomic propositions of KS M and BA B respectively.
L_M L_B	propositional labeling of the states of KS M and BA B respectively.
T_M T_B	transition relation of KS M and BA B respectively.
Acc_B	set of accepting states of BA B respectively.
$M \times B$	standard product of KS M and BA B.
$M \otimes B$	state/event product of LKS M and BA B.
Lasso	lasso-shaped counterexample to an SE-LTL formula.

Chapter 9 : Two-Level Abstraction Refinement

$[s]^R$	equivalence class of state s induced by equivalence relation R.
M^R	quotient LKS induced by LKS M and equivalence relation R.
$\widehat{Succ}(s, \alpha)$	set of abstract successors of state s under action α.
$Split(M, R, [s]^R, A)$	refined equivalence relation obtained from equivalence relation R by splitting the equivalence class $[s]^R$.

Chapter 10 : Deadlock

$Ref(s)$	refusal of state s, i.e., set of actions refused by s.
$Fail(M)$	set of all failures of LKS M.
$\theta \downharpoonright i$	projection of trace θ on alphabet of LKS M_i.
$\widehat{Ref}(\alpha)$	abstract refusal of state α.
$AbsFail(\widehat{M})$	set of all abstract failures of LKS $\widehat{M}$.

Chapter 1

Introduction

The ability to reason about the correctness of programs is no longer a subject of primarily academic interest. With each passing day the complexity of software artifacts being produced and employed is increasing dramatically. There is hardly any aspect of our day-to-day lives where software agents do not play an often silent yet crucial role. The fact that many of such roles are safety-critical mandates that these software artifacts be validated rigorously before deployment. So far, however, this goal has largely eluded us.

In this chapter we will first layout the problem space which is of concern to this thesis, viz., automated formal verification of concurrent programs. We will present the core issues and problems, as well as the major paradigms and techniques that have emerged in our search for effective solutions. We will highlight the important hurdles that remain to be scaled. The later portion of this chapter presents an overview of the major techniques proposed by this thesis to surmount these hurdles. The chapter ends with a summary of the core contributions of this dissertation.

1.1 Software Complexity

Several factors hinder our ability to reason about non-trivial concurrent programs in an automated manner. First, the sheer complexity of software. Binaries obtained from hundreds of thousands of lines of source code are routinely executed. The source code is written in languages ranging from C/C++/Java to ML/Ocaml. These languages differ not only in their flavor (imperative, functional) but also in their constructs (procedures, objects, pattern-matching, dynamic memory allocation, garbage collection), semantics (loose, rigorous) and so on.

This *sequential* complexity is but one face of the coin. Matters are further exacerbated by what can be called *parallel* complexity. State of the art software agents rarely operate in isolation. Usually they communicate and cooperate with other agents while performing their tasks. With the advent of the Internet, and the advance in networking technology, the scope of such communication could range from multiple threads communicating via shared memory on the same computer to servers and clients communicating via SSL channels across the Atlantic. Verifying the correctness of such complex behavior is a daunting challenge.

1.2 Software Development

Another, much less visible yet important, factor is the development process employed in the production of most software and the role played by validation and testing methodologies in such processes. A typical instance of a software development cycle consists of five phases - (i) requirement specification, (ii) design, (iii) design validation, (iv) implementation and (v) implementation validation. The idea is that defects found in the design (in phase iii) are used to improve the design and those found in the implementation (in phase v) are used to improve the implementation. The cycle is repeated until each stage concludes successfully.

Usually the design is described using a formal notation like UML. The dynamic behavior is often described using Statecharts (or some variant of it). The design validation is done by some exhaustive technique (like model checking). However, what matters in the final tally is not so much the correctness of the design but rather the correctness of the implementation. Nevertheless, in reality, verification of the implementation is done much less rigorously. This makes it imperative that we focus more on developing techniques that enable us to verify actual code that will be compiled and executed. A major fraction of such code has been written, continues to be written and, in my opinion, will continue to be written in C.

Present day code validation falls in two broad categories - testing and formal verification. The merits and demerits of testing [88] are well-known and thus it is unnecessary to dwell on them in detail here. It suffices to mention that the necessity of being *certain* about the correctness of a piece of code precludes exclusive reliance on testing as the validation methodology, and forces us to adopt more formal approaches.

1.3 Software Verification

State of the art formal *software verification* is an extremely amorphous entity. Originally, most approaches in this field could be categorized as belonging to either of two schools of thought: theorem proving and model checking. In theorem proving (or deductive verification [70]), one typically attempts to construct a formula ϕ (in some suitable logic like higher-order predicate calculus) that represents both the system to be verified and the correctness property to be established. The validity of ϕ is then established using a theorem prover. As can be imagined, deductive verification is extremely powerful and can be used to verify virtually any system (including *infinite* state systems) and property. The flip-side is that it involves a lot of manual effort. Furthermore it yields practically no diagnostic feedback that can be used for debugging if ϕ is found to be invalid.

1.4 Model Checking

Where theorem proving fails, model checking [39] shines. In this approach, the system to be verified is represented by a *finite* state transition system $\mathcal{M}$ (often a Kripke structure) and the property to be established is expressed as a temporal logic [81] (usually CTL [32] with fairness or LTL [78]) formula ϕ. The model checking problem is then to decide whether $\mathcal{M}$ is a model of ϕ. Not only can this process be automated to a large degree, it also yields extremely useful diagnostic feedback (often in the form of counterexamples) if $\mathcal{M}$ is found not to model ϕ. Owing to these and other factors, the last couple of decades have witnessed the emergence of model checking as the eminent formal verification technique. Various kinds of temporal logics have been extensively studied [59] and efficient model checking algorithms have been designed [35, 99]. The development of techniques like *symbolic* model checking [21], *bounded* model checking [11, 12], *compositional* reasoning [34] and *abstraction* [36] have enabled us to verify systems with enormous state spaces [22].

One of the original motivations behind the development of model checking was to extract and verify *synchronization skeletons* of concurrent programs, a typical software verification challenge. Somewhat ironically, the meteoric rise of model checking to fame was largely propelled by its tremendous impact on the field of hardware verification. I believe that a major factor behind this phenomenon is that model checking can only be used if a finite model of the system is available. Also since real system descriptions are often quite large, the models must be extracted automatically or at least semi-automatically. While this process is often straightforward for hardware, it is much more involved for software. Typically software systems have *infinite* state spaces. Thus, extracting a finite model often involves a process of abstraction as well.

1.5 Predicate Abstraction

For a long time, the applicability of model checking to software was somewhat handicapped by the absence of powerful automated model extraction techniques. This scenario changed with the advent of predicate abstraction [63] (a related notion called data type abstraction used by systems like Bandera [8, 58] can be viewed as a special instance of predicate abstraction). Even though predicate abstraction was quickly picked up for research in hardware verification as well [49, 50], its effect on code verification was rather dramatic. It forms the backbone of two of the major code verifiers in existence, SLAM [6, 107] and BLAST [13, 66].

Predicates abstraction is parameterized by a set of predicates involving the variables of the concrete system description. It also involves non-trivial use of theorem provers (in fact the its original use [63] was to create abstract state transition graphs using the theorem prover PVS). Thus it has triggered a more subtle effect - it has caused the boundary between model checking and theorem proving to become less distinct.

Challenge 1 *Predicate abstraction essentially works by aggregating system states* *that are similar in terms of their data valuations. It is insensitive to the events that a system can perform from a given state. Can we develop other notions of abstraction that leverage the similarities between system states in terms of their dynamic (event-based) behavior? Such abstractions would complement predicate abstraction and lead to further reduction of state-space size.*

1.6 Abstraction Refinement

Even with progress in automated model extraction techniques, verifying large software systems remains an extremely tedious task. A major obstacle is created by the abstraction that happens during model extraction. Abstraction usually introduces additional behavior that is absent in the concrete system. Suppose that the model check fails and the model checker returns a counterexample CE. This does not automatically indicate a bug in the system because it is entirely possible that CE is an additional behavior introduced by abstraction (such a CE is often called a *spurious* counterexample). Thus we need to verify whether CE is spurious, and if so we need to refine our model so that it no longer allows CE as an admissible behavior. This process is called abstraction refinement. Since the extracted models and counterexamples generated are quite large, abstraction refinement must be automated (or at least semi-automated) to be practically effective.

The above requirements lead naturally to the paradigm called counterexample guided abstraction refinement (CEGAR). In this approach, the entire verification process is captured by a three step *abstract-verify-refine* loop. The actual details of each step depend on the kind of abstraction and refinement methods being used. The steps are described below in the context of predicate abstraction, where $Pred$ denotes the set of predicates being used for the abstraction.

1. **Step 1 : Model Creation.** Extract a finite model from the code using predicate abstraction with $Pred$ and go to step 2.

2. **Step 2 : Verification.** Check whether the model satisfies the desired property. If this is the case, the verification successfully terminates; otherwise, extract a counterexample CE and go to step 3.

3. **Step 3 : Refinement.** Check if CE is spurious. If not we have an actual bug and the verification terminates unsuccessfully. Otherwise we improve $Pred$ and go to step 1. Let us refer to the improved $Pred$ as $\overline{Pred}$. Then $\overline{Pred}$ should be such that CE and all previous spurious counterexamples will be eliminated if the model is extracted using $\overline{Pred}$.

Challenge 2 *Software model checking has focused almost exclusively on the verification of safety properties via some form of trace containment. It would be desirable to extend its applicability to more general notions of conformance such as simulation and richer class of specifications such as liveness.*

Challenge 3 *The complexity of predicate abstraction is exponential in the number of predicates used. The naive abstraction refinement approach keeps on adding new predicates on the basis of spurious counterexamples. Previously added predicates are not removed even if they have been rendered redundant by predicated discovered subsequently. Can we improve this situation?*

1.7 Compositional Reasoning

CEGAR coupled with predicate abstraction has become an extremely popular approach toward the automated verification of sequential software, especially C programs [13] such as device drivers [107]. However, considerably less research has been devoted to-wards the application of these techniques for verifying concurrent programs.

Compositional reasoning has long been recognized as one of the most potent solutions to the state-space explosion which plagues the analysis of concurrent systems. Compositionality appears explicitly in the theory of process algebras such as CSP [69], CCS [85] and the π-Calculus [86]. A wide variety of process algebraic formalisms have been developed with the intention of modeling concurrent systems and it is therefore natural [9] to investigate whether process algebraic concepts are useful in the verification domain as well.

One of the key concepts arising out of the process algebraic research is the need to focus on communication [85] when reasoning about concurrent systems. For instance CSP advocates the use of *shared actions* as the principal communication mechanism between concurrent components of a system. Moreover, shared action communication can model message-passing C programs such as client-server systems and web-services in a very natural manner.

Challenge 4 *The CEGAR paradigm has been used with considerable success on sequential programs. Can we also use it to compositionally verify concurrent programs? What, if any, are the restrictions that we might need to impose in order to achieve this goal?*

1.8 State/event based Analysis

A major difficulty in applying model checking for practical software verification lies in the modeling and specification of meaningful properties. The most common instantiations of model checking to date have focused on finite-state models and either branching-time (CTL [32]) or linear-time (LTL [78]) temporal logics. To apply model checking to software, it is necessary to specify (often complex) properties on the finite-state abstracted models of computer programs. The difficulties in doing so are even more pronounced when reasoning about *modular* software, such as concurrent or component-based sequential programs. Indeed, in modular programs, communication among modules proceeds via actions (or events), which can represent function calls, requests and acknowledgments, etc. Moreover, such communication is commonly data-dependent. Software behavioral claims, therefore, are often specifications defined over combinations of program actions and data valuations.

Existing modeling techniques usually represent finite-state machines as finite annotated directed graphs, using either *state-based* or *event-based* formalisms. It is well-known that the two frameworks are interchangeable. For instance, an action can be encoded as a change in state variables, and likewise one can equip a state with different actions to reflect different values of its internal variables. However, converting from one representation to the other often leads to a significant enlargement of the state space. Moreover, neither approach on its own is practical when it comes to modular software, in which actions are often data-dependent: considerable domain expertise is then required to annotate the program and to specify proper claims.

Challenge 5 *Can we develop a formalism for succinctly expressing and efficiently* *verifying state/event-based properties of programs? In particular we should be able to verify a state/event system directly without having to translate it to an equivalent pure-state or pure-event version. Further, can we combine state/event-based analysis with a compositional CEGAR scheme?*

1.9 Deadlock Detection

Ensuring that standard software components are assembled in a way that guarantees the delivery of reliable services is an important task for system designers. Certifying the absence of deadlock in a composite system is an example of a stringent requirement that has to be satisfied before the system can be deployed in real life. This is especially true for safety-critical systems, such as embedded systems or plant controllers, that are expected to always service requests within a fixed time limit or be responsive to external stimuli.

In addition, many formal analysis techniques, such as temporal logic model checking [32, 39], assume that the systems being analyzed are deadlock-free. In order for the results of such analysis to be valid, one usually needs to establish deadlock freedom separately. Last but not least, in case a deadlock is detected, it is highly desirable to be able to provide system designers and implementers with appropriate diagnostic feedback.

However, despite significant efforts, validating the absence of deadlock in systems of realistic complexity remains a major challenge. The problem is especially acute in the context of concurrent programs that communicate via mechanisms with blocking semantics, e.g., synchronous message-passing and semaphores. The primary obstacle is the well-known *state space explosion* problem whereby the size of the state space of a concurrent system increases exponentially with the number of components.

As mentioned before, two paradigms are usually recognized as being the most effective against the state space explosion problem: *abstraction* and *compositional reasoning*. Even though these two approaches have been widely studied in the context of formal verification [36, 64, 67, 84], they find much less use in deadlock detection. This is possibly a consequence of the fact that deadlock is inherently non-compositional and its absence is not preserved by standard abstractions. Intuitively, the fundamental problem here is that deadlock is an *existential* safety property. Therefore, a compositional CEGAR scheme for deadlock detection would be especially significant.

Challenge 6 *In the light of the above discussion, can we develop a compositional CEGAR-based procedure for deadlock detection?*

1.10 Summary

This dissertation presents a framework for verifying concurrent message-passing C programs with specific emphasis on addressing the challenges enumerated earlier in this chapter. Among other things, we addresses Challenge 5 by enabling both state-based and action-based properties to be expressed, combined, and efficiently verified. To this end we propose the use of *labeled Kripke structures* (LKSs) as the modeling formalism. In essence, an LKS is a finite state machines in which states are labeled with atomic propositions and transitions are labeled with events (or actions). In the rest of this chapter we will refer to a concurrent message-passing C program as simply a program.

Our state/event-based modeling methodology is described in two stages. We first present a semantics of programs in terms of LKSs (cf. Chapter 3). We then develop a generalized form of predicate abstraction to construct conservative LKS abstractions from programs (cf. Chapter 4) in an automated manner. We provide formal justification for our claim that the extracted LKS models are indeed conservative abstractions of the concrete programs from which they have been constructed.

Subsequently we address Challenge 2 and Challenge 4 by presenting a compositional CEGAR procedure for verifying simulation conformance between a program and an LKS specification. We define the notion of witness LKSs as counterexamples to simulation conformance and present algorithms for efficiently constructing such counterexamples upon the failure of a simulation check (cf. Chapter 5). We next present algorithms for checking the validity of witness LKSs and refining the LKS models if the witness is found to be spurious (cf. Chapter 6). The entire CEGAR procedure is compositional in the sense that the model construction, witness validation and abstraction refinement are performed component-wise. Note that we do not delve into the compositional nature of the model checking step. Compositional model checking has been the focus of considerable research and we hope to leverage the significant breakthroughs that have emerged from this effort.

Moving on, we propose the use of predicate minimization as a solution to Challenge 3 (cf. Chapter 7). Our approach uses pseudo-Boolean constraints to minimize the number of predicates used for predicate abstraction and thus eliminates redundant predicates as new ones are discovered. We also present an action-guided abstraction refinement scheme to address Challenge 1 (cf. Chapter 9). This abstraction works by aggregating states based on the events they can perform and complements predicate abstraction naturally. Both these solutions are seamlessly integrated with the compositional CEGAR scheme presented earlier.

In Chapter 8 we present the logic SE-LTL, a *state/event* derivative of the standard linear temporal logic LTL. We present efficient SE-LTL model checking algorithms to help reason about state/event-based systems. We also present a compositional CEGAR procedure [23, 26, 37] for the automated verification of concurrent C programs against SE-LTL specifications. SE-LTL enriches our specification mechanism by allowing state/event-based liveness properties and is thus relevant to both Challenge 2 and Challenge 5.

Finally, in Chapter 10 we address Challenge 6 by presenting a compositional CEGAR scheme to perform deadlock detection on concurrent message-passing programs [27]. In summary, the demand for better formal techniques to verify concurrent and distributed C programs is currently overwhelming. This dissertation identifies some notable stumbling blocks in this endeavor and provides a road map to their solution.

Chapter 2

Preliminaries

In this chapter we present preliminary notations and definitions that will be used in the rest of the thesis. We assume a denumerable set of atomic propositions $\mathbf{AP}$.

Definition 1 (Labeled Kripke Structure) *A Labeled Kripke Structure (LKS) is a 6-tuple* $(S, Init, AP, L, \Sigma, T)$ *where: (i)* S *is a non-empty set of states, (ii)* $Init \subseteq S$ *is a set of initial states, (iii)* $AP \subseteq \mathbf{AP}$ *is a finite set of atomic propositions, (iv)* $L : S \to 2^{AP}$ *is a propositional labeling function that maps every state to a set of atomic propositions that are true in that state, (v)* Σ *is a set of actions, also known as the alphabet, and (vi)* $T \subseteq S \times \Sigma \times S$ *is a transition relation.*

Important note: In the rest of this thesis we will write $Field_{Tup}$ to mean the field $Field$ of a tuple Tup. Thus, for any LKS $M = (S, Init, AP, L, \Sigma, T)$, we will write S_M, $Init_M$, AP_M, L_M, Σ_M and T_M to mean S, $Init$, AP, L, Σ and T respectively. Also we will write $s \xrightarrow{\alpha}_M s'$ to mean $(s, \alpha, s') \in T_M$. When M is clear from the context we will write simply $s \xrightarrow{\alpha} s'$.

Example 1 *Figure 2.1 shows a simple LKS* $M = (S, Init, AP, L, \Sigma, T)$ *with five* *states* $\{1, 2, 3, 4, 5\}$*. The alphabet* Σ *is* $\{\alpha, \beta, \chi, \delta\}$ *and the set of atomic propositions is* $AP = \{p, q, r\}$*. Transitions are shown as arrows labeled with actions. The initial state* 1 *is indicated by an incoming transition with no source state. The propositional labellings are shown beside the respective states.*

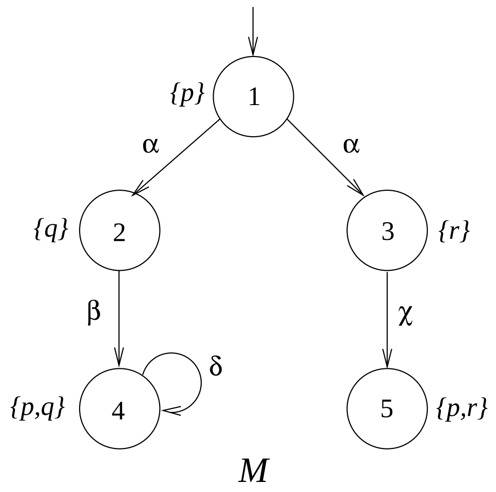


Figure 2.1: A simple LKS.

Intuitively, an LKS can model the behavior of a system in terms of both states and events. We denote the set of all LKSs by $\mathcal{LKS}$. The successor function $Succ$ maps a state s and an action α to the set of α-successors of s. Additionally, the propositional successor function $PSucc$ maps a state s, an action α and a set of atomic propositions P to the (possibly empty) set of α-successors of s that are labeled with P. We now present these two functions formally.

Definition 2 (Successor Functions) *Let* $M = (S, Init, AP, L, \Sigma, T)$ *be an LKS. The successor functions* $Succ : S \times \Sigma \to 2^S$ *and* $PSucc : S \times \Sigma \times 2^{AP} \to 2^S$ *are defined as follows:*

$$Succ(s, \alpha) = \{s' \in S \mid s \xrightarrow{\alpha} s'\}$$

$$PSucc(s, \alpha, P) = \{s' \in S \mid s \xrightarrow{\alpha} s' \wedge L(s') = P\}$$

Example 2 *For the LKS M shown in Figure 2.1, we have the following:*

- $Succ(1, \alpha) = \{2, 3\}$, $Succ(1, \beta) = Succ(1, \chi) = \emptyset$.
- $Succ(2, \alpha) = Succ(2, \chi) = \emptyset$, $Succ(2, \beta) = \{4\}$.
- $PSucc(1, \alpha, \{p\}) = \emptyset$, $PSucc(1, \alpha, \{q\}) = \{2\}$, $PSucc(1, \alpha, \{p, q\}) = \emptyset$.
- $PSucc(2, \beta, \{p\}) = \emptyset$, $PSucc(2, \beta, \{q\}) = \emptyset$, $PSucc(2, \beta, \{p, q\}) = \{4\}$.

Note that both the set of states and the alphabet of an LKS can in general be infinite. Also, an LKS can be non-deterministic, i.e., an LKS $M = (S, Init, AP, L, \Sigma, T)$ could have a state $s \in S$ and an action $\alpha \in \Sigma$ such that $|Succ(s, \alpha)| > 1$. However, M is said to have *finite non-determinism* if for any state $s \in S$ and any action $\alpha \in \Sigma$, the set $Succ(s, \alpha)$ is always finite. In the rest of this thesis we will only consider LKSs with finite non-determinism.

Actions are used to model observable or unobservable behaviors of systems. Accordingly, we assume that observable actions are drawn from a denumerable set *ObsAct*, while unobservable (or silent) actions are drawn from a denumerable set *SilAct*. We assume a distinguished action $\tau \in SilAct$.

Definition 3 (Simulation) *Let $M_1 = (S_1, Init_1, AP_1, L_1, \Sigma_1, T_1)$ and $M_2 = (S_2, Init_2, AP_2, L_2, \Sigma_2, T_2)$ be two LKSs such that $AP_1 = AP_2$ and $\Sigma_1 = \Sigma_2$. A relation $\mathcal{R} \subseteq S_1 \times S_2$ is said to be a simulation relation iff it obeys the following two conditions:*

1. $\forall s_1 \in S_1 \centerdot \forall s_2 \in S_2 \centerdot s_1 \mathcal{R} s_2 \implies L_1(s_1) = L_2(s_2)$
2. $\forall s_1 \in S_1 \centerdot \forall s_2 \in S_2 \centerdot \forall \alpha \in \Sigma_1 \centerdot \forall s_1' \in S_1 \centerdot$

$$(s_1 \mathcal{R} s_2 \ \wedge \ s_1 \xrightarrow{\alpha} s_1') \implies \exists s_2' \in S_2 \centerdot s_2 \xrightarrow{\alpha} s_2' \ \wedge \ s_1' \mathcal{R} s_2'$$

We say that M_1 is simulated by M_2, and denote this by $M_1 \preccurlyeq M_2$, iff there exists a simulation relation $\mathcal{R}$ such that the following condition holds:

$$\forall s_1 \in Init_1 \,.\, \exists s_2 \in Init_2 \,.\, s_1 \mathcal{R} s_2$$

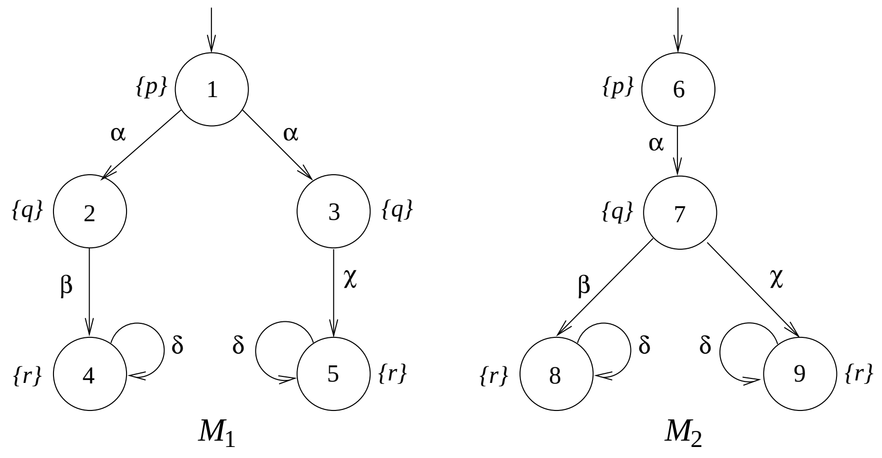


Figure 2.2: Two LKSs demonstrating simulation.

Example 3 *Consider the LKSs M_1 and M_2 shown in Figure 2.2. It is clear that $M_1 \preccurlyeq M_2$ since the the following is a simulation relation that relates the pair of initial states* $(1, 6)$.

$$\mathcal{R} = \{(1,6),(2,7),(3,7),(4,8),(5,9)\}$$

On the other hand $M_2 \not\preccurlyeq M_1$. Intuitively this is because of state 7 of M_2. Note that state 7 can do both actions β and χ, but no state of M_1 can do both β and χ. Hence no state of M_1 can correspond to state 7 in accordance with a simulation relation.

Definition 4 (Weak Simulation) *Let $M_1 = (S_1, Init_1, AP_1, L_1, \Sigma_1, T_1)$ and $M_2 = (S_2, Init_2, AP_2, L_2, \Sigma_2, T_2)$ be two LKSs such that $AP_1 = AP_2$ and $\Sigma_1 = \Sigma_2 \cup \{\tau\}$. A relation $\mathcal{R} \subseteq S_1 \times S_2$ is said to be a weak simulation relation iff it obeys the following three conditions:*

1. $\forall s_1 \in S_1 \,.\, \forall s_2 \in S_2 \,.\, s_1 \mathcal{R} s_2 \implies L_1(s_1) = L_2(s_2)$

2. $\forall s_1 \in S_1 \,.\, \forall s_2 \in S_2 \,.\, \forall s_1' \in S_1 .$

$$(s_1 \mathcal{R} s_2 \;\wedge\; s_1 \xrightarrow{\tau} s_1') \implies s_1' \mathcal{R} s_2 \bigvee \exists s_2' \in S_2 \,.\, s_2 \xrightarrow{\tau} s_2' \;\wedge\; s_1' \mathcal{R} s_2'$$

3. $\forall s_1 \in S_1 \,.\, \forall s_2 \in S_2 \,.\, \forall \alpha \in \Sigma_1 \setminus \{\tau\} \,.\, \forall s_1' \in S_1 .$

$$(s_1 \mathcal{R} s_2 \;\wedge\; s_1 \xrightarrow{\alpha} s_1') \implies \exists s_2' \in S_2 \,.\, s_2 \xrightarrow{\alpha} s_2' \;\wedge\; s_1' \mathcal{R} s_2'$$

Note that if $\tau \notin \Sigma_2$ then condition 2 above is equivalent to the following:

$$\forall s_1 \in S_1 \,.\, \forall s_2 \in S_2 \,.\, \forall s_1' \in S_1 \,.\, (s_1 \mathcal{R} s_2 \;\wedge\; s_1 \xrightarrow{\tau} s_1') \implies s_1' \mathcal{R} s_2$$

We say that M_1 is weakly simulated by M_2, and denote this by $M_1 \precsim M_2$, iff there exists a weak simulation relation $\mathcal{R}$ such that the following condition holds:

$$\forall s_1 \in Init_1 \,.\, \exists s_2 \in Init_2 \,.\, s_1 \mathcal{R} s_2$$

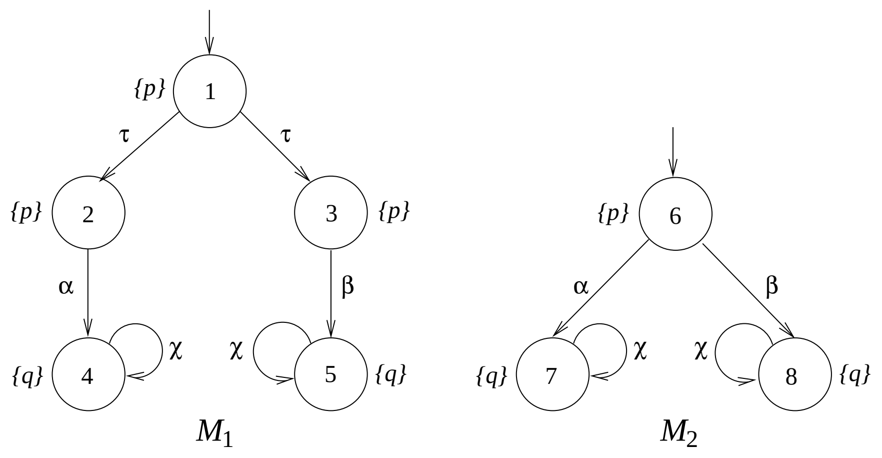


Figure 2.3: Two LKSs demonstrating weak simulation.

Example 4 *Consider the LKSs M_1 and M_2 shown in Figure 2.3. It is clear that $M_1 \precsim M_2$ since the the following is a weak simulation relation that relates the pair of initial states $(1,6)$.*

$$\mathcal{R} = \{(1,6),(2,6),(3,6),(4,7),(5,8)\}$$

On the other hand $M_2 \not\precsim M_1$. Intuitively this is because of state 6 of M_2. Note that state 6 can do both actions α and β, but no state of M_1 can do both α and β. Hence no state of M_1 can correspond to state 6 in accordance with a weak simulation relation.

Note the important difference between simulation and weak simulation. Clearly $M_1 \not\preccurlyeq M_2$ since the initial state 1 of M_1 can do the τ action while the initial state 6 of M_2 cannot. In general, weak simulation allows M_2 to simply ignore τ actions performed by M_1.

Our notion of weak simulation is derived from that of CCS [85] and differs critically from notions of weak simulation presented by others [39]. In particular, our notion of weak simulation does not preserve liveness properties since it is completely insensitive to (even infinite sequences of) τ actions. For example, consider the the two LKSs shown in Figure 2.4. Clearly $M_1 \precsim M_2$. Now consider the liveness property ϕ that "eventually action α occurs". Clearly M_2 satisfies ϕ while M_1 does not. However the non-preservation of liveness properties by our notion of weak simulation will not be a problem since we will use weak simulation only for compositional validation of counterexamples. Further details will be presented in Chapter 6.

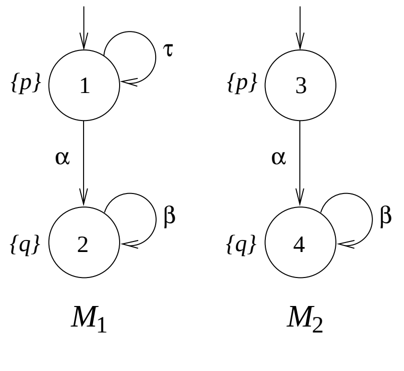


Figure 2.4: Two LKSs showing that weak simulation does not preserve liveness properties.

The following two results about simulation are well-known and will be used crucially in the later part of the thesis.

Theorem 1 (Transitivity) *The following statement is valid:*

$$\forall M_1 \in \mathcal{LKS} \centerdot \forall M_2 \in \mathcal{LKS} \centerdot \forall M_3 \in \mathcal{LKS} \centerdot M_1 \preccurlyeq M_2 \wedge M_2 \preccurlyeq M_3 \implies M_1 \preccurlyeq M_3$$

Proof. Let $M_1 = (S_1, Init_1, AP_1, L_1, \Sigma_1, T_1)$, $M_2 = (S_2, Init_2, AP_2, L_2, \Sigma_2, T_2)$ and $M_3 = (S_3, Init_3, AP_3, L_3, \Sigma_3, T_3)$. Let $\mathcal{R}_{12} \subseteq S_1 \times S_2$ be a simulation relation such that $\forall s_1 \in Init_1 \centerdot \exists s_2 \in Init_2 \centerdot (s_1, s_2) \in \mathcal{R}_{12}$ and $\mathcal{R}_{23} \subseteq S_2 \times S_3$ be a simulation relation such that $\forall s_2 \in Init_2 \centerdot \exists s_3 \in Init_3 \centerdot (s_2, s_3) \in \mathcal{R}_{23}$. Define a relation $\mathcal{R}_{13} \subseteq S_1 \times S_3$ as follows:

$$\mathcal{R}_{13} = \{(s_1, s_3) \mid \exists s_2 \centerdot (s_1, s_2) \in \mathcal{R}_{12} \wedge (s_2, s_3) \in \mathcal{R}_{23}\}$$

Show that $\mathcal{R}_{13}$ is a simulation relation and $\forall s_1 \in Init_1 \centerdot \exists s_3 \in Init_3 \centerdot (s_1, s_3) \in \mathcal{R}_{13}$.

□

Theorem 2 (Witness) *The following statement is valid:*

$$\forall M_1 \in \mathcal{LKS} \centerdot \forall M_2 \in \mathcal{LKS} \centerdot \forall M_3 \in \mathcal{LKS} \centerdot M_1 \preccurlyeq M_2 \wedge M_1 \not\preccurlyeq M_3 \implies M_2 \not\preccurlyeq M_3$$

Proof. This is a direct consequence of Theorem 1.

□

Theorem 1 states that simulation is a transitive relation on $\mathcal{LKS}$. Theorem 2 provides a notion of a *witness* to the absence of simulation between two LKSs. Essentially it states that an LKS M_1 is a witness to $M_2 \not\preccurlyeq M_3$ iff M_1 is simulated by M_2 but not simulated by M_3. We will use a witness LKS as a counterexample to simulation later in this thesis.

Definition 5 (Parallel Composition) *Let* $M_1 = (S_1, Init_1, AP_1, L_1, \Sigma_1, T_1)$ *and* $M_2 = (S_2, Init_2, AP_2, L_2, \Sigma_2, T_2)$ *be two LKSs such that* $AP_1 \cap AP_2 = \emptyset$*. Then the parallel composition of* M_1 *and* M_2*, denoted by* $M_1 \parallel M_2$*, is an LKS* $\big(S_\parallel, Init_\parallel, AP_\parallel, L_\parallel, \Sigma_\parallel, T_\parallel\big)$ *such that: (i)* $S_\parallel = S_1 \times S_2$*, (ii)* $Init_\parallel = Init_1 \times Init_2$*, (iii)* $AP_\parallel = AP_1 \cup AP_2$*, (iv)* $\Sigma_\parallel = \Sigma_1 \cup \Sigma_2$*, and the state labeling function and transition relation are defined as follows:*

- $\forall s_1 \in S_1 \centerdot \forall s_2 \in S_2 \centerdot L_\parallel(s_1, s_2) = L_1(s_1) \cup L_2(s_2)$.

- $\forall s_1 \in S_1 \centerdot \forall s_2 \in S_2 \centerdot \forall \alpha \notin \Sigma_2 \centerdot \forall s_1' \in S_1 \centerdot$
$$s_1 \xrightarrow{\alpha} s_1' \implies (s_1, s_2) \xrightarrow{\alpha} (s_1', s_2)$$

- $\forall s_1 \in S_1 \centerdot \forall s_2 \in S_2 \centerdot \forall \alpha \notin \Sigma_1 \centerdot \forall s_2' \in S_2 \centerdot$
$$s_2 \xrightarrow{\alpha} s_2' \implies (s_1, s_2) \xrightarrow{\alpha} (s_1, s_2')$$

- $\forall s_1 \in S_1 \centerdot \forall s_2 \in S_2 \centerdot \forall \alpha \in \Sigma_1 \cap \Sigma_2 \centerdot \forall s_1' \in S_1 \centerdot \forall s_2' \in S_2 \centerdot$
$$s_1 \xrightarrow{\alpha} s_1' \wedge s_2 \xrightarrow{\alpha} s_2' \implies (s_1, s_2) \xrightarrow{\alpha} (s_1', s_2')$$

Example 5 *Figure 2.5 shows two LKSs* M_1 *and* M_2 *and their parallel composition* $M_1 \parallel M_2$*. Note that in general parallel composition leads to more states and transitions.*

The above notion of parallel composition is central to our approach. We assume that when several components are executed concurrently, they *synchronize* on *shared actions* and proceed independently on local actions. We will see later that the LKSs we compose will not contain the τ action in their alphabet. Hence we do not need to define parallel composition specially for τ. This notion of parallel composition has been used in, e.g., CSP [68, 69, 100], and by Anantharaman et. al. [3]. Parallel composition is commutative and associative. In addition, both simulation and weak

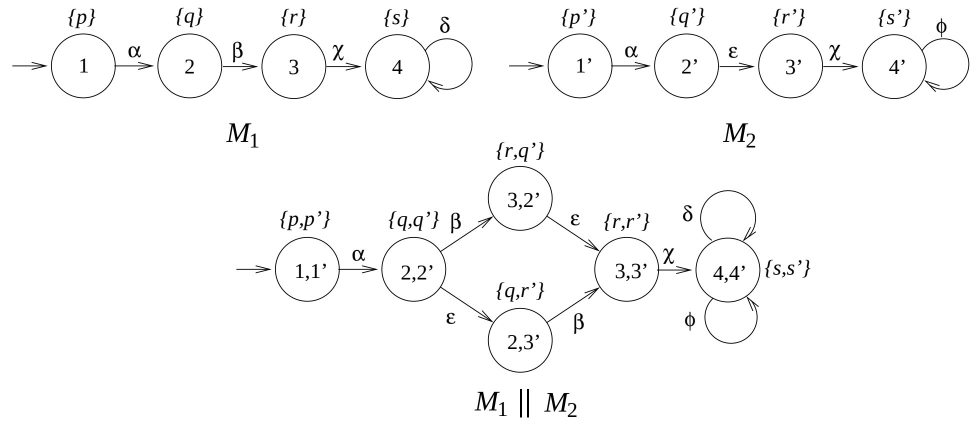


Figure 2.5: Three LKSs demonstrating parallel composition.

simulation are congruences with respect to parallel composition, as stated by the following result.

Theorem 3 *Let M_1, M_1', M_2 and M_2' be LKSs. Then the following holds:*

$$M_1 \preccurlyeq M_1' \wedge M_2 \preccurlyeq M_2' \implies (M_1 \parallel M_2) \preccurlyeq (M_1' \parallel M_2')$$

$$M_1 \precsim M_1' \wedge M_2 \precsim M_2' \implies (M_1 \parallel M_2) \precsim (M_1' \parallel M_2')$$

Chapter 3

C Programs

In this chapter we provide a formalization of a C component and a C program. Our goal is to present formally the syntax and semantics of components and programs. The semantics will be defined in an operational manner using LKSs. The correctness of the rest of this thesis depends critically on this semantics. In particular, this semantics will be used in the next chapter to show that the models we construct from C programs via predicate abstraction are conservative in a very precise sense.

In the rest of this thesis we will write program and component to mean a C program and a C component, respectively. Each component will correspond to a single non-recursive C procedure. This procedure will be obtained by automatically inlining all library routines except the ones used for communication and synchronization with other components. This is always possible if the component is non-recursive and the source code for library routines to be inlined is available. We assume that components of a C program communicate with each other via blocking message-passing. We will also assume several other restrictions on our C programs. We discuss these restrictions further in Section 3.3.

We assume that all program variables are drawn from a denumerable set Var and that structure fields are drawn from a denumerable set $Field$. We will write $Dom(\varphi)$ and $Range(\varphi)$ to denote the domain and range respectively of a function φ. We will write $D \hookrightarrow R$ to denote a partial function from a domain D to a range R.

We will assume that every variable and address value is drawn from a domain $\mathbb{D}$. The only requirement on this domain $\mathbb{D}$ is that the familiar arithmetic, bitwise and logical operators be defined on it. For instance $\mathbb{D}$ could be the set of 32-bit integers. Furthermore, every variable and structure field has a type. The set of types is denoted by Type and consists of a single base type $\mathbb{D}$ and **struct** types. An object of **struct** type is a record containing fields of other types. Therefore, the set of types can be defined by the following BNF grammar:

$$\mathsf{Type} := \mathbb{D} \mid \mathbf{struct}(Field \times \mathsf{Type})^{+}$$

We denote the type of any variable v by $\mathsf{Type}(v)$. We assume an *injective* address function $Address$ whose domain is $Var \cup (\mathbb{D} \times Field)$, whose range is $\mathbb{D}$, and which obeys the following additional constraints:

- Let v be any variable such that $\mathsf{Type}(v) = \mathbf{struct}\langle (f_1, t_1), \ldots, (f_k, t_k)\rangle$. Then:

  $$\forall 1 \leq i \leq k \centerdot (Address(v), f_i) \in Dom(Address)$$

  This ensures that for any structure variable v with a field f, $Address$ assigns an address to the location $v.f$.

- Let f be any field such that (i) $\exists z \in \mathbb{D} \centerdot (z, f) \in Dom(Address)$ and (ii) $\mathsf{Type}(f) = \mathbf{struct}\langle (f_1, t_1), \ldots, (f_k, t_k)\rangle$. Then:

  $$\forall 1 \leq i \leq k \centerdot (Address((z, f)), f_i) \in Dom(Address)$$

This ensures that for every location $v.f$ which is itself a structure with a field f', *Address* assigns an address to the location $v.f.f'$.

Intuitively, the above two constraints ensure that every location has a well-defined address. Then a store is simply a mapping from addresses to values. A store is intended to model the memory configuration at any point during the execution of a C program.

Definition 6 (Store) *A store is a mapping $\sigma : \mathbb{D} \rightarrow \mathbb{D}$ from addresses to values. The set of all stores is denoted by* **Store**.

3.1 Expressions

Let *Expr* denote the set of all *side-effect free* expressions over Var. In addition, expressions (such as variables and structure fields) that correspond to some valid memory location are called *lvalues* and form a subset of *Expr* denoted by $LValue$. Intuitively an lvalue is an expression on which the *address-of* (`&`) operator can be applied. The syntaxes of a *Expr* and $LValue$ are given by the BNF grammars shown in Table 3.1.

3.1.1 Expression Evaluation

The function Val : **Store** $\times$ *Expr* $\rightarrow \mathbb{D}$ maps a store σ and an expression e to the evaluation of e under σ. Similarly the function *Add* : **Store** $\times$ $LValue \rightarrow \mathbb{D}$ maps a store σ and an lvalue l to the address of l under σ. *Add* is defined inductively as shown in Table 3.2. Similarly, Val is defined inductively as shown in Table 3.3. The definition of Val requires the following functions over integers:

$$
\begin{array}{lll}
Expr & := & \mathbb{D} \mid Var \mid LValue\,\texttt{.}\,Field \mid \\
 & & \texttt{\&}\ LValue \mid \texttt{*}\ Expr \mid \\
 & & \texttt{-}\ Expr \mid Expr\ \texttt{+}\ Expr \mid Expr\ \texttt{-}\ Expr \mid \\
 & & Expr\ \texttt{*}\ Expr \mid Expr\ \texttt{/}\ Expr \mid \\
 & & \texttt{!}\ \ Expr \mid Expr\ \texttt{\&\&}\ Expr \mid Expr\ \texttt{||}\ Expr \mid \\
 & & \texttt{\~{}}\ Expr \mid Expr\ \texttt{\&}\ Expr \mid Expr\ \texttt{|}\ Expr \mid \\
 & & Expr\ \texttt{\^{}}\ Expr \mid Expr\ \texttt{<<}\ Expr \mid Expr\ \texttt{>>}\ Expr \\
LValue & := & Var \mid \texttt{*}Expr \mid LValue.Field
\end{array}
$$

Table 3.1: BNF grammars for $Expr$ and $LValue$.

- $+$, $-$, $\times$, $\div$ are the standard arithmetic functions of type $\mathbb{D} \times \mathbb{D} \to \mathbb{D}$.

- Neg : $\mathbb{D} \to \mathbb{D}$ is the logical negation function that maps 0 to 1 and any non-zero integer to 0.

- And : $\mathbb{D} \times \mathbb{D} \to \mathbb{D}$ is the logical conjunction function that maps any pair of non-zero integers to 1 and any other pair of integers to 0.

- Or : $\mathbb{D} \times \mathbb{D} \to \mathbb{D}$ is the logical disjunction function that maps the pair $(0, 0)$ to 0 and any other pair of integers to 1.

- BNeg : $\mathbb{D} \to \mathbb{D}$ is the bitwise negation function.

- BAnd, BOr, BXor, BLsh, and BRsh are the bitwise AND, OR, exclusive-OR, left-shift and right-shift functions of type $\mathbb{D} \to \mathbb{D}$.

Example 6 *Let v be a variable. Let us denote the address of v, i.e., $Address(v)$ by A. Let σ be a store such that $\sigma(A) = 5$. Then we want the expression ($\texttt{*}\ \texttt{\&}\ v$) to evaluate to 5 under the store σ. Let us see how this happens. First,*

$Add(\sigma, v) = Address(v) = A$. Hence, $Val(\sigma, \texttt{\&}\ v) = Add(\sigma, v) = A$. Finally, $Val(\sigma, \texttt{*}\ \texttt{\&}\ v) = \sigma(Val(\sigma, \texttt{\&}\ v)) = \sigma(A) = 5$ which is what we want.

$Add(\sigma, v)$	$=$	$Address(v)$
$Add(\sigma, \texttt{*}\ e)$	$=$	$Val(\sigma, e)$
$Add(\sigma, e\texttt{.}f)$	$=$	$Address(Add(\sigma, e), f)$

Table 3.2: Definition of function Add which maps lvalues to addresses. Note that the function $Address$ takes either a single variable argument or a pair of arguments, the first of which is an element of $\mathbb{D}$ and the second is a structure field.

$Val(\sigma, v)$	$=$	$\sigma(Add(v))$	$Val(\sigma, z)$	$=$	z
$Val(\sigma, e\texttt{.}f)$	$=$	$\sigma(Add(\sigma, e\texttt{.}f))$	$Val(\sigma, \texttt{-}\ e)$	$=$	$0 - Val(\sigma, e)$
$Val(\sigma, \texttt{\~{}}\ e)$	$=$	$\mathsf{BNeg}(Val(\sigma, e))$	$Val(\sigma, \texttt{!}\ e)$	$=$	$\mathsf{Neg}(Val(\sigma, e))$
$Val(\sigma, \texttt{\&}\ e)$	$=$	$Add(\sigma, e)$	$Val(\sigma, \texttt{*}\ e)$	$=$	$\sigma(Val(\sigma, e))$

$Val(\sigma, e_1\texttt{+}\ e_2)$	$=$	$Val(\sigma, e_1) + Val(\sigma, e_2)$
$Val(\sigma, e_1\texttt{-}\ e_2)$	$=$	$Val(\sigma, e_1) - Val(\sigma, e_2)$
$Val(\sigma, e_1\texttt{*}\ e_2)$	$=$	$Val(\sigma, e_1) \times Val(\sigma, e_2)$
$Val(\sigma, e_1\texttt{/}\ e_2)$	$=$	$Val(\sigma, e_1) \div Val(\sigma, e_2)$
$Val(\sigma, e_1\texttt{\&\&}\ e_2)$	$=$	$Val(\sigma, e_1)$ And $Val(\sigma, e_2)$
$Val(\sigma, e_1 \texttt{\|\|}\ e_2)$	$=$	$Val(\sigma, e_1)$ Or $Val(\sigma, e_2)$
$Val(\sigma, e_1\texttt{\&}\ e_2)$	$=$	$Val(\sigma, e_1)$ BAnd $Val(\sigma, e_2)$
$Val(\sigma, e_1 \texttt{\|}\ e_2)$	$=$	$Val(\sigma, e_1)$ BOr $Val(\sigma, e_2)$
$Val(\sigma, e_1\texttt{\^{}}\ e_2)$	$=$	$Val(\sigma, e_1)$ BXor $Val(\sigma, e_2)$
$Val(\sigma, e_1\texttt{<<}\ e_2)$	$=$	$Val(\sigma, e_1)$ BLsh $Val(\sigma, e_2)$
$Val(\sigma, e_1\texttt{>>}\ e_2)$	$=$	$Val(\sigma, e_1)$ BRsh $Val(\sigma, e_2)$

Table 3.3: Definition of function Val which maps expressions to values.

3.1.2 Expressions as Formulas

As we have seen before, expressions in C always evaluate to integers. The ANSI C standard does not define a separate class of Boolean expressions for use in contexts where a Boolean value is required, e.g., in branch conditions. Instead C adopts the following convention to handle such situations. The integer zero represents the truth value FALSE while any non-zero integer represents TRUE. This means that whenever an expression e is used in a Boolean context, it is implicitly compared to zero and *promoted* to TRUE if it is non-zero and to FALSE otherwise.

In the rest of this thesis we will follow the same convention and use C expressions freely even in situations where formulas are normally expected. For instance, we will soon define the weakest precondition operator which takes a C expression and an assignment statement as arguments and returns a C expression. In the literature, weakest preconditions have been traditionally defined for formulas and not expressions. In the case of C however, expressions can also be interpreted as formulas as we have just seen. Hence a weakest precondition operator on C expressions makes perfect sense.

If you find this use of expressions in the place of formulas unsettling, it might help to mentally convert expressions to formulas by comparing with zero. For example, if you see the sentence "the weakest precondition of the expression e with respect to the assignment statement a", read it instead as the following sentence: "the weakest precondition of the formula $e \neq 0$ with respect to the assignment statement a". In the rest of this thesis we will usually omit such explicit comparisons with zero for the sake of brevity.

We will carry this idea of the promotion of C expressions to formulas a little bit further. Recall that a C expression e evaluates to $Val(\sigma, e)$ under a store σ. Since e can be viewed as a formula, we can also view σ as an interpretation in the traditional logical sense. Thus, for instance, we can say that σ satisfies (is a model of) e iff $Val(\sigma, e) \neq 0$. We will denote the satisfaction of an expression e by a store σ by $\sigma \models e$ to make this correspondence even more explicit.

3.1.3 Propositions and Expressions

We wish to think of atomic propositions and expressions as counterparts in the abstract and concrete domains. Intuitively, a proposition is an abstract representative of its corresponding expression while an expression is a concrete version of its corresponding proposition. To make this notion more formal, recall that atomic propositions are drawn from a denumerable set **AP** while the set of expressions *Expr* is also denumerable.. The correspondence between the abstract propositions and the concrete expressions is captured by a concretization bijection $Concrete : \mathbf{AP} \rightarrow Expr$. In this chapter, we will use the correspondence between propositions and expressions to determine which atomic propositions hold in a concrete state of a component. In Chapter 4 we will use it additionally to present predicate abstraction.

3.2 Component

At at very high level, a component can be thought of as the control flow graph of a C procedure such as the one shown in Figure 3.1. Thus, it is essentially a directed graph whose nodes correspond to statements, and whose edges model the possible flow of control between statements. Every statement of a component has a type drawn from the set $\mathbb{T} = \{\mathsf{ASGN}, \mathsf{CALL}, \mathsf{BRAN}, \mathsf{EXIT}\}$. Intuitively, ASGN represents an assignment statement, CALL represents the invocation of a library routine, BRAN represents an **if-then-else** branch statement, and EXIT represents the exit point of a component's execution. Also, statements are associated with branch conditions, left and right hand sides, and with appropriate successor statements. We now define a component formally.

Definition 7 (Component) *A component is a tuple with eight components*

```
void component() {
  int x,y,z;
  x = y;
  if(z) {
    if(x) alpha();
    else chi();
  } else {
    if(y) beta();
    else delta();
  }
}
```

Figure 3.1: Syntax of a simple component. We will use this as a running example.

$(Stmt, Type, entry, Cond, LHS, RHS, Then, Else)$ where : (i) $Stmt$ is a finite non-empty set of statements, (ii) $Type : Stmt \rightarrow \mathbb{T}$ is a function mapping each statement to a type, (iii) $entry \in Stmt$ is the initial or entry statement, (iv) $Cond$ is a partial function of type $Stmt \hookrightarrow Expr$ which maps branch statements to their branch conditions, (v) RHS is a partial function of type $Stmt \hookrightarrow Expr$ which maps assignments to their right-hand-sides, (vi) LHS is a partial function of type $Stmt \hookrightarrow LValue$ which maps assignments to their left-hand-sides, and (vii) $Then$ and $Else$ are partial successor functions of type $Stmt \hookrightarrow Stmt$ which map statements to their **then** *and* **else** *successors respectively.*

Let $\mathcal{C}$ be a component. In order to be valid, $\mathcal{C}$ must satisfy certain sanity conditions. For instance, the type-labeling function $Type_{\mathcal{C}}$ must obey the following condition:

- **(COMP1)** There is exactly one exit statement.

$$\exists s \in Stmt_{\mathcal{C}} \centerdot Type_{\mathcal{C}}(s) = \mathsf{EXIT} \wedge \forall s' \in Stmt_{\mathcal{C}} \centerdot Type_{\mathcal{C}}(s') = \mathsf{EXIT} \implies s' = s$$

Moreover the expression-labeling functions $Cond_{\mathcal{C}}, LHS_{\mathcal{C}}$ and $RHS_{\mathcal{C}}$ must obey the following conditions:

- **(COMP2)** The **if-then-else** statements and only the **if-then-else** statements have branch conditions.

$$Dom(Cond_{\mathcal{C}}) = \{s \in Stmt_{\mathcal{C}} \mid Type_{\mathcal{C}}(s) = \mathsf{BRAN}\}$$

- **(COMP3)** The assignment statements and only the assignment statements have left-hand-sides and right-hand-sides.

$$Dom(LHS_{\mathcal{C}}) = Dom(RHS_{\mathcal{C}}) = \{s \in Stmt_{\mathcal{C}} \mid Type_{\mathcal{C}}(s) = \mathsf{ASGN}\}$$

Let us denote the set of call statements (statements of type CALL) of $\mathcal{C}$ by $Call(\mathcal{C})$.

$$Call(\mathcal{C}) = \{s \in Stmt_{\mathcal{C}} \mid Type_{\mathcal{C}}(s) = \mathsf{CALL}\}$$

Note that we have disallowed library routine calls that return values. This is because, as mentioned before, in our framework a library routine is expected to perform externally observable actions without altering the data state of the program. Finally, the successor-labeling functions $Then_{\mathcal{C}}$ and $Else_{\mathcal{C}}$ must obey the following conditions:

- **(COMP4)** Every statement except the exit point has a **then** successor.

$$Dom(Then_{\mathcal{C}}) = \{s \in Stmt_{\mathcal{C}} \mid Type_{\mathcal{C}}(s) \neq \mathsf{EXIT}\}$$

- **(COMP5)** The **if-then-else** statements and only the **if-then-else** statements have **else** successors.

$$Dom(Else_{\mathcal{C}}) = \{s \in Stmt_{\mathcal{C}} \mid Type_{\mathcal{C}}(s) = \mathsf{BRAN}\}$$

In the rest of this thesis we will only consider valid components, i.e., components that obey conditions **COMP1–COMP5**.

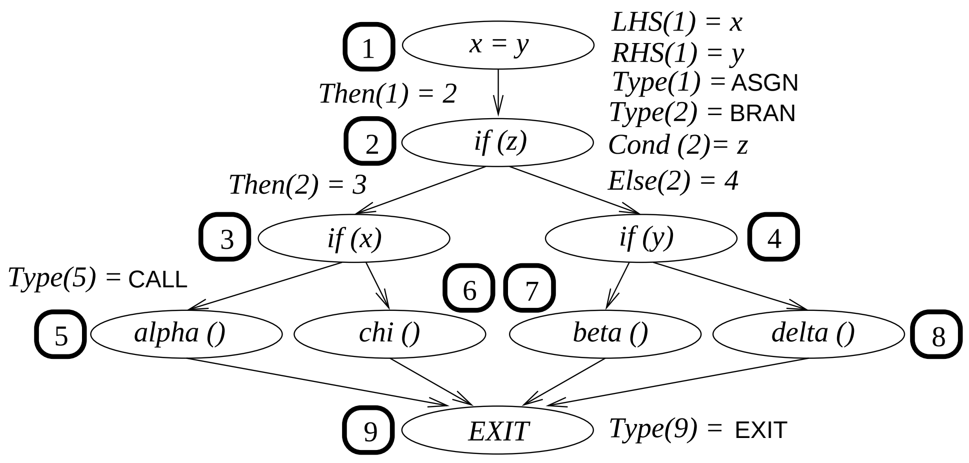


Figure 3.2: Component for the C procedure shown in Figure 3.1. We will use this as a running example.

Example 7 *Figure 3.2 shows the component $\mathcal{C}$ corresponding to the C procedure shown in Figure 3.1. The set of statements of $\mathcal{C}$ is $Stmt_{\mathcal{C}} = \{1, \ldots, 9\}$ and its initial statement is $entry_{\mathcal{C}} = 1$. Some of the statements are labeled with their types, associated expressions and successors. The set of library routines invoked by $\mathcal{C}$ is $\{alpha, beta, chi, delta\}$. Note that $\mathcal{C}$ satisfies conditions* **COMP1–COMP5**.

3.2.1 Component Semantics

In this section, we will present the concrete semantics of a component in terms of an LKS, which we will refer to as a *semantic LKS*. Intuitively, a component $\mathcal{C}$ by itself only represents the control flow structure along with the assignments, branch conditions and library routines that are invoked at each control point. In order to describe the semantic LKS, we need information about the behavior of the library routines invoked by $\mathcal{C}$ and information about the initial states, set of atomic propositions, and alphabet of the semantic LKS. This information will be provided by a *context*. Therefore we will first describe contexts before going into the details of the semantic LKS.

In general, a library routine may perform externally observable actions and also alter the data state of a component. Additionally, such behavior may be guarded by certain conditions on the data state of the component. Thus, we need a formalism that can finitely summarize the behavior of a routine, and yet is powerful enough to express guarded actions and assignments. The natural alternative is an extended finite state machine which is essentially a finite automata whose transitions are labeled with guarded commands, where a command is either an action or an assignment. We now present extended finite state machines formally:

Definition 8 (Extended Finite State Machine) *Let Σ be an alphabet. Then an extended finite state machine (EFSM for short) over Σ is a triple $(S, Init, \Delta)$ where (i) S is a finite set of states, (ii) $Init \subseteq S$ is a set of initial states, and (iii) $\delta \subseteq (S \times Expr \times \Sigma \times S) \cup (S \times Expr \times LValue \times Expr \times S)$ is a transition relation.*

The only subtle aspect of above definition is the description of the transition relation. As can be seen, the transition relation Δ consists of two kinds of elements. The first kind of element is a 4-tuple of the form (s_1, g, α, s_2). This represents a transition from state s_1 to state s_2 guarded by the expression g with the command being the action α. We will denote such a transition by $s_1 \xrightarrow{g/\alpha} s_2$. The second kind of element is a 5-tuple of the form (s_1, g, l, r, s_2). This represents a transition from state s_1 to state s_2 guarded by the expression g with the command being the assignment $l := r$. We will denote such a transition by $s_1 \xrightarrow{g/l:=r} s_2$.

The set of all EFSMs is denoted by $\mathcal{FSM}$. Let $F \in \mathcal{FSM}$ be any EFSM. Then F has a distinguished state STOP which has no outgoing transitions. Intuitively, STOP represents the termination of a library routine.

Example 8 *For example, consider a library routine lib with a single parameter p. Suppose that the behavior of lib can be described as follows:*

- *If lib is invoked with a* TRUE *(non-zero) argument, it* **either** *does action α and assigns* 1 *to variable v* **or** *does action β and assigns* 2 *to variable v.*
- *If lib is invoked with a* FALSE *(zero) argument, it assigns* 0 *to variable v.*

Then the behavior of lib can be expressed by the EFSM shown in Figure 3.3.

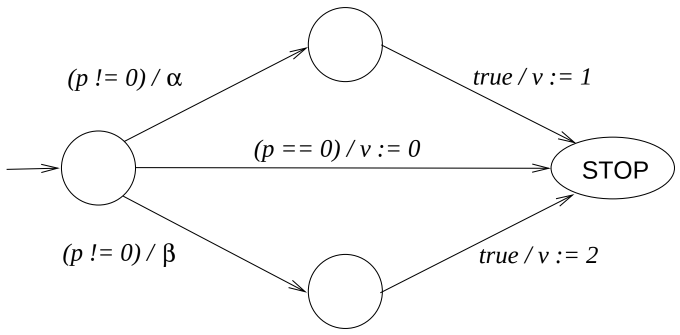


Figure 3.3: The EFSM corresponding to the library routine from Example 8. The initial state is indicated by an incoming transition with no source state.

A context for a component $\mathcal{C}$ will employ EFSMs to express the behavior of the library routines invoked by $\mathcal{C}$. However, in order to describe the semantics of $\mathcal{C}$, we need to also know about the initial states, set of atomic propositions and alphabet of the semantic LKS. As mentioned before, this information will also be provided by the context. We now define a context formally.

Definition 9 (Component Context) *A context for a component $\mathcal{C}$ is a 5-tuple $(InitCond, AP, \Sigma, Silent, FSM)$ where (i) $InitCond \in Expr$ is an initial condition, (ii) $AP \subseteq$ **AP** is a finite set of atomic propositions, (iii) $\Sigma \subseteq ObsAct$ is a set of observable actions, (iv) $Silent \in SilAct \setminus \{\tau\}$ is a silent action, and (v)*

$FSM : Call(\mathcal{C}) \rightarrow \mathcal{FSM}$ is a function mapping call statements of $\mathcal{C}$ to EFSMs over the alphabet $\Sigma \cup \{Silent\}$.

Example 9 *Recall the component $\mathcal{C}$ from Figure 3.2. Now we let γ be the context for $\mathcal{C}$ such that: (i) $InitCond_\gamma = \text{TRUE}$, (ii) $AP_\gamma = \emptyset$, (iii) $\Sigma_\gamma = \{\alpha, \beta, \chi, \delta\}$, (iv) $Silent_\gamma = \tau_1$, and (v) recall that $\mathcal{C}$ has four call statements $\{5, 6, 7, 8\}$ which invoke library routines $alpha, chi, beta$ and $delta$ respectively; then FSM_γ maps the call statements $\{5, 6, 7, 8\}$ respectively to the EFSMs $F_\alpha, F_\chi, F_\beta$ and F_δ shown in Figure 3.4. Intuitively this means that these routines simply perform the actions α, χ, β and δ respectively.*

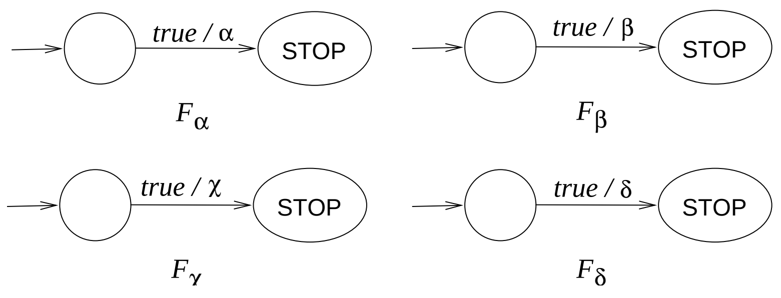


Figure 3.4: The EFSMs corresponding to the context from Example 9.

Note that a context is specific to $\mathcal{C}$ since it must provide EFSMs for only the call statements of $\mathcal{C}$. Let $\gamma = (InitCond, AP, \Sigma, Silent, FSM)$ be any context for $\mathcal{C}$. Then the semantics of $\mathcal{C}$ under γ is denoted by $[\![\mathcal{C}]\!]_\gamma$. In the rest of this section the context γ will be fixed and therefore we will often omit it when it is obvious. For example, we will write $[\![\mathcal{C}]\!]$ to mean $[\![\mathcal{C}]\!]_\gamma$. Formally, $[\![\mathcal{C}]\!]$ is an LKS such that:

- $[\![\mathcal{C}]\!]$ has two kinds of states - *normal* and *inlined*. A normal state is simply a pair consisting of a statement of $\mathcal{C}$ and a store.

$$S^{normal} = Stmt_{\mathcal{C}} \times \textbf{Store}$$

An inlined state is obtained by inlining EFSMs at corresponding call statements. Recall that for any call statement s, $FSM(s)$ denotes the EFSM corresponding to s. Therefore, an inlined state is simply a triple (s, σ, ι) where s is a call statement, σ is a store and ι is a state of $FSM(s)$. More formally the set of inlined states $S^{inlined}$ is defined as:

$$S^{inlined} = \{(s, \sigma, \iota) \mid s \in Call(\mathcal{C}) \land \sigma \in \textbf{Store} \land \iota \in S_{FSM(s)}\}$$

where S_F denotes the set of states of an EFSM F. Finally, a state of $[\![\mathcal{C}]\!]$ is either a normal state or an inlined state.

$$S_{[\![\mathcal{C}]\!]} = S^{normal} \cup S^{inlined}$$

- An initial state of $[\![\mathcal{C}]\!]$ corresponds to the entry statement of $\mathcal{C}$ and a store that satisfies the initial condition $InitCond$ specified by the context γ.

$$Init_{[\![\mathcal{C}]\!]} = \{(entry_{\mathcal{C}}, \sigma) \mid \sigma \vDash InitCond\}$$

- Recall that AP is the set of atomic propositions specified by the context γ. The atomic propositions of $[\![\mathcal{C}]\!]$ are the same as those specified by γ.

$$AP_{[\![\mathcal{C}]\!]} = AP$$

- The labeling function $L_{[\![\mathcal{C}]\!]}$ of $[\![\mathcal{C}]\!]$ is consistent with the expressions corresponding to the atomic propositions. Recall that the bijection $Concrete$ maps atomic propositions to expressions. Since the propositional labeling does not depend on the inlined EFSM state, its definition will be identical for normal and inlined states. More formally:

$$L_{[\![\mathcal{C}]\!]}(s, \sigma) = L_{[\![\mathcal{C}]\!]}(s, \sigma, \iota) = \{p \in AP_{[\![\mathcal{C}]\!]} \mid \sigma \vDash Concrete(p)\}$$

- The alphabet of $[\![\mathcal{C}]\!]$ contains the specified observable and silent actions. Recall that Σ is the set of observable actions associated with the context γ and $Silent$ is the silent action associated with the context γ.

  $$\Sigma_{[\![\mathcal{C}]\!]} = \Sigma \cup \{Silent\}$$

  Note that $\tau \notin \Sigma_{[\![\mathcal{C}]\!]}$. This fact will be used crucially for the compositional verification technique presented later in this thesis.

- The transition relation of the semantics $[\![\mathcal{C}]\!]$ is defined in the following section.

3.2.2 Transition Relation of $[\![\mathcal{C}]\!]$

In the rest of this section we will write $Type$, $Then$, $Else$, $Cond$, LHS and RHS to mean $Type_{\mathcal{C}}$, $Then_{\mathcal{C}}$, $Else_{\mathcal{C}}$, $Cond_{\mathcal{C}}$, $LHS_{\mathcal{C}}$ and $RHS_{\mathcal{C}}$ respectively. We will describe outgoing transitions from normal and inlined states separately.

Normal States. Let (s, σ) be a normal state of $[\![\mathcal{C}]\!]$. Recall that $s \in Stmt_{\mathcal{C}}$ is a statement of $\mathcal{C}$ and $\sigma \in \textbf{Store}$. We consider each possible value of the type $Type(s)$ of s separately.

- $Type(s) = \mathsf{EXIT}$. In this case (s, σ) has no outgoing transitions.

- $Type(s) = \mathsf{BRAN}$. Recall that $Cond(s)$ is the branch condition associated with s while $Then(s)$ and $Else(s)$ are the **then** and **else** successors of s. In this case (s, σ) performs the $Silent$ action and moves to either $Then(s)$ or $Else(s)$ depending on the satisfaction of the branch condition. The store remains unchanged. Formally:

  $$\sigma \vDash Cond(s) \implies (s, \sigma) \xrightarrow{Silent} (Then(s), \sigma)$$

$$\sigma \not\models Cond(s) \implies (s, \sigma) \xrightarrow{Silent} (Else(s), \sigma)$$

- $Type(s) = \mathsf{ASGN}$. In this case (s, σ) performs the $Silent$ action and moves to the **then** successor while the store is updated as per the assignment. Formally, for any assignment statement a, let $\sigma[a]$ be the store such that the following two conditions hold:

$$\forall x \in \mathbb{D} \centerdot x \neq Add(\sigma, LHS(s)) \implies \sigma[a](x) = \sigma(x)$$

$$\sigma[a](Add(\sigma, LHS(s))) = Val(\sigma, RHS(s))$$

  In other words, $\sigma[a]$ is the store obtained by updating σ with the assignment a. Recall that $LHS(s)$ and $RHS(s)$ are the left and right hand side expressions associated with s. Then:

$$(s, \sigma) \xrightarrow{Silent} (Then(s), \sigma[LHS(s) := RHS(s)])$$

- $Type(s) = \mathsf{CALL}$. In this case (s, σ) performs the $Silent$ action and moves to an initial state of the EFSM $FSM(s)$ corresponding to s. The store remains unchanged. Recall that $Init_{FSM(s)}$ denotes the set of initial states of $FSM(s)$. Then:

$$\forall \iota \in Init_{FSM(s)} \centerdot (s, \sigma) \xrightarrow{Silent} (s, \sigma, \iota)$$

Inlined States. Let $s \in Call(\mathcal{C})$, $\sigma \in$ **Store** and (s, σ, ι) be an inlined state. Recall that in this case ι must be a state of the EFSM $FSM(s)$ corresponding to s. Also recall that the transitions of $FSM(s)$ are labeled with guarded commands of the form g/α or $g/l := r$. We consider four possible types of outgoing transitions of $FSM(s)$ from the state ι.

- $\iota \xrightarrow{g/\alpha} \iota'$ **and** $\iota' \neq \mathsf{STOP}$. If the store σ satisfies the guard g, then (s, σ, ι) performs action α and moves to the inlined state corresponding to ι'. The store remains unchanged. Formally:

$$\sigma \vDash g \implies (s, \sigma, \iota) \xrightarrow{\alpha} (s, \sigma, \iota')$$

- $\iota \xrightarrow{g/\alpha} \iota'$ **and** $\iota' = \mathsf{STOP}$. If the store σ satisfies the guard g, then (s, σ, ι) performs action α and returns from the library routine call. The store remains unchanged. Formally:

$$\sigma \vDash g \implies (s, \sigma, \iota) \xrightarrow{\alpha} (Then(s), \sigma)$$

- $\iota \xrightarrow{g/l:=r} \iota'$ **and** $\iota' \neq \mathsf{STOP}$. If the store σ satisfies the guard g, then (s, σ, ι) performs *Silent* and moves to the inlined state corresponding to ι'. The store is updated as per the assignment $l := r$. Recall that $\sigma[l := r]$ denotes the new store obtained by updating σ with $l := r$. Then:

$$\sigma \vDash g \implies (s, \sigma, \iota) \xrightarrow{Silent} (s, \sigma[l := r], \iota')$$

- $\iota \xrightarrow{g/l:=r} \iota'$ **and** $\iota' = \mathsf{STOP}$. If the store σ satisfies the guard g, then (s, σ, ι) performs *Silent* and returns from the library routine call. The store is updated as per the assignment $l := r$. Recall that $\sigma[l := r]$ denotes the new store obtained by updating σ with $l := r$. Then:

$$\sigma \vDash g \implies (s, \sigma, \iota) \xrightarrow{Silent} (Then(s), \sigma[l := r])$$

3.3 Restrictions on C Programs

We assume that our input C programs are in the CIL [92] format. This can be easily achieved by preprocessing the input C programs using the CIL [30] tool developed by Necula et. al. The CIL format is essentially a very simple subset of C with precise semantics. For instance, the format does not allow procedure calls inside the argument of another procedure call. Also, expressions with side-effects (such as `x++`) and shortcut evaluations (such as `a && b`) are disallowed. However CIL is expressive enough so that any valid C program can be tranlated into an equivalent CIL program. This is precisely what the CIL tools achieves.

Additionally we disallow pointer dereferences on the left-hand-sides of assignments as well as the use of function pointers in making indirect library routine calls. This can be achieved using alias information about the pointers being dereferenced. For instance suppose we have the following expression:

```
*p = e;
*fp();
```

Given that the pointer `p` can point to either variables `x` or `y.z`, and the pointer `fp` can point to either routines `foo` or `bar`, we can rewrite the above code fragment into the following while preserving its semantics:

```
if (p == & x) x = e;
else y.z = e;
if (fp == & foo) foo();
else bar();
```

Note that we could also include such aliasing information in a component's context since the semantics of a component clearly depends on its aliasing scenario. However, we chose not to do this for two main reasons. First, aliasing scenarios are more *integral* to a component's definition than information which should be present in a context. While a context is like a component's environment, aliasing information is usually embedded more directly in the actual source code and could be extracted, e.g., via alias analysis. More importantly, the code transformation above converts aliasing possibilities into branch conditions. These branch conditions can be subsequently used to infer appropriate predicates for abstraction refinement. The abstraction refinement procedure in MAGIC is presented in further detail in Chapter 6. Finally, preprocessing away pointer dereferences on the left-hand-sides of assignments will simplify our presentation of predicate abstraction in Chapter 4. In particular, it will enable us to use a standard version of the weakest precondition operator (cf. Definition 15) instead of an extended version. This usually leads to simpler theorem prover queries while performing admissibility checks.

3.4 Symbolic Representation of States

Let **State** denote the set of all states of the concrete semantics $[\![\mathcal{C}]\!]$ of component $\mathcal{C}$ with respect to context γ. At several points in the rest of this thesis, we will require the ability to manipulate (possibly infinite) subsets of **State**. In this section we present such a framework. The basic idea is to represent the statements explicitly and to model the stores symbolically using C expressions.

Consider any subset $\mathcal{S}$ of the set of states **State** of $[\![\mathcal{C}]\!]$. Recall that each element of $\mathcal{S}$ is either a normal state of the form (s, σ) or an inlined state (s, σ, ι) where s is a statement of $\mathcal{C}$, σ is a store and ι is an inlined EFSM state. Since both s and ι are finitary, we can easily partition $\mathcal{S}$ into a finite number of equivalence classes where all the elements of a particular equivalence class agree on their s and ι components. In other words, for a fixed s and ι, the equivalence class $\mathcal{S}(s, \iota)$ is defined as follows:

$$\mathcal{S}(s, \iota) = \{(s', \sigma) \in \mathcal{S} \mid s' = s\} \cup \{(s', \sigma, \iota') \in \mathcal{S} \mid s' = s \wedge \iota' = \iota\}$$

Clearly, the states within a particular equivalence class differ only in their σ components. Now suppose that for each equivalence class $\mathcal{S}(s, \iota)$ there exists an expression e such that $(s, \sigma) \in \mathcal{S} \iff \sigma \vDash e$ and $(s, \sigma, \iota) \in \mathcal{S} \iff \sigma \vDash e$. Then we can represent the equivalence class $\mathcal{S}(s, \iota)$ symbolically using the triple (s, ι, e) and hence we can represent the entire set $\mathcal{S}$ using a collection of such triples, one for each value of s and ι.

Clearly not all subsets of **State** are representable (or expressible) by the above scheme. This follows from a simple counting argument – only countably many $\mathcal{S}$'s are expressible while the set of all possible $\mathcal{S}$'s, i.e., $2^{\mathbf{State}}$, is clearly uncountable. However, as we shall see in the rest of this section, the expressible $\mathcal{S}$'s form an important subset of $2^{\mathbf{State}}$ which we shall denote by $\mathcal{E}$. A set $\mathcal{S} \subseteq \mathbf{State}$ is said to be *expressible* if it belongs to $\mathcal{E}$. We now present this notion formally.

Recall that $Stmt_{\mathcal{C}}$ denotes the set of statements of $\mathcal{C}$ and for any call statement s of $\mathcal{C}$, $FSM(s)$ denotes the EFSM corresponding to s. Instead of using triples of the form (s, ι, e) we will represent an expressible set of states using a function. In particular, let D be the set of all valid statements s and pairs (s, ι) of statements and inlined EFSM states. In other words:

$$D = Stmt_{\mathcal{C}} \cup \{(s, \iota) \mid s \in Call(\mathcal{C}) \wedge \iota \in S_{FSM(s)}\}$$

where S_F denotes the set of states of an EFSM F. Then a *representation* $R : D \to Expr$ is a function from the set D to the set of expressions.

Definition 10 *A representation is a function $D \to Expr$. A representation R corresponds to a set $\mathcal{S}$, of concrete states iff the following two conditions holds:*

$$\forall s \in D \centerdot \forall \sigma \in \mathbf{Store} \centerdot (s, \sigma) \in \mathcal{S} \iff \sigma \vDash R(s)$$

$$\forall (s, \iota) \in D \centerdot \forall \sigma \in \mathbf{Store} \centerdot (s, \sigma, \iota) \in \mathcal{S} \iff \sigma \vDash R(s, \iota)$$

Let us denote the set of all representations by **Rep**. *Recall that* **State** *denotes the set* *of all states of $[\![\mathcal{C}]\!]$. For any $R \in \mathbf{Rep}$ and any $\mathcal{S} \subseteq \mathbf{State}$, we will write $R \equiv \mathcal{S}$ to mean that R corresponds to $\mathcal{S}$.*

A set of concrete states $\mathcal{S} \subseteq \mathbf{State}$ is said to be expressible iff there exists a representation R such that $R \equiv \mathcal{S}$. We denote the set of all expressible subsets of **State** by $\mathcal{E}$. In other words:

$$\mathcal{E} = \{\mathcal{S} \subseteq \mathbf{State} \mid \exists R \in \mathbf{Rep} \centerdot R \equiv \mathcal{S}\}$$

Theorem 4 *We note below a set of simple results about expressible sets of states.*

1. *The set* **State** *of all states of $[\![\mathcal{C}]\!]$ is expressible.*

2. *The set $Init_{[\![\mathcal{C}]\!]}$ of initial states of $[\![\mathcal{C}]\!]$ is expressible.*

3. *If $\mathcal{S}$ is expressible then so is its complement* $\mathbf{State} \setminus \mathcal{S}$.

4. *If $\mathcal{S}_1$ and $\mathcal{S}_2$ are expressible then so are $\mathcal{S}_1 \cup \mathcal{S}_2$ and $\mathcal{S}_1 \cap \mathcal{S}_2$.*

5. *For any proposition p the set of states labeled with p is expressible.*

Proof. Recall that D denotes the domain of any representation.

1. The following representation R corresponds to **State**: $\forall d \in D \centerdot R(d) = \text{TRUE}$.

2. Recall that any initial state of $[\![\mathcal{C}]\!]$ is of the form $(entry_{\mathcal{C}}, \sigma)$ where $entry_{\mathcal{C}}$ is the entry statement and the store σ satisfies the initial guard $InitCond$ of the context γ. Hence the following representation R corresponds to the set $Init_{[\![\mathcal{C}]\!]}$ of initial states of $[\![\mathcal{C}]\!]$.

$$R(entry_{\mathcal{C}}) = InitCond \bigwedge \forall d \in D \centerdot d \neq entry_{\mathcal{C}} \implies R(d) = \text{FALSE}$$

3. Let R be a representation corresponding to $\mathcal{S}$. Then the following representation R' corresponds to **State** $\setminus \mathcal{S}$: $\forall d \in D \centerdot R'(d) = \neg R(d)$.

4. Let R_1 and R_2 be representations corresponding to $\mathcal{S}_1$ and $\mathcal{S}_2$ respectively. Then the following representations $R_\cup$ and $R_\cap$ correspond to $\mathcal{S}_1 \cup \mathcal{S}_2$ and $\mathcal{S}_1 \cap \mathcal{S}_2$ respectively.
$$\forall d \in D \centerdot R_\cup(d) = R_1(d) \vee R_2(d)$$
$$\forall d \in D \centerdot R_\cap(d) = R_1(d) \wedge R_2(d)$$

5. Recall that the bijection *Concrete* maps atomic propositions to expressions. Let p be any proposition. Then the following representation R corresponds to the set of states labeled with p: $\forall d \in D \centerdot R(d) = Concrete(p)$.

$\square$

In the rest of this thesis we will manipulate sets of concrete states using their representations. In particular, we will use the following two functions for restriction with respect to a set of atomic propositions and pre-image computation with respect to an action.

3.4.1 Restriction

The function $Restrict : \mathcal{E} \times 2^{AP} \rightarrow \mathcal{E}$ restricts an expressible set of states with respect to a propositional labeling. Intuitively, $Restrict(\mathcal{S}, P)$ contains every state in $\mathcal{S}$ with propositional labeling P. Formally:

$$Restrict(\mathcal{S}, P) = \{s \in \mathcal{S} \mid L_{[\![\mathcal{C}]\!]}(s) = P\}$$

3.4.2 Pre-image

The function $PreImage : \mathcal{E} \times \Sigma \rightarrow \mathcal{E}$ maps an expressible set of states to its pre-image under a particular action. Intuitively, $PreImage(\mathcal{S}, \alpha)$ contains every state which has an α-successor in $\mathcal{S}$. Formally:

$$PreImage(\mathcal{S}, \alpha) = \{s \in S_{[\![\mathcal{C}]\!]} \mid Succ_{[\![\mathcal{C}]\!]}(s, \alpha) \cap \mathcal{S} \neq \emptyset\}$$

We note that due to Theorem 4 both *Restrict* and *PreImage* are effectively computable in the sense that if a representation corresponding to the argument $\mathcal{S}$ is available, then we can also compute a representation corresponding to the final result. Pre-image computation is possible since we are only concerned with assignments, **if-then-else**'s etc. and not, for example, with **while** statements. Similar approaches for representing and manipulating sets of states symbolically have been used previously by, among others, Clarke [31] and Cook [43]. Finally, emptiness and universality of an expressible set of states $\mathcal{S}$ can be checked using a theorem prover if, once again, we have a representation corresponding to $\mathcal{S}$. Of course, emptiness and universality are undecidable in general.

3.5 Program

A program consists of a set of components. The execution of a program involves the concurrent execution of its constituent components.

Definition 11 (Program) *A program $\mathcal{P}$ is a finite sequence $\langle \mathcal{C}_1, \ldots, \mathcal{C}_n \rangle$ where each $\mathcal{C}_i$ is a component.*

Naturally, a context for $\mathcal{P}$ must consist of a sequence of component contexts, one for each $\mathcal{C}_i$. In addition, the silent actions of each of these component contexts must be different. This prohibits the components from synchronizing with each other on their silent actions during execution.

Definition 12 (Program Context) *A context* Γ *for a program* $\mathcal{P} = \langle \mathcal{C}_1, \ldots, \mathcal{C}_n \rangle$ *is a sequence of component contexts* $\langle \gamma_1, \ldots, \gamma_n \rangle$ *such that: (i)* $\forall i \in \{1, \ldots, n\} \centerdot \gamma_i$ *is a context for component* $\mathcal{C}_i$*, and (ii)* $\forall i \in \{1, \ldots, n\} \centerdot \forall j \in \{1, \ldots, n\} \centerdot i \neq j \implies Silent_{\gamma_i} \neq Silent_{\gamma_j}$*.*

The semantics of a program is obtained by the parallel composition of the semantics of its components.

Definition 13 (Program Semantics) *Let* $\mathcal{P} = \langle \mathcal{C}_1, \ldots, \mathcal{C}_n \rangle$ *be a program and* $\Gamma = \langle \gamma_1, \ldots, \gamma_n \rangle$ *be a context for* $\mathcal{P}$*. The semantics of* $\mathcal{P}$ *under* Γ*, denoted by* $[\![\mathcal{P}]\!]_\Gamma$*, is an LKS defined as follows:*

$$[\![\mathcal{P}]\!]_\Gamma = [\![\mathcal{C}_1]\!]_{\gamma_1} \parallel \cdots \parallel [\![\mathcal{C}_n]\!]_{\gamma_n}$$

Chapter 4

Abstraction

In this chapter we will present our abstraction scheme used for obtaining finite LKS models from C programs. An LKS $\widehat{M}$ is said to be an abstraction of an LKS M if $M \preccurlyeq \widehat{M}$. Therefore, for our purposes, the concepts of abstraction and simulation are synonymous. First, we will present a general notion of abstraction based on *abstraction mappings*. Later on we will describe a specific abstraction technique called *predicate abstraction* [63] which we actually employ for model construction. This abstraction technique has also been used by others for the verification of both hardware [33] and software [6, 66] systems.

4.1 Abstraction mapping

Let M and $\widehat{M}$ be two LKSs. A function $\mathcal{H} : S_M \rightarrow S_{\widehat{M}}$ is said to be an abstraction mapping iff it obeys the following conditions:

- $\forall s \in S_M \centerdot L_M(s) = L_{\widehat{M}}(\mathcal{H}(s))$
- $\forall\, (s, \alpha, s') \in T_M \centerdot (\mathcal{H}(s), \alpha, \mathcal{H}(s')) \in T_{\widehat{M}}$

- $\forall s \in Init_M \centerdot \mathcal{H}(s) \in Init_{\widehat{M}}$

The following well-known result [36] captures the relationship between abstraction mappings and abstractions. Abstractions obtained via abstraction mappings, particularly in the context of hardware verification, are often referred to in the literature as *existential* abstractions.

Theorem 5 *Let M and $\widehat{M}$ be two LKSs such that: (i) $AP_M = AP_{\widehat{M}}$, (ii) $\Sigma_M = \Sigma_{\widehat{M}}$, and (iii) there exists an abstraction mapping $\mathcal{H} : S_M \to S_{\widehat{M}}$. Then $M \preccurlyeq \widehat{M}$.*

Proof. Define the relation $\mathcal{R} = \{(s, \mathcal{H}(s)) \mid s \in S_M\}$. Prove that (i) $\mathcal{R}$ is a simulation relation, and (ii) $\forall s \in Init_M \centerdot \exists \widehat{s} \in Init_{\widehat{M}} \centerdot s\mathcal{R}\widehat{s}$.

□

4.2 Predicate

A predicate is simply a C expression. Recall that any C expression can also be viewed as a formula (cf. Section 3.1.2). Hence our use of expressions as predicates is perfectly natural. Let us denote the set of Boolean values $\{\text{TRUE}, \text{FALSE}\}$ by $\mathbb{B}$. Let $Pred$ be a set of predicates. Recall that $\mathbf{AP}$ denotes the set of atomic propositions and that the bijection $Concrete$ maps atomic propositions to expressions. Let us denote the inverse of $Concrete$ by $Prop$. In other words, $Prop : Expr \to \mathbf{AP}$ is a bijection defined as follows:

$$\forall e \in Expr \centerdot Prop(e) = p \in \mathbf{AP} \iff Concrete(p) = e$$

We extend the function $Prop$ to operate over sets of expressions in the natural manner.

$$Prop(Pred) = \{Prop(e) \mid e \in Pred\}$$

$Prop(Pred)$ can be thought of as the set of propositions obtained from the set of predicates $Pred$ by replacing each predicate in $Pred$ with its corresponding proposition.

Let $AP = \{p_1, \ldots, p_k\}$ be a set of propositions. Since each element of AP can be assigned a Boolean value, a valuation of AP is simply a mapping from AP to $\mathbb{B}$. The set of all valuations of AP is denoted by $PropVal(AP)$. Given a proposition p and a Boolean value b, let us define p^b as follows: if $b =$ TRUE then $p^b = Concrete(p)$ else $p^b = \,!\,Concrete(p)$. Then the induced concretization function $Concrete : PropVal(AP) \rightarrow Expr$ is defined as follows:

$$Concrete(V) = p_1^{V(p_1)} \texttt{ \&\& } \ldots \texttt{ \&\& } p_k^{V(p_k)}$$

where $V(p)$ denotes the value of proposition p according to the valuation V. As a special case, if $AP = \emptyset$, then it has just one valuation which we denote by $\perp$ and we adopt the convention that $Concrete(\perp) = 1$. Recall that 1 represents TRUE as per the C expression semantics. This means that an empty valuation always concretizes to TRUE.

The **key idea behind predicate abstraction** is that sets of concrete stores are abstractly represented by propositional valuations. In particular, a *valuation V abstractly represents the set of stores σ which satisfy the concretization $Concrete(V)$ of V.* Moreover, just as there is a notion of satisfaction $\models$ of expressions by stores, there is a notion of abstract satisfaction of expressions by valuations. Intuitively, a valuation V abstractly satisfies an expression e iff there is a store σ such that V abstractly represents σ and $\sigma \models e$. We call this notion of abstract satisfaction *admissibility* and present it formally next.

Definition 14 (Admissibility) *Let AP be a set of propositions, $V \in PropVal(AP)$*

be a valuation of AP and $e \in Expr$ be an expression. Recall that the C expression $Concrete(V)$ denotes the concretization of the valuation V. Then V is said to be admissible with e, denoted by $V \Vdash e$, iff the following condition holds:

$$\exists \sigma \in \mathbf{Store} \,\centerdot\, \sigma \models Concrete(V) \;\texttt{\&\&}\; e$$

If $V' \in PropVal(AP')$ is a valuation of another set of propositions AP', then we write $V \Vdash V'$ to mean $V \Vdash Concrete(V')$.

Admissibility and Satisfaction. We have intentionally used a symbol for denoting admissibility ($\Vdash$) which is similar to that used to denote satisfaction ($\models$) of an expression by a store. Our intention is to highlight the fact that admissibility is essentially a form of consistency between a valuation and an expression or between two valuations. At an abstract level, admissibility plays a role similar to that of satisfaction or logical entailment. This correspondence is further highlighted by the similarity between the description of the concrete and abstract semantics of a component presented in Chapter 3 and Chapter 4 respectively.

Checking Admissibility in Practice. In order to perform predicate abstraction it will be essential to perform several admissibility checks. Suppose we wish to check admissibility between a valuation V and an expression e (or between two valuations V and V'). This boils down to checking the satisfiability of the expression $Concrete(V)$ `&&` e (or $Concrete(V)$ `&&` $Concrete(V')$). We will use a theorem prover to discharge this satisfiability check. However the problem is undecidable in general and hence the theorem prover might time-out with an “I don't know”. In such inconclusive cases, to guarantee the soundness of our predicate abstraction we must make a conservative decision, i.e., assume that V and e (or V and V') are indeed admissible.

Definition 15 (Weakest Precondition) *Given an expression e, and an assignment a of the form* `lhs = rhs`*, the weakest precondition [54, 70] of e with respect to a, denoted by $\mathcal{WP}[a](e)$, is the expression obtained from e by simultaneously replacing each occurrence of lhs with rhs.*

4.3 Predicate Mapping

In order to do predicate abstraction, we need to know the set of predicates associated with each statement of a component. This association is captured by a predicate mapping. Let $\mathcal{C}$ be a component and γ be a context for $\mathcal{C}$. Recall that AP_γ is the set of atomic propositions specified by γ. Then a predicate mapping is a function from the statements of $\mathcal{C}$ to sets of predicates such that we have sufficient predicates to determine whether an atomic proposition $p \in AP_\gamma$ holds or does not hold at any abstract state. Recall that $Concrete(p)$ denotes the expression associated with any atomic proposition p. Therefore, for any $p \in AP_\gamma$, and for any statement s, a predicate mapping must associate the expression $Concrete(p)$ as a predicate at s. We now give a more formal definition.

Definition 16 (Predicate Mapping) *Let $\mathcal{C}$ be a component and γ be a context for $\mathcal{C}$. Recall that $Stmt_{\mathcal{C}}$ denotes the set of statements of $\mathcal{C}$, AP_γ denotes the set of atomic propositions specified by γ, and Concrete is a mapping from the atomic propositions to expressions. A function $\Pi : Stmt_{\mathcal{C}} \rightarrow 2^{Expr}$ is said to be a predicate mapping for $\mathcal{C}$*

compatible with γ iff the following condition holds:

$$\forall s \in Stmt_{\mathcal{C}} \centerdot \forall p \in AP_{\gamma} \centerdot Concrete(p) \in \Pi(s)$$

In other words, Π is compatible with γ iff the set of predicates associated by Π with every statement s of $\mathcal{C}$ contains the predicates corresponding to the atomic propositions specified by the context γ. The predicate mapping Π will be fixed in the rest of this chapter. Hence, for any statement s we will simply write $PropVal(s)$ to mean $PropVal(Prop(\Pi(s)))$ where $Prop(\Pi(s))$ denotes the set of atomic propositions corresponding to the set of predicates associated with s by Π.

4.4 Predicate Abstraction

We are now ready to present predicate abstraction formally. We first present predicate abstraction for a component and then extend it to a program. The **reader is advised** to perform a comparative study of the remainder of this section and Section 3.2.1. This will enable him to get a clearer picture of the relationship between component semantics and predicate abstraction, and grasp at an intuitive level the significance and correctness of Theorem 6.

Let $\mathcal{C}$ be a component, $\gamma = (InitCond, AP, \Sigma, Silent, FSM)$ be any context for $\mathcal{C}$ and Π be a predicate mapping for $\mathcal{C}$ compatible with γ. Then the predicate abstraction of $\mathcal{C}$ under γ and with respect to Π is denoted by $[\![\mathcal{C}]\!]^{\Pi}_{\gamma}$. In the rest of this section the context γ and the predicate mapping Π will be fixed and therefore we will often omit them when they are obvious. For example, we will write $[\![\mathcal{C}]\!]$ to mean $[\![\mathcal{C}]\!]^{\Pi}_{\gamma}$. Formally, $[\![\mathcal{C}]\!]$ is an LKS such that:

- $[\![\mathcal{C}]\!]$ has two kinds of states - *normal* and *inlined*. A normal state is simply a pair consisting of a statement of $\mathcal{C}$ and a valuation.

$$S^{normal} = \{(s, V) \mid s \in Stmt_{\mathcal{C}} \wedge V \in PropVal(s)\}$$

where $PropVal(s)$ denotes the set of valuations of the propositions corresponding to the set of predicates associated with the statement s. An inlined state is obtained by inlining EFSMs at corresponding call statements. Recall that for any call statement s, $FSM(s)$ denotes the EFSM corresponding to s. Therefore, an inlined state is simply a triple (s, V, ι) where s is a call statement, V is a valuation and ι is a state of $FSM(s)$. More formally, the set of inlined states $S^{inlined}$ is defined as:

$$S^{inlined} = \{(s, V, \iota) \mid s \in Call(\mathcal{C}) \wedge V \in PropVal(s) \wedge \iota \in S_{FSM(s)}\}$$

where $PropVal(s)$ denotes the set of valuations of the propositions corresponding to the set of predicates associated with the statement s, and S_F denotes the set of states of an EFSM F. Finally, a state of $[\![\mathcal{C}]\!]$ is either a normal state or an inlined state.

$$S_{[\![\mathcal{C}]\!]} = S^{normal} \cup S^{inlined}$$

- An initial state of $[\![\mathcal{C}]\!]$ corresponds to the entry statement $entry_{\mathcal{C}}$ of $\mathcal{C}$ and a valuation that is admissible with the initial condition $InitCond$ specified by the context γ.

$$Init_{[\![\mathcal{C}]\!]} = \{(entry_{\mathcal{C}}, V) \mid V \Vdash InitCond\}$$

- Recall that AP is the set of atomic propositions specified by the context γ. The atomic propositions of $[\![\mathcal{C}]\!]$ are the same as those specified by γ.

$$AP_{[\![\mathcal{C}]\!]} = AP$$

- The labeling function $L_{[\![\mathcal{C}]\!]}$ of $[\![\mathcal{C}]\!]$ is consistent with the valuations. Since the propositional labeling does not depend on the inlined EFSM state, its definition will be identical for normal and inlined states. More formally:

  $$L_{[\![\mathcal{C}]\!]}(s, V) = L_{[\![\mathcal{C}]\!]}(s, V, \iota) = \{p \in AP \mid V(p) = \text{TRUE}\}$$

  Note that in the above definitions, the value $V(p)$ is always well-defined because the predicate mapping Π is compatible with the context γ. This ensures that for any atomic proposition p, and any statement s of the component $\mathcal{C}$, the concretization $\mathcal{C}(p)$ of p always belongs to the predicate set $\Pi(s)$ associated with the statement s by the predicate mapping Π.

- The alphabet of $[\![\mathcal{C}]\!]$ contains the specified observable and silent actions. Recall that Σ is the set of observable actions associated with the context γ and $Silent$ is the silent action associated with the context γ.

  $$\Sigma_{[\![\mathcal{C}]\!]} = \Sigma \cup \{Silent\}$$

  Note once again that $\tau \notin \Sigma_{[\![\mathcal{C}]\!]}$ and that this fact will be used for the compositional verification technique presented later in this thesis.

- The transition relation of the predicate abstraction $[\![\mathcal{C}]\!]$ is defined in the following section.

4.4.1 Transition Relation of $[\![\mathcal{C}]\!]$

In the rest of this section we will write $Type$, $Then$, $Else$, $Cond$, LHS and RHS to mean $Type_{\mathcal{C}}$, $Then_{\mathcal{C}}$, $Else_{\mathcal{C}}$, $Cond_{\mathcal{C}}$, $LHS_{\mathcal{C}}$ and $RHS_{\mathcal{C}}$ respectively. We will describe outgoing transitions from normal and inlined states separately.

Normal States. Let (s, V) be a normal state of $[\![\mathcal{C}]\!]$. Recall that $s \in Stmt_{\mathcal{C}}$ is a statement of $\mathcal{C}$ and $V \in PropVal(s)$ is a valuation of the propositions corresponding to the set of predicates associated with s. We consider each possible value of $Type(s)$ separately.

- $Type(s) = \mathsf{EXIT}$. In this case (s, V) has no outgoing transitions.

- $Type(s) = \mathsf{BRAN}$. Recall that $Cond(s)$ is the branch condition associated with s while $Then(s)$ and $Else(s)$ are the **then** and **else** successors of s. In this case (s, V) performs the $Silent$ action and moves to $Then(s)$ or $Else(s)$ depending on the satisfaction of the branch condition. The new valuation must be admissible with the old one. Let $V'_{then} \in PropVal(Then(s))$ and $V'_{else} \in PropVal(Else(s))$. Then:

$$V \Vdash Cond(s) \wedge V \Vdash V'_{then} \implies (s, V) \xrightarrow{Silent} (Then(s), V'_{then})$$

$$V \Vdash \,!\, Cond(s) \wedge V \Vdash V'_{else} \implies (s, V) \xrightarrow{Silent} (Else(s), V'_{else})$$

- $Type(s) = \mathsf{ASGN}$. In this case (s, V) performs the $Silent$ action and moves to the **then** successor while the valuation is updated as per the assignment. Recall that $LHS(s)$ and $RHS(s)$ are the left and right hand side expressions associated with s. Formally, let $V' \in PropVal(Then(s))$ and e be the expression $\mathcal{WP}[LHS(s)\texttt{=}\ RHS(s)](Concrete(V'))$. Then:

$$V \Vdash e \implies (s, V) \xrightarrow{Silent} (Then(s), V')$$

- $Type(s) = \mathsf{CALL}$. In this case (s, V) performs the $Silent$ action and moves to an initial state of the EFSM $FSM(s)$ corresponding to s. The valuation remains unchanged. Recall that $Init_{FSM(s)}$ denotes the set of initial states of $FSM(s)$. Then:

$$\forall \iota \in Init_{FSM(s)} \centerdot (s, V) \xrightarrow{Silent} (s, V, \iota)$$

Inlined States. Let $s \in Call(\mathcal{C})$, $V \in PropVal(s)$ and (s, V, ι) be an inlined state. Recall that in this case ι must be a state of the EFSM $FSM(s)$ corresponding to s. Also recall that the transitions of $FSM(s)$ are labeled with guarded commands of the form g/α or $g/l := r$. We consider four possible types of outgoing transitions of $FSM(s)$ from the state ι.

- $\iota \xrightarrow{g/\alpha} \iota'$ **and** $\iota' \neq \mathsf{STOP}$. If the valuation V is admissible with the guard g, (s, V, ι) performs action α and moves to the inlined state corresponding to ι'. The valuation remains unchanged. Formally:

$$V \Vdash g \implies (s, V, \iota) \xrightarrow{\alpha} (s, V, \iota')$$

- $\iota \xrightarrow{g/\alpha} \iota'$ **and** $\iota' = \mathsf{STOP}$. If the valuation V is admissible with the guard g, (s, V, ι) performs action α and returns from the library routine call. The new valuation must be admissible with the old one. Formally, let $V' \in PropVal(Then(s))$. Then:

$$V \Vdash g \wedge V \Vdash V' \implies (s, V, \iota) \xrightarrow{\alpha} (Then(s), V')$$

- $\iota \xrightarrow{g/l:=r} \iota'$ **and** $\iota' \neq \mathsf{STOP}$. If the valuation V is admissible with the guard g, (s, V, ι) performs *Silent* and moves to the inlined state corresponding to ι'. The valuation is updated as per the assignment $l := r$. Formally, let $V' \in PropVal(s)$ and e be the expression $\mathcal{WP}[l := r](Concrete(V'))$. Then:

$$V \Vdash g \wedge V \Vdash V' \implies (s, V, \iota) \xrightarrow{Silent} (s, V, \iota')$$

- $\iota \xrightarrow{g/l:=r} \iota'$ **and** $\iota' = \mathsf{STOP}$. If the valuation V is admissible with the guard g, (s, V, ι) performs *Silent* and returns from the library routine call. The valuation is updated as per the assignment $l := r$. Formally, let $V' \in PropVal(Then(s))$ and e be the expression $\mathcal{WP}[l := r](Concrete(V'))$. Then:

$$V \Vdash g \wedge V \Vdash V' \implies (s, V, \iota) \xrightarrow{Silent} (Then(s), V')$$

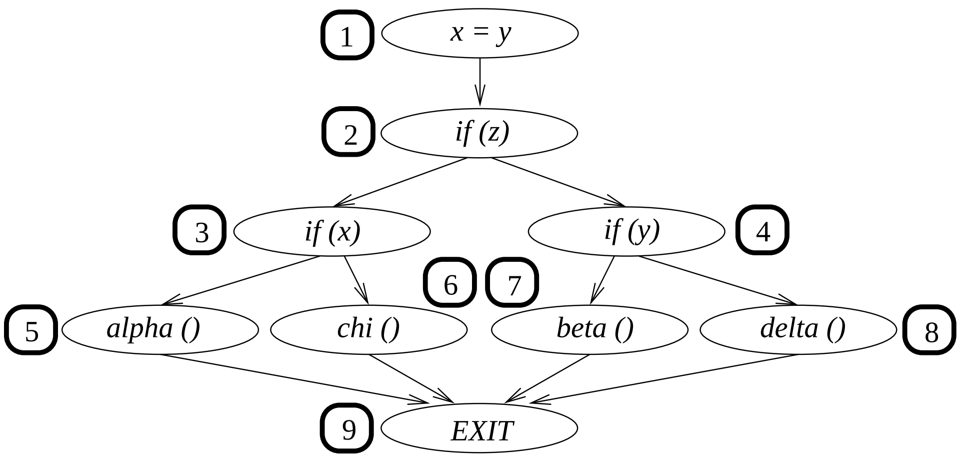


Figure 4.1: The component from Example 9.

Example 10 *Recall the component $\mathcal{C}$ and the context γ from Figure 4.1 and Example 9. The key thing to remember is that according to γ, the library routines $alpha, beta, chi$ and $delta$ perform the actions α, β, χ and δ respectively and terminate. Let Π be the predicate mapping of $\mathcal{C}$ which maps every statement of $\mathcal{C}$ to $\emptyset$. Hence the set of valuations for the propositions corresponding to the set of predicates associated with each statement of $\mathcal{C}$ is simply $\{\perp\}$.*

Figure 4.2 shows the reachable states of the LKS $[\![\mathcal{C}]\!]^{\Pi}_{\gamma}$. Since the valuations are always $\perp$ we omit them for simplicity. Thus the normal states are only labeled by the associated component statement. The inlined states are also labeled by the state of the EFSMs associated with their corresponding library routine calls. In particular these are the initial states of the EFSMs $F_\alpha, F_\chi, F_\beta$ and F_δ shown in Figure 3.4, and are denoted by $Init_\alpha, Init_\beta, Init_\chi$ and $Init_\delta$ respectively.

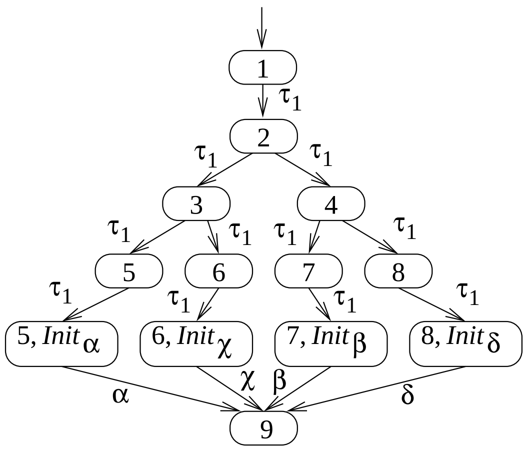


Figure 4.2: The LKS obtained by predicate abstraction of the component in Figure 4.1.

The fact that a predicate abstraction is indeed an abstraction is captured by the following theorem.

Theorem 6 *Let $\mathcal{C}$ be a component, γ be a context for $\mathcal{C}$ and $\Pi : Stmt_{\mathcal{C}} \rightarrow 2^{Expr}$ be a predicate mapping for $\mathcal{C}$ compatible with γ. Then $[\![\mathcal{C}]\!]_\gamma \preccurlyeq [\![\mathcal{C}]\!]_\gamma^\Pi$.*

Proof. The key idea is to define an abstraction mapping from the concrete states of $[\![\mathcal{C}]\!]_\gamma$ to the abstract states of $[\![\mathcal{C}]\!]_\gamma^\Pi$. The mapping must reflect the fact that abstract states are obtained from concrete states by representing stores abstractly using valuations. In other words, the correspondence between concrete and abstract states must be captured by the correspondence between the stores and the valuations.

This is achieved by the function $\mathcal{H} : S_{[\![\mathcal{C}]\!]_\gamma} \rightarrow S_{[\![\mathcal{C}]\!]_\gamma^\Pi}$. Intuitively, $\mathcal{H}$ maps a concrete state s to an abstract state $\widehat{s}$ iff the store associated with s satisfies the concretization of the valuation associated with $\widehat{s}$. Formally, $\mathcal{H}$ is defined as follows:

$$\mathcal{H}(s, \sigma) = (s, V) \centerdot \sigma \models Concrete(V)$$

$$\mathcal{H}(s, \sigma, \iota) = (s, V, \iota) \centerdot \sigma \models Concrete(V)$$

We can prove that $\mathcal{H}$ is an abstraction mapping. Then the result follows from Theorem 5.

□

The fact that simulation is a congruence with respect to parallel composition means that predicate abstraction can be performed on a program component-wise. To present this idea formally we need to extend the notion of predicate mapping and predicate abstraction to programs.

Definition 17 (Program Predicate Mapping) *Let $\mathcal{P} = \langle \mathcal{C}_1, \ldots, \mathcal{C}_n \rangle$ be a program and $\Gamma = \langle \gamma_1, \ldots, \gamma_n \rangle$ be a context for $\mathcal{P}$. A predicate mapping for $\mathcal{P}$ compatible with Γ is a sequence $\langle \Pi_1, \ldots, \Pi_n \rangle$ such that Π_i is a predicate mapping for $\mathcal{C}_i$ compatible with γ_i for $i \in \{1, \ldots, n\}$.*

Definition 18 (Program Predicate Abstraction) *Let $\mathcal{P} = \langle \mathcal{C}_1, \ldots, \mathcal{C}_n \rangle$ be a program, $\Gamma = \langle \gamma_1, \ldots, \gamma_n \rangle$ be a context for $\mathcal{P}$ and $\Pi = \langle \Pi_1, \ldots, \Pi_n \rangle$ be a predicate mapping for $\mathcal{P}$ compatible with Γ. Then the predicate abstraction of $\mathcal{P}$ under Γ and with respect to Π, denoted by $[\![\mathcal{P}]\!]_\Gamma^\Pi$, is the LKS defined as follows:*

$$[\![\mathcal{P}]\!]_\Gamma^\Pi = [\![\mathcal{C}_1]\!]_{\gamma_1}^{\Pi_1} \parallel \cdots \parallel [\![\mathcal{C}_n]\!]_{\gamma_n}^{\Pi_n}$$

Theorem 7 *Let $\mathcal{P}$ be a program, Γ be a context for $\mathcal{P}$ and Π be a predicate mapping for $\mathcal{P}$ compatible with Γ. Then the following holds:*

$$[\![\mathcal{P}]\!]_\Gamma \preccurlyeq [\![\mathcal{P}]\!]_\Gamma^\Pi$$

Proof. Immediately from Theorem 3 and Theorem 6.

□

4.5 Predicate Inference

Recall that the predicate abstraction of a component $\mathcal{C}$ is parameterized by a predicate mapping Π for $\mathcal{C}$. In the framework being presented in this thesis, Π is constructed on the basis of a set of *seed* branches of $\mathcal{C}$ via a process of *predicate inference*. Let us denote by $\mathcal{B}_{\mathcal{C}}$ the set of branch statements of $\mathcal{C}$. Formally, $\mathcal{B}_{\mathcal{C}} = \{s \in Stmt_{\mathcal{C}} \mid \mathbb{T}(s) = \mathsf{BRAN}\}$. Further, for any EFSM F, let $Guard(F)$ denote the set of guards associated with the transitions of F.

Procedure 4.1 PredInfer computes a predicate mapping for component $\mathcal{C}$ that is compatible with a context γ using a set of seed branches. It continues as long as some condition CONTINUE holds.

```
**Algorithm PredInfer**$(\mathcal{C}, \gamma, B)$
**let** $\mathcal{C} = (Stmt, Type, entry, Cond, LHS, RHS, Then, Else)$;
**let** $\gamma = (InitCond, AP, \Sigma, Silent, FSM)$;
**for each** $s \in Stmt$ **let** $\Pi[s] := \{Concrete(p) \mid p \in AP_\gamma\}$;
**for each** $s \in B$ **let** $\Pi[s] := \Pi[s] \cup \{Cond(s)\}$;
**for each** $s \in Call(\mathcal{C})$ **let** $\Pi[s] := \Pi[s] \cup Guard(FSM_\gamma(s))$;
**while** (CONTINUE) **do**
    **for each** $s \in Stmt$ **case** $Type(s)$ **of**
        ASGN : **let** $a = LHS(s) := RHS(s)$;
                $\Pi[s] := \Pi[s] \cup \{\mathcal{WP}[a](p) \mid p \in \Pi[Then(s)]\}$;
        CALL : $\Pi[s] := \Pi[s] \cup \Pi[Then(s)]$;
        BRAN : $\Pi[s] := \Pi[s] \cup \Pi[Then(s)] \cup \Pi[Else(s)]$;
**return** $\Pi$;
```

Algorithm **PredInfer**, presented in Procedure 4.1, takes as input a component $\mathcal{C}$, a context γ for $\mathcal{C}$ and a set of branches $B \subseteq \mathcal{B}_{\mathcal{C}}$. It computes and returns a predicate mapping for $\mathcal{C}$ compatible with γ. Essentially, **PredInfer** works as follows. First it initializes Π with the expressions corresponding to propositions. This is crucial to ensure that the final result is compatible with γ. Then **PredInfer** seeds Π using the branch conditions of the seed branches B and guards from the EFSMs corresponding

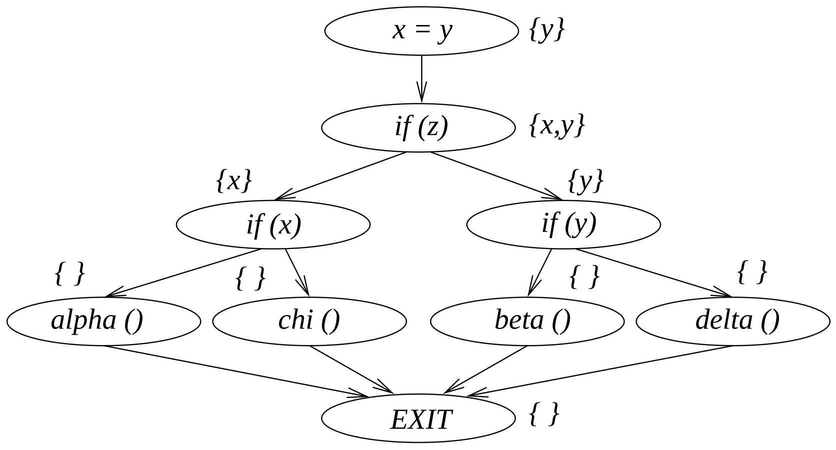


Figure 4.3: The component of Figure 4.1 with each statement labeled by inferred predicates computed by **PredInfer** from the set of seed branches $\{3, 4\}$.

to the call statements of $\mathcal{C}$. Finally it iteratively adds new predicates to the statements of $\mathcal{C}$ on the basis of the predicates that have been already inferred at their successor statements.

Intuitively, whenever a predicate p is inferred at a statement s, the weakest precondition of p is inferred at every predecessor statement of s. If s is an assignment statement, the weakest precondition of p is computed in the obvious manner. Otherwise the weakest precondition of p is p itself. This iterative procedure might not terminate in general but can be terminated on the basis of some criterion (represented in Procedure 4.1 as CONTINUE) such as the size of Π.

Example 11 *Recall the component $\mathcal{C}$ from Figure 4.1 and the context γ from Example 10. Let B be the set of branch statements $\{3, 4\}$, i.e., the two branch statements of $\mathcal{C}$ with conditions x and y respectively. Then Figure 4.3 shows $\mathcal{C}$ with each statement labeled by the set of predicates inferred by* **PredInfer** *when invoked with arguments $\mathcal{C}$, γ and B.*

Chapter 5

Simulation

As mentioned before, in our approach, verification amounts to checking that the specification LKS simulates the implementation LKS. Therefore, we consider simulation in more detail. In this chapter we will write Im to denote the implementation, Sp to denote the specification, Σ to denote Σ_{Sp} and AP to denote AP_{Sp}. Recall that in general the implementation LKS will be obtained by predicate abstraction of a program. In other words, $Im = [\![\mathcal{P}]\!]_{\Gamma}^{\Pi}$ (cf. Definition 18) where Γ is a program context (cf. Definition 12) and Π is a program predicate mapping (cf. Definition 17). Also recall that $\Sigma_{Im} = \Sigma_{Sp}$ and $AP_{Im} = AP_{Sp}$. Further, we will assume that Im and Sp have a single initial state each, i.e., $|Init_{Im}| = |Init_{Sp}| = 1$. The extension to multiple initial states is straightforward. Finally, we will write $Init_{Im}$ and $Init_{Sp}$ to denote the initial state of Im and Sp respectively.

5.1 Simulation Games

Consider an implementation $Im = (S_{Im}, Init_{Im}, AP_{Im}, L_{Im}, \Sigma_{Im}, T_{Im})$ and a specification $Sp = (S_{Sp}, Init_{Sp}, AP_{Sp}, L_{Sp}, \Sigma_{Sp}, T_{Sp})$ such that $\Sigma_{Im} = \Sigma_{Sp}$ and $AP_{Im} = AP_{Sp}$. Suppose we want to determine whether $Im \preccurlyeq Sp$. It is well-known [110] that this can be verified using a two-player game between the implementation Im and the specification Sp. In each round of the game, the implementation poses a challenge and the specification attempts to provide a response. Each player has one pebble located on some state of his LKS which he can move along transitions of his LKS. The location of the pebbles at the start of each round is called a game state, or position, and is denoted by (s_{Im}, s_{Sp}) where s_{Im} and s_{Sp} are the locations of the implementation's and specification's pebbles respectively.

A game position (s_{Im}, s_{Sp}) is said to be *admissible* iff the corresponding implementation and specification states agree on the set of atomic propositions, i.e., $L_{Im}(s_{Im}) = L_{Sp}(s_{Sp})$. We will only consider admissible game positions. Further we will assume that the position $(Init_{Im}, Init_{Sp})$ is admissible. If $(Init_{Im}, Init_{Sp})$ is not admissible, $Im \not\preccurlyeq Sp$ holds trivially. From a given admissible position (s_{Im}, s_{Sp}), the game proceeds as follows:

- **Implementation Challenge.** The implementation picks an action α and a successor state $s'_{Im} \in Succ_{Im}(s_{Im}, \alpha)$ and moves its pebble to s'_{Im}. We denote such a challenge as simply $(s_{Im}, s_{Sp}) \xrightarrow{\alpha} (s'_{Im}, ?)$.

- **Specification Response.** Recall that $L_{Im}(s'_{Im})$ denotes the propositional labeling of state s'_{Im}. The specification responds by moving its pebble to a state s'_{Sp} such that s'_{Sp} is an α-successor of s_{Sp} and s'_{Sp} has the same propositional labeling as s'_{Im}. In other words, $s'_{Sp} \in PSucc_{Sp}(s_{Sp}, \alpha, L_{Im}(s'_{Im}))$. Thus, the specification completes the challenge $(s_{Im}, s_{Sp}) \xrightarrow{\alpha} (s'_{Im}, ?)$ into a transition $(s_{Im}, s_{Sp}) \xrightarrow{\alpha} (s'_{Im}, s'_{Sp})$.

The game continues into the next round from position (s'_{Im}, s'_{Sp}). Note that the response must involve the same action (α) and atomic propositions $(L_{Im}(s'_{Im}))$ as the corresponding challenge. In particular, as per the definition of $PSucc$ (cf. Definition 2), $L_{Sp}(s'_{Sp}) = L_{Im}(s'_{Im})$ and hence position (s'_{Im}, s'_{Sp}) is once again admissible.

- **Winning Condition.** The implementation wins iff the specification is unable to respond to some move of the implementation.

A simulation game is completely defined by Im, Sp and the initial position. Let us denote the simulation game with (s_{Im}, s_{Sp}) as the initial position by **Game**(s_{Im}, s_{Sp}). A position (s_{Im}, s_{Sp}) is called a *winning position* iff Im has a well-defined strategy to win **Game**(s_{Im}, s_{Sp}). The relationship between simulation and simulation games is well-known and is captured by Theorem 8.

Theorem 8 *$Im \preccurlyeq Sp$ iff the implementation Im does not have a strategy to win* **Game**$(Init_{Im}, Init_{Sp})$*, i.e., if $(Init_{Im}, Init_{Sp})$ is not a winning position.*

As the implementation Im can only win after a finite number of moves, it is easy to see that every winning strategy for Im in any simulation game can be described by a finite tree with the following characteristic. For each position (s_{Im}, s_{Sp}), the tree explains how Im should pick a challenge $(s_{Im}, s_{Sp}) \xrightarrow{\alpha} (s'_{Im}, ?)$ in order to ultimately win. Each such tree constitutes a counterexample for the simulation relation and will be referred to as a Counterexample Tree. In general, for each game position, there may exist several ways for Im to challenge and still win eventually. This element of choice leads to the existence of multiple Counterexample Trees.

We will now give a formal framework which describes the game in such a way that Counterexample Trees can be easily extracted. We will write Pos to mean the set of all game positions, i.e., $Pos = S_{Im} \times S_{Sp}$. Let $Challenge$ denote the set of all challenges. We begin by defining the functions $Response : Challenge \rightarrow 2^{Pos}$ which maps a challenge c to the set of all new game positions that can result after Sp has responded to c.

$$Response((s_{Im}, s_{Sp}) \xrightarrow{\alpha} (s'_{Im}, ?)) = \{s'_{Im}\} \times PSucc_{Sp}(s_{Sp}, \alpha, L_{Im}(s'_{Im}))$$

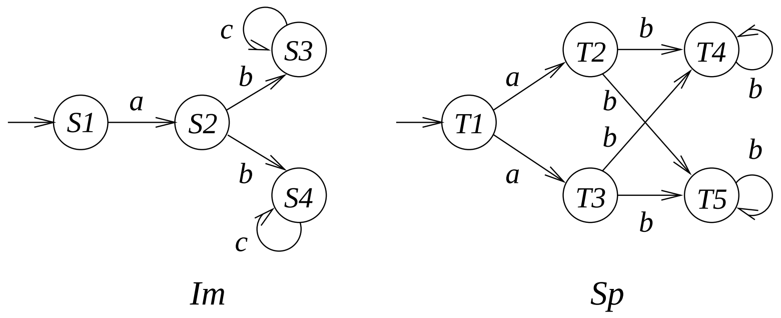


Figure 5.1: Two simple LKSs.

Example 12 *Let Im and Sp be the LKSs from Figure 5.1. From position $(S2, T2)$, Im can pose the following two challenges due to two possible moves from S2 on action b.*

$$(S2, T2) \xrightarrow{b} (S3, ?) \text{ and } (S2, T2) \xrightarrow{b} (S4, ?)$$

For each of these challenges Sp can respond in two ways due to two possible moves from T2 on action b.

$$Response((S2, T2) \xrightarrow{b} (S3, ?)) = \{(S3, T4), (S3, T5)\}$$

$$Response((S2, T2) \xrightarrow{b} (S4, ?)) = \{(S4, T4), (S4, T5)\}$$

5.2 Strategy Trees as Counterexamples

Formally, a Counterexample Tree for $\mathbf{Game}(s_{Im}, s_{Sp})$ is given by a labeled tree (N, E, r, St, Ch) where:

- N, the set of nodes, describes the states *of the winning strategy*
- $E \subseteq N \times N$, the set of edges, describes the transitions between theses states
- $r \in N$ is the root of the tree
- $St : N \to Pos$ maps each tree node to a game position
- $Ch : N \to Challenge$ maps each tree node n to the challenge that Im must pose from position $St(n)$ in accordance with the strategy

Note that, for a given node n, if $St(n) = (s_{Im}, s_{Sp})$ then $Ch(n) = (s_{Im}, s_{Sp}) \xrightarrow{\alpha} (s'_{Im}, ?)$ for some action α and successor state $s'_{Im} \in Succ_{Im}(s_{Im}, \alpha)$. Also, let $Child(n)$ denote the set of children of n. Then the Counterexample Tree (N, E, r, St, Ch) has to satisfy the following conditions:

CE1 The root of the tree is mapped to the initial game state, i.e., $St(r) = (Init_{Im}, Init_{Sp})$.

CE2 The children of a node n cover $Response(Ch(n))$, i.e., the game positions to which the response of Sp can lead. In other words:

$$Response(Ch(n)) = \{St(s) \mid c \in Child(n)\}$$

CE3 The leaves of the tree are mapped to victorious challenges, i.e., challenges from which the specification has no response. In other words, a leaf node l has to obey the following condition: $Response(Ch(l)) = \emptyset$.

Example 13 *Consider again Im and Sp from Figure 5.1. Figure 5.2 shows a* Counterexample Tree *for* **Game**$(S1, T1)$*. Inside each node n we show the challenge $Ch(n)$.*

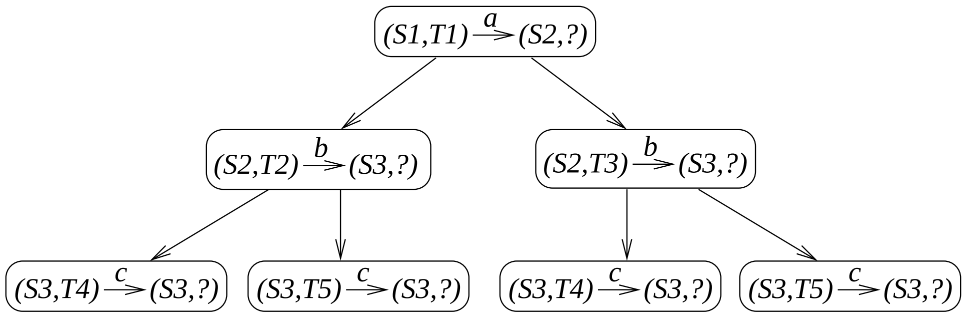


Figure 5.2: Counterexample Tree for a simulation game.

5.3 Checking Simulation

In this section we describe a verification algorithm that checks whether $Im \preccurlyeq Sp$ and computes a Counterexample Tree if $Im \not\preccurlyeq Sp$. Recall that a CounterexampleTree describes a winning strategy for the implementation Im to win the the simulation game. We will first describe the algorithm **ComputeWinPos** which computes the set of winning positions along with their associated challenges; this data is then used to construct a CounterexampleTree.

The Algorithm **ComputeWinPos** is described in Procedure 5.1. It collects the winning positions of Im in the set $WinPos$. Starting with $WinPos = \emptyset$, it adds new winning positions to $WinPos$ until no more winning positions can be found. Note that in the first iteration $WinPos = \emptyset$, and therefore the condition $Response(c) \subseteq WinPos$ amounts to $Response(c) = \emptyset$. The latter condition in turn expresses that c is a victorious challenge.

Procedure 5.1 ComputeWinPos computes the set $WinPos$ of winning positions for the implementation Im; the challenges are stored in Ch.

```
Algorithm ComputeWinPos($Im, Sp$)
$WinPos, Chal := \emptyset$;
forever do
    find challenge $c := (s_{Im}, s_{Sp}) \xrightarrow{\alpha} (s'_{Im}, ?)$ such that $Response(c) \subseteq WinPos$
        // all responses are winning positions
    if not found return ($WinPos$, $Chal$);
    $WinPos := WinPos \cup \{(s_{Im}, s_{Sp})\}$;
    $Chal(s_{Im}, s_{Sp}) := c$;
```

In Procedure 5.2 we present the verification algorithm **SimulCETree** that works as follows: it first invokes **ComputeWinPos** to compute the set $WinPos$ of winning positions. If the initial position $(Init_{Im}, Init_{Sp})$ is not in $WinPos$, then the implementation cannot win the simulation game **Game**$(Init_{Im}, S_{Sp})$. In this case, **SimulCETree** declares that "$Im \not\preccurlyeq Sp$" (recall Theorem 8) and terminates. Otherwise, it invokes algorithm **ComputeStrategy** (presented in Procedure 5.3) to compute a Counterexample Tree for **Game**$(Init_{Im}, Init_{Sp})$.

Theorem 9 *Algorithm* **SimulCETree** *is correct.*

Proof. The correctness of **SimulCETree** follows from the fact that the maximal simulation relation between Im and Sp is a greatest fixed point and **SimulCETree** effectively computes its complement.

$\square$

Algorithm **ComputeStrategy** takes the following as inputs: (i) a winning position (s_{Im}, s_{Sp}), (ii) the set of all winning positions $WinPos$, and (iii) additional challenge information $Chal$. It constructs a Counterexample Tree for the simulation game **Game**(s_{Im}, s_{Sp}) and returns the root of this Counterexample Tree. Note that at the top level, **ComputeStrategy** is invoked by **SimulCETree** with the winning position ($Init_{Im}, Init_{Sp}$). This call therefore returns a Counterexample Tree for the complete simulation game **Game**($Init_{Im}, Init_{Sp}$).

Procedure 5.2 **SimulCETree** checks for simulation, and returns a Counterexample Tree in case of violation.

```
Algorithm SimulCETree($Im, Sp$)
($WinPos$, $Chal$) := ComputeWinPos($Im, Sp$);
if ($Init_{Im}, Init_{Sp}$) $\notin WinPos$ return "$Im \preccurlyeq Sp$";
else return ComputeStrategy($Init_{Im}, Init_{Sp}, WinPos, Chal$);
```

When **ComputeStrategy** is invoked with position (s_{Im}, s_{Sp}), it first creates a root node r and associates position (s_{Im}, s_{Sp}) and challenge $Chal(s_{Im}, s_{Sp})$ with r. It then considers all the positions reachable by responding to $Chal(s_{Im}, s_{Sp})$, i.e., all the positions with which the next round of the game might begin. For each of these positions, **ComputeStrategy** constructs a Counterexample Tree by invoking itself recursively. Finally, **ComputeStrategy** returns r as the root of a new tree, in which the children of r are the roots of the recursively computed trees. Note that if $Response(Chal(s_{Im}, s_{Sp})) = \emptyset$, i.e., if $Chal(s_{Im}, s_{Sp})$ is a victorious challenge, then r becomes a leaf node as expected from condition **CE3** above.

As described in Procedure 5.1, **ComputeWinPos** is essentially a least fixed point algorithm for computing the set of winning positions $WinPos$ and additional challenge information $Chal$. In fact, **ComputeWinPos** can be viewed as the dual of the greatest fixed point algorithm for computing the maximal simulation relation between Im and Sp. Since fixed point computation is quite expensive in practice, **ComputeWinPos** is quite naive and is presented for its simplicity and ease of understanding. In practice, **ComputeWinPos** is implemented by: (i) reducing it to a satisfiability problem for weakly negated HORNSAT (N-HORNSAT) formulas and, (ii) using an N-HORNSAT algorithm that not only checks for satisfiability but also computes $WinPos$ and *Chal*. This procedure is presented in detail in Section 5.4.

5.3.1 Computing Multiple Counterexample Trees

For given Im and Sp, the set of winning positions $WinPos$ computed by **ComputeWinPos** is uniquely defined, i.e., each position (s_{Im}, s_{Sp}) is either the root of some winning strategy (i.e., $(s_{Im}, s_{Sp}) \in WinPos$) or not (i.e, $(s_{Im}, s_{Sp}) \notin WinPos$). There may, however, be multiple winning strategies from position (s_{Im}, s_{Sp}), simply because there may be different challenges Im can pose, which all will ultimately lead to Im's victory.

In the algorithm **ComputeWinPos**, this is reflected by the fact that at each time when the algorithm selects a challenge c, there may be several candidates for c, and only one of them is stored in $Ch(s_{Im}, s_{Sp})$. The challenge information stored in Ch is subsequently used by **ComputeStrategy**, the algorithm which constructs the winning strategy. Thus, depending on **ComputeWinPos**'s choices for the challenges c, **ComputeStrategy** will output different winning strategies. While all these strategies are by construction winning strategies for Im, they may differ in various aspects, for example, the tree size or the actions and states involved. In Section 6.5, we will see that in our experiments, using a set of different winning strategies instead of one indeed helps to save time and memory.

Procedure 5.3 ComputeStrategy recursively computes a winning strategy for showing that $(s_{Im}, s_{Sp}) \in WinPos$; it outputs the root of the strategy tree.

```
**Algorithm ComputeStrategy**($s_{Im}, s_{Sp}, WinPos, Chal$);
    // $(s_{Im}, s_{Sp})$ is a winning position in $WinPos$
**create new tree node** $r$ with $St(r) := (s_{Im}, s_{Sp})$ and $Ch(r) := Chal(s_{Im}, s_{Sp})$;
**for all** $(c_{Im}, c_{Sp}) \in Response(Chal(s_{Im}, s_{Sp}))$
    **create tree edge** $r \longrightarrow$ **ComputeStrategy**($c_{Im}, c_{Sp}, WinPos, Chal$);
**return** $r$;
```

5.4 Simulation using N-HORNSAT

Given two LKSs Im and Sp we can verify whether $Im \preccurlyeq Sp$ efficiently by reducing the problem to an instance of Boolean satisfiability [103] or SAT. Interestingly the SAT instances produced by this method always belong to a restricted class of SAT formulas known as the *weakly negated* HORN formulas. The satisfiability problem for such formulas is also known as N-HORNSAT. In contrast to general SAT (which has no known polynomial time algorithm), N-HORNSAT can be solved in linear time [57].

In this section we present the N-HORNSAT based simulation check algorithm. We also describe a procedure to compute the set of winning positions $WinPos$ and associated challenge information *Chal* that can be used subsequently by **ComputeStrategy** to compute a Counterexample Tree in case the simulation is found not to exist. We begin with a few preliminary definitions.

5.4.1 Definitions

A *literal* is either a boolean variable (in which case it is said to be positive) or its negation (in which case it is said to be negative). A *clause* is a disjunction of literals, i.e., a formula of the form $(l_1 \vee \cdots \vee l_m)$ where l_i is a literal for $1 \leq i \leq m$. A formula is said to be in conjunctive normal form (CNF) iff it is a conjunction of clauses, i.e., of the form $(c_1 \wedge \cdots \wedge c_n)$ where c_i is a clause for $1 \leq i \leq n$.

Recall that $\mathbb{B}$ denotes the set of Boolean values $\{\textsc{true}, \textsc{false}\}$. A *valuation* is a function from boolean variables to $\mathbb{B}$. A valuation $\mathcal{V}$ automatically induces a function $\mathcal{V}$ from literals to $\mathbb{B}$ as follows: (i) $\mathcal{V}(l) = \mathcal{V}(b)$ if l is of the form b and (ii) $\mathcal{V}(l) = \neg\mathcal{V}(b)$ if l is of the form $\neg b$. A valuation $\mathcal{V}$ automatically induces a function $\mathcal{V}$ from clauses to $\mathbb{B}$ as follows. Let $c = (l_1 \vee \cdots \vee l_m)$ be a clause. Then $\mathcal{V}(c) = \bigvee_{i=1}^{m} \mathcal{V}(l_i)$. In the same spirit, a valuation $\mathcal{V}$ automatically induces a function $\mathcal{V}$ from CNF formula to $\mathbb{B}$ as follows. Let $\phi = (c_1 \wedge \cdots \wedge c_n)$ be a CNF formula. Then $\mathcal{V}(\phi) = \bigwedge_{i=1}^{n} \mathcal{V}(c_i)$. A CNF formula ϕ is said to be satisfiable iff there exists a valuation $\mathcal{V}$ such that $\mathcal{V}(\phi) = \textsc{true}$.

A CNF formula $(c_1 \wedge \cdots \wedge c_n)$ is said to be a weakly negated HORN (N-HORN) formula iff each c_i contains at most one negative literal for $1 \leq i \leq n$. The problem of checking the satisfiability of an arbitrary N-HORN formula is known as N-HORNSAT. There exists a well-known algorithm [4] for solving the N-HORNSAT problem that requires linear time and space in the size of the input formula. We are now ready to present the N-HORNSAT based simulation checking algorithm.

5.4.2 Reducing Simulation to N-HORNSAT

Let Im and Sp be two LKSs such that $\Sigma_{Im} = \Sigma_{Sp}$. Our goal is to create an N-HORN formula $\phi(Im, Sp)$ such that $\phi(Im, Sp)$ is satisfiable iff $Im \preccurlyeq Sp$. For each $s_{Im} \in S_{Im}$ and $s_{Sp} \in S_{Sp}$ we introduce a boolean variable that we denote $\overline{WP}(s_{Im}, s_{Sp})$. Intuitively, $\overline{WP}(s_{Im}, s_{Sp})$ stands for the proposition that (s_{Im}, s_{Sp}) is **not** a winning position. We then generate a set of clauses that constrain the various boolean variables in accordance with the rules of a simulation game.

In particular suppose $\overline{WP}(s_{Im}, s_{Sp})$ is TRUE. Then (s_{Im}, s_{Sp}) is not a winning position. Now suppose $s_{Im} \xrightarrow{\alpha}_{Im} s'_{Im}$. Then according to the rules of a simulation game, there must exist a successor game state which is also not a winning position. In other words, there must exist some state s'_{Sp} such that: (i) $s_{Sp} \xrightarrow{\alpha}_{Sp} s'_{Sp}$, (ii) $L_{Sp}(s'_{Sp}) = L_{Im}(s'_{Im})$, and (iii) (s'_{Im}, s'_{Sp}) is not a winning position. But this argument can be expressed formally by the following clause:

$$\overline{WP}(s_{Im}, s_{Sp}) \implies \bigvee_{s'_{Sp} \in PSucc_{Sp}(s_{Sp}, \alpha, L_{Im}(s'_{Im}))} \overline{WP}(s'_{Im}, s'_{Sp})$$

In essence, most of our target formula $\phi(Im, Sp)$ is composed of such clauses (which we shall call the *transition* clauses), one for each appropriate choice of s_{Im}, s_{Sp}, α and s'_{Im}. As a special case, when $PSucc_{Sp}(s_{Sp}, \alpha, L_{Im}(s'_{Im})) = \emptyset$, the generated clause is simply $\neg\overline{WP}(s_{Im}, s_{Sp})$. In addition to the transition clauses, $\phi(Im, Sp)$ contains a single clause $\overline{WP}(Init_{Im}, Init_{Sp})$ which expresses the constraint that $(Init_{Im}, Init_{Sp})$ is not a winning position. Let us call this clause the *initial* clause. The algorithm to generate $\phi(Im, Sp)$ is called **GenerateHORN** and is shown in Procedure 5.4. Note that the generated $\phi(Im, Sp)$ is a N-HORN formula.

Procedure 5.4 GenerateHORN to generate $\phi(Im, Sp)$.

```
Algorithm GenerateHORN ($Im, Sp$)
for each $s_{Im} \in S_{Im}$, for each $s_{Sp} \in S_{Sp}$
    for each $\alpha \in \Sigma_{Im}$, for each $s'_{Im} \in Succ_{Im}(s_{Im}, \alpha)$
        output clause $\overline{WP}(s_{Im}, s_{Sp}) \implies \bigvee_{s'_{Sp} \in PSucc_{Sp}(s_{Sp}, \alpha, L_{Im}(s'_{Im}))} \overline{WP}(s'_{Im}, s'_{Sp})$
            //generate transition clause
output clause $\overline{WP}(Init_{Im}, Init_{Sp})$    //generate initial clause
```

The above method of checking simulation via N-HORNSAT is well-known [103]. Further, N-HORNSAT can be solved in linear time and space [57]. This yields extremely efficient algorithms for checking simulation between two LKSs. In addition, our CEGAR framework requires a counterexample if the simulation check fails. As part of MAGIC we have implemented an extended version of the N-HORNSAT algorithm presented by Ausiello and Italiano [4] to achieve precisely this goal. In other words, not only does our algorithm check for satisfiability of N-HORN formulas, but it also constructs a counterexample for the simulation relation if the formula is found to be unsatisfiable. To the best of my knowledge, this is the first attempt to construct counterexamples in the context of simulation using SAT procedures.

5.4.3 Computing $WinPos$ and *Chal*

Recall that in order to check simulation between Im and Sp, we first construct an N-HORNSAT formula $\phi(Im, Sp)$ such that $\phi(Im, Sp)$ is satisfiable iff $Im \preccurlyeq Sp$. In this section we describe an algorithm to check for the satisfiability of $\phi(Im, Sp)$. We also describe a procedure to compute the set of winning positions $WinPos$ and associated challenge information *Chal* that can be used subsequently by **ComputeStrategy** to compute a Counterexample Tree.

In the rest of this section we shall denote $\phi(Im, Sp)$ as simply ϕ. The satisfiability check occurs in two phases. In the first phase, a directed hypergraph, $\mathcal{HG}$ is constructed on the basis of the clauses in ϕ. The nodes of $\mathcal{HG}$ correspond to the Boolean variables in ϕ. We shall denote the node corresponding to Boolean variable b as simply $\mathcal{N}_b$. Additionally there are two special nodes called $\mathcal{N}_{\text{TRUE}}$ and $\mathcal{N}_{\text{FALSE}}$. The edges of $\mathcal{HG}$ are constructed as follows:

- For each clause of the form $\neg b$ in ϕ, we add a hyper-edge from the hyper-node $\{\mathcal{N}_{\text{FALSE}}\}$ to node $\mathcal{N}_b$.
- For each clause of the form $(b_1 \vee \cdots \vee b_k)$ in ϕ, we add a hyper-edge from the

hyper-node $\{\mathcal{N}_{b_1}, \ldots, \mathcal{N}_{b_k}\}$ to node $\mathcal{N}_{\text{TRUE}}$.

- Finally, for each clause of the form $(\neg b_0 \vee b_1 \vee \cdots \vee b_k)$ in ϕ, we add a hyper-edge from the hyper-node $\{\mathcal{N}_{b_1}, \ldots, \mathcal{N}_{b_k}\}$ to node $\mathcal{N}_{b_0}$.

Essentially the edges of $\mathcal{HG}$ represent the logical flow of *falsehood* as forced by the clauses of ϕ. Suppose we define the notion of reachability of nodes in $\mathcal{HG}$ from $\mathcal{N}_{\text{FALSE}}$ as follows: (i) $\mathcal{N}_{\text{FALSE}}$ is reachable from $\mathcal{N}_{\text{FALSE}}$, and (ii) a hyper-node $\{\mathcal{N}_{b_1}, \ldots, \mathcal{N}_{b_k}\}$ is reachable from $\mathcal{N}_{\text{FALSE}}$ iff each of the nodes $\mathcal{N}_{b_1}, \ldots, \mathcal{N}_{b_k}$ is reachable from $\mathcal{N}_{\text{FALSE}}$, and (iii) a node n is reachable from $\mathcal{N}_{\text{FALSE}}$ iff there is a hyper-node h such that h is reachable from $\mathcal{N}_{\text{FALSE}}$ and there is a hyper-edge from h to n.

In the second phase of our N-HORNSAT satisfiability algorithm, we compute the set of nodes of $\mathcal{HG}$, denoted by *Reach*, that are reachable from $\mathcal{N}_{\text{FALSE}}$. *Reach* can be computed using linear time and space in the size of $\mathcal{HG}$ (and hence ϕ). It can be shown that a node $\mathcal{N}_{\overline{WP}(s_{Im}, s_{Sp})}$ belongs to *Reach* iff in order to satisfy ϕ the variable $\overline{WP}(s_{Im}, s_{Sp})$ must be assigned FALSE. As a consequence, ϕ is satisfiable iff $\mathcal{N}_{\text{TRUE}} \notin$ *Reach*. In addition, *Reach* has the following significance. Recall that the boolean variables in ϕ are of the form $\overline{WP}(s_{Im}, s_{Sp})$. It can be shown that the following holds:

$$\forall s_{Im} \in S_{Im} \centerdot \forall s_{Sp} \in S_{Sp} \centerdot \mathcal{N}_{\overline{WP}(s_{Im}, s_{Sp})} \in \mathit{Reach} \iff (s_{Im}, s_{Sp}) \in WinPos$$

In other words, the elements in *Reach*$\setminus\{\mathcal{N}_{\text{TRUE}}, \mathcal{N}_{\text{FALSE}}\}$ are exactly those nodes that correspond to boolean variables $\overline{WP}(s_{Im}, s_{Sp})$ such that (s_{Im}, s_{Sp}) is a winning position. Therefore, once *Reach* has been computed it is trivial to compute $WinPos$. To compute *Chal* we note the following. Suppose that a node $\mathcal{N}_{\overline{WP}(s_{Im}, s_{Sp})}$ gets added to *Reach* at some point. This means that the following two conditions must hold:

CH1 A transition clause of the following form was added to ϕ at line 3 of Procedure 5.4.

$$\overline{WP}(s_{Im}, s_{Sp}) \implies \bigvee_{s'_{Sp} \in PSucc_{Sp}(s_{Sp}, \alpha, L_{Im}(s'_{Im}))} \overline{WP}(s'_{Im}, s'_{Sp})$$

CH2 Every node of $\mathcal{HG}$ of the form $\mathcal{N}_{\overline{WP}(s'_{Im}, s'_{Sp})}$ such that $s'_{Sp} \in PSucc_{Sp}(s_{Sp}, \alpha, L_{Im}(s'_{Im}))$ must already be contained in *Reach*. In other words every such (s'_{Im}, s'_{Sp}) must be a winning position.

From conditions **CH1** and **CH2** above, it is clearly appropriate to set $\mathit{Chal}(s_{Im}, s_{Sp}) := (s_{Im}, s_{Sp}) \xrightarrow{\alpha} (s'_{Im}, ?)$. Therefore, as soon as $\mathcal{N}_{\overline{WP}(s_{Im}, s_{Sp})}$ gets added to *Reach*, one can compute the clause postulated by condition **CH1** above and set $\mathit{Chal}(s_{Im}, s_{Sp})$ appropriately using this clause. Since this can be done for every node added to *Reach*, we can effectively compute the challenge information *Chal* associated with every winning position in $WinPos$.

5.5 Witnesses as Counterexamples

Counterexample Trees provide a natural notion of counterexamples to simulation conformance. However, we introduce witness LKSs since they enable us to prove some key results more easily. In the rest of this thesis we will refer to witness LKSs as Counterexample Witnesses. Recall from Theorem 2 that a Counterexample Witness to $Im \not\preccurlyeq Sp$ is an LKS CW such that: (i) $CW \preccurlyeq Im$ and (ii) $CW \not\preccurlyeq Sp$. Fortunately a Counterexample Witness can be obtained from a Counterexample Tree in a straightforward manner using the recursive algorithm **TreeToWitness**, presented in Procedure 5.5.

The inputs to **TreeToWitness** are a Counterexample Tree CT, a node n of CT,

Procedure 5.5 TreeToWitness computes a Counterexample Witness corresponding to a Counterexample Tree CT.

```
Algorithm TreeToWitness($CT, n, Im, s$)
let $CT = (N, E, r, St, Ch)$ and $Ch(n) = (s_{Im}, s_{Sp}) \xrightarrow{\alpha} (s'_{Im}, ?)$;
create state $s'$;
$S := \{s'\}$; $T := \{s \xrightarrow{\alpha} s'\}$; $L(s') := L_{Im}(s'_{Im})$;
for each $c \in Child(n)$
    $(S', T', L') :=$ TreeToWitness($CT, c, Im, s'$);
    $S := S \cup S'$; $T := T \cup T'$; $L := L \cup L'$;
return $(S, T, L)$;
```

the implementation Im, and a state s. **TreeToWitness** computes and returns the set of states, transitions and propositional labellings of the Counterexample Witness corresponding to the subtree of CT rooted at n. Intuitively the state s can be viewed as the initial state of the computed Counterexample Witness. We note any Counterexample Witness is a tree if we view its states as nodes and its transitions as edges.

Procedure 5.6 SimulWitness checks for simulation, and returns a Counterexample Witness in case of violation.

```
Algorithm SimulWitness($Im, Sp$)
if (SimulCETree($Im, Sp$) = "$Im \preccurlyeq Sp$") return "$Im \preccurlyeq Sp$";
else let $CT :=$ SimulCETree($Im, Sp$);
create state $init$; $(S, T, L) :=$ TreeToWitness($CT, r_{CT}, Im, init$);
$S := S \cup \{init\}$; $L(init) := L_{Im}(Init_{Im})$;
return $(S, \{init\}, AP_{Im}, L, \Sigma_{Im}, T)$;
```

Algorithm **SimulWitness**, presented in Procedure 5.6, is similar to **SimulCETree** except that it returns a Counterexample Witness as a counterexample. In fact, it first invokes **SimulCETree**. If **SimulCETree** returns "$Im \preccurlyeq Sp$", so does **SimulWitness**. Otherwise, it invokes **TreeToWitness** to compute and return the Counterexample Witness corresponding to the Counterexample Tree returned by **SimulCETree**. The following two results prove the correctness of **SimulWitness**.

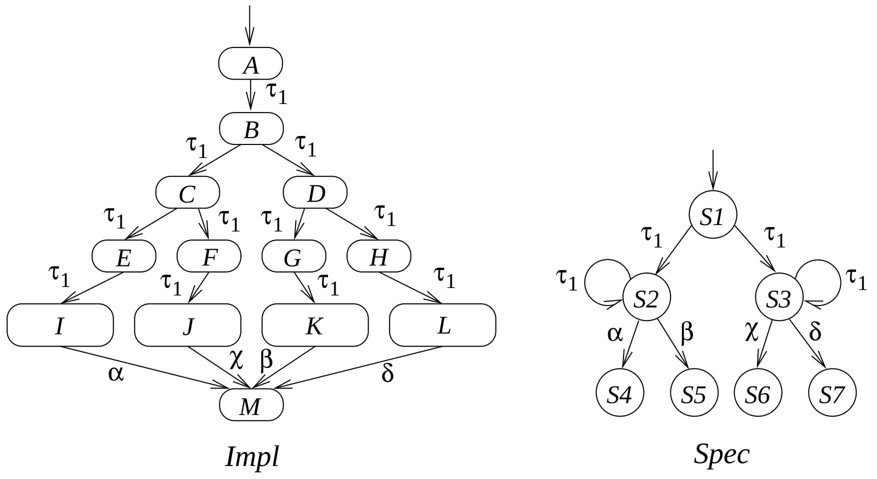
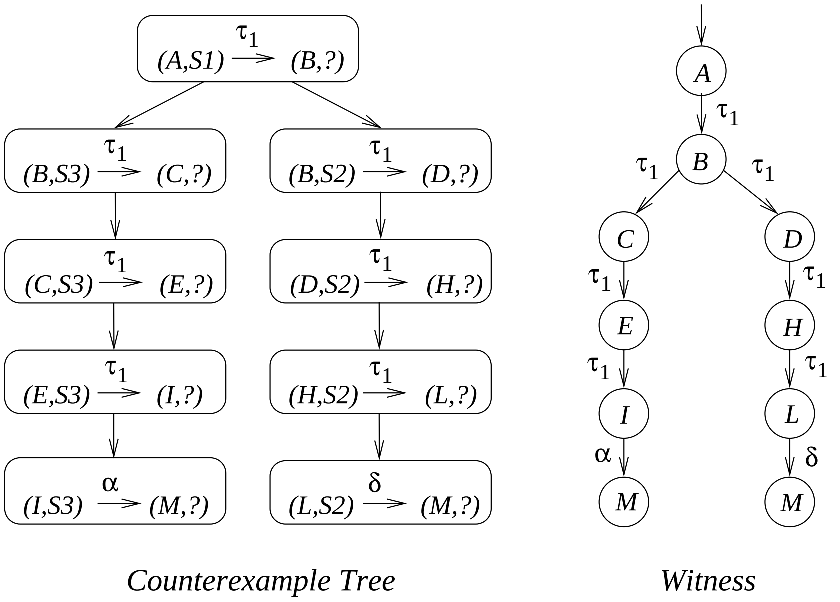


Figure 5.3: An implementation Im and a specification Sp. Im is the LKS from Figure 4.2.

Example 14 *Figure 5.3 once again shows the LKS Im obtained by predicate abstraction in Chapter 4 (cf. Figure 4.2). It also shows a specification LKS Sp. Note that $Im \not\preccurlyeq Sp$. Figure 5.4 shows a* CounterexampleTree *returned by* **SimulCETree** *when invoked with Im and Sp and also the* CounterexampleWitness *obtained from the* CounterexampleTree *by invoking* **TreeToWitness**. *For ease of understanding, each state of the* CounterexampleWitness *is labeled by the corresponding state of Im which simulates it.*

Theorem 10 *Let CW be a* Counterexample Witness *returned by* **SimulWitness**(Im, Sp). *Then the following holds: (i) $CW \preccurlyeq Im$, and (ii) $CW \not\preccurlyeq Sp$.*

Proof. Recall that **SimulWitness** invokes **TreeToWitness** in order to construct the Counterexample Witness CW. We begin by defining a mapping $\mathcal{H} : S_{CW} \to S_{Im}$



(A,S1) $\xrightarrow{\tau_1}$ (B,?)
(B,S3) $\xrightarrow{\tau_1}$ (C,?)
(B,S2) $\xrightarrow{\tau_1}$ (D,?)
(C,S3) $\xrightarrow{\tau_1}$ (E,?)
(D,S2) $\xrightarrow{\tau_1}$ (H,?)
(E,S3) $\xrightarrow{\tau_1}$ (I,?)
(H,S2) $\xrightarrow{\tau_1}$ (L,?)
(I,S3) $\xrightarrow{\alpha}$ (M,?)
(L,S2) $\xrightarrow{\delta}$ (M,?)
Counterexample Tree
A
B
C
D
E
H
I
L
M
M
Witness

Figure 5.4: A CounterexampleTree and CounterexampleWitness corresponding to the simulation game between Im and Sp from Figure 5.3. Each state of the CounterexampleWitness is labeled by the corresponding state of Im which simulates it.

from the states of CW to the states of the implementation Im as follows. Suppose **TreeToWitness** was invoked by **SimulWitness** with arguments (CT, n, Im, s) where CT is a Counterexample Tree for **Game**(Im, Sp). Let $CT = (N, E, r, St, Ch)$ and $St(n) = (s_{Im}, s_{Sp})$. Then $\mathcal{H}(s) = s_{Im}$. It is easy to see that $\mathcal{H}$ is well-defined. Further one can show that $\mathcal{H}$ is also an abstraction mapping. This completes the proof of $CW \preccurlyeq Im$.

To prove that $CW \not\preccurlyeq Sp$ we show how to create a Counterexample Tree $CT' = (N', E', r', St', Ch')$ for **Game**(CW, Sp). This is done on the basis of the Counterexample Tree $CT = (N, E, r, St, Ch)$ for **Game**(Im, Sp). Formally, the components of CT' obey the following constraints:

- The nodes, edges and root of CT' are the same as those of CT.

$$N' = N \qquad E' = E \qquad r' = r$$

- The state labeling of CT' is defined as follows. Suppose **TreeToWitness** was invoked with arguments (CT, n, Im, s). Recall that $CT = (N, E, r, St, Ch)$. Let $St(n) = (s_{Im}, s_{Sp})$. Then $St'(n) = (s, s_{Sp})$.

- The challenge labeling of CT' is defined as follows. Suppose **TreeToWitness** was invoked by **SimulWitness** with arguments (CT, n, Im, s). Recall that $CT = (N, E, r, St, Ch)$. Suppose $Ch(n) = (s_{Im}, s_{Sp}) \xrightarrow{\alpha} (s'_{Im}, ?)$ and s' was the new state created during this invocation. Then $Ch'(n) = (s, s_{Sp}) \xrightarrow{\alpha} (s', ?)$.

Finally, we show that CT' is a valid Counterexample Tree for **Game**(CW, Sp). This can be done by showing that CT' satisfies conditions **CE1–CE3** described in Section 5.2.

$\square$

Theorem 11 *Algorithm* **SimulWitness** *is correct.*

Proof. By Theorem 9 and Theorem 10.

$\square$

Chapter 6

Refinement

In this chapter we describe the process of counterexample validation and abstraction refinement in the context of simulation. Once a Counterexample Witness CW has been constructed, we need to perform two steps: (i) check if CW is a valid Counterexample Witness, and (ii) if CW is spurious then refine Im so as to prevent CW from reappearing in future iterations. We now describe these two steps in more detail. We end this chapter with a description of the complete CEGAR algorithm in the context of simulation conformance.

6.1 Witness Validation

Recall that checking the validity of CW means verifying whether CW is a valid Counterexample Witness for $\mathbf{Game}([\![\mathcal{P}]\!]_\Gamma, Sp)$, where $[\![\mathcal{P}]\!]_\Gamma$ is the concrete program semantics and Sp is the specification. Further this means we have to show that the following two conditions are satisfied: (i) $CW \preccurlyeq [\![\mathcal{P}]\!]_\Gamma$ and (ii) $CW \not\preccurlyeq Sp$. Since CW is a Counterexample Witness for $\mathbf{Game}(Im, Sp)$, the condition $CW \not\preccurlyeq Sp$ is automatically satisfied. Hence we have to only verify that $CW \preccurlyeq [\![\mathcal{P}]\!]_\Gamma$.

In this section we present a compositional (component-wise) algorithm to achieve this goal. We assume that $\mathcal{P} = \langle \mathcal{C}_1, \ldots, \mathcal{C}_n \rangle$ and that $\Gamma = \langle \gamma_1, \ldots, \gamma_n \rangle$ is the context for $\mathcal{P}$ which was used for the simulation check. We begin with the notion of projection of an LKS on an alphabet.

Definition 19 (LKS Projection) *Let $M = (S, Init, AP, L, \Sigma, T)$ be an LKS such that $\tau \notin \Sigma$, and $\overline{\Sigma} \subseteq \Sigma$ be an alphabet. Then the projection of M on $\overline{\Sigma}$, denoted by $M \downharpoonleft \overline{\Sigma}$, is the LKS $M' = (S, Init, AP, L', \Sigma \cup \{\tau\}, T')$ such that L' and T' are defined as follows:*

- $\forall s \in S' \centerdot L'(s) = L(s) \cap AP$
- $\forall (s, \alpha, s') \in T \centerdot \alpha \in \overline{\Sigma} \implies (s, \alpha, s') \in T'$
- $\forall (s, \alpha, s') \in T \centerdot \alpha \notin \overline{\Sigma} \implies (s, \tau, s') \in T'$

Note that $M \downharpoonleft \overline{\Sigma}$ has the same states, initial states and atomic proposition as M.

Let $\gamma = (InitCond, AP, \Sigma, Silent, FSM)$ be a context for a component $\mathcal{C}$. Then we write $M \downharpoonleft \gamma$ to mean $M \downharpoonleft (AP \cup \{Silent\})$. Let CW be a CounterexampleWitness LKS. Intuitively, the projection $CW \downharpoonleft \gamma$ retains the contribution of $\mathcal{C}$ toward CW and eliminates the contributions of the other components. We note that since CW has a tree structure, so does $CW \downharpoonleft \gamma$. Also note that $\tau \notin \Sigma_{CW}$ but $\tau \in \Sigma_{CW \downharpoonleft \gamma}$. This fact enables us to derive Theorem 12 which will allow us to verify $CW \preccurlyeq [\![\mathcal{P}]\!]_\Gamma$ in a component-wise manner using weak simulation and the projections of CW. Recall that we write $M_1 \precsim M_2$ to mean that LKS M_1 is weakly simulated by LKS M_2.

Theorem 12 (Compositional Validation) *Let* $M_1 = (S_1, Init_1, AP_1, L_1, \Sigma_1, T_1)$ *and* $M_2 = (S_2, Init_2, AP_2, L_2, \Sigma_2, T_2)$ *be two LKSs such that* $\tau \notin \Sigma_1$ *and* $\tau \notin \Sigma_2$. *Let* $M = (S, Init, AP, L, \Sigma, T)$ *be another LKS such that* $\Sigma = \Sigma_1 \cup \Sigma_2$. *Then the following holds:*

$$M \preccurlyeq M_1 \parallel M_2 \iff ((M \downharpoonleft \Sigma_1) \precsim M_1) \bigwedge ((M \downharpoonleft \Sigma_2) \precsim M_2)$$

Proof. For the forward implication, let $\mathcal{R} \subseteq S \times (S_1 \times S_2)$ be a simulation relation such that:

$$\forall s \in Init \centerdot \exists s_1 \in Init_1 \centerdot \exists s_2 \in Init_2 \centerdot s\mathcal{R}(s_1, s_2) \tag{6.1}$$

Recall that the set of states of $M \downharpoonleft \Sigma_1$ is S and that the set of initial states of $M \downharpoonleft \Sigma_1$ is $Init$. Define relation $\mathcal{R}_1 \subseteq S \times S_1$ as follows:

$$\mathcal{R}_1 = \{(s, s_1) \mid \exists s_2 \centerdot s\mathcal{R}\,(s_1, s_2)\} \tag{6.2}$$

From 6.1 and 6.2 we have directly:

$$\forall s \in Init \centerdot \exists s_1 \in Init_1 \centerdot (s, s_1) \in \mathcal{R}_1 \tag{6.3}$$

Now we need to prove that $\mathcal{R}_1$ is a weak simulation. Consider any two states $s \in S$ and $s_1 \in S_1$ such that $(s, s_1) \in \mathcal{R}_1$. From 6.2 we know that:

$$\exists s_2 \centerdot s\mathcal{R}\,(s_1, s_2) \tag{6.4}$$

Suppose that $M \downharpoonleft \Sigma_1$ contains the following transition where $\alpha \in \Sigma_1$:

$$s \xrightarrow{\alpha} s' \tag{6.5}$$

The following commutative diagram explains the basic idea behind the proof.

$$\begin{tikzcd}
s \arrow[r, "\alpha"] \arrow[d, "\mathcal{R}"'] & s' \arrow[d, "\mathcal{R}"] \\
(s_1, s_2) \arrow[r, "\alpha"] \arrow[d, "\mathcal{R}_1"'] & (s_1', s_2') \arrow[d, "\mathcal{R}_1"] \\
s_1 \arrow[r, "\alpha"] & s_1'
\end{tikzcd}$$

From 6.5 and Definition 19 we can conclude that M contains the following transition:

$$s \xrightarrow{\alpha} s' \tag{6.6}$$

Since $\mathcal{R}$ is a simulation relation, from 6.4 and 6.6 we know that:

$$\exists s_1' \in S_1 \bullet \exists s_2' \in S_2 \bullet (s_1, s_2) \xrightarrow{\alpha} (s_1', s_2') \bigwedge s'\mathcal{R}(s_1', s_2') \tag{6.7}$$

From 6.2 and the fact that $s'\mathcal{R}(s_1', s_2')$ (cf. 6.7), we have:

$$(s', s_1') \in \mathcal{R}_1 \tag{6.8}$$

Then from the fact that $(s_1, s_2) \xrightarrow{\alpha} (s_1', s_2')$ (cf. 6.7) and that $\alpha \in \Sigma_1$, we have:

$$s_1 \xrightarrow{\alpha} s_1' \tag{6.9}$$

From 6.8 and 6.9 we conclude that $\mathcal{R}_1$ is a weak simulation relation.

Now suppose that $M \rfloor \Sigma_1$ contains the following transition:

$$s \xrightarrow{\tau} s' \tag{6.10}$$

From 6.10 and Definition 19 we can conclude that there exists $\beta \notin \Sigma_1$ such that M contains the following transition:

$$s \xrightarrow{\beta} s' \tag{6.11}$$

The following commutative diagram explains the basic idea behind the proof.

$$\boxed{\begin{tikzcd} s \arrow[r, "\beta"] \arrow[d, "\mathcal{R}"'] & s' \arrow[d, "\mathcal{R}"] \\ (s_1, s_2) \arrow[r, "\beta"] \arrow[d, "\mathcal{R}_1"'] & (s_1, s_2') \arrow[d, "\mathcal{R}_1"] \\ s_1 \arrow[r] & s_1 \end{tikzcd}}$$

Since $\mathcal{R}$ is a simulation relation, from 6.4 and 6.11 and the fact that $\beta \notin \Sigma_1$ and the definition of parallel composition we know that:

$$\exists s_2' \in S_2 \bullet (s_1, s_2) \xrightarrow{\beta} (s_1, s_2') \bigwedge s' \mathcal{R} (s_1, s_2') \tag{6.12}$$

From 6.2 and the fact that $s' \mathcal{R} (s_1, s_2')$ (cf. 6.12), we have:

$$(s', s_1) \in \mathcal{R}_1 \tag{6.13}$$

From 6.13 we can again conclude that $\mathcal{R}_1$ is a weak simulation relation. This completes the proof that $(M \downharpoonleft \Sigma_1) \precsim M_1$. We can show in a similar manner that $(M \downharpoonleft \Sigma_2) \precsim M_2$ and hence we have the proof of the forward implication.

For the reverse implication let $\mathcal{R}_1 \subseteq S \times S_1$ be a weak simulation relation such that:

$$\forall s \in \mathit{Init} \bullet \exists s_1 \in \mathit{Init}_1 \bullet (s, s_1) \in \mathcal{R}_1 \tag{6.14}$$

Similarly let $\mathcal{R}_2 \subseteq S \times S_2$ be a weak simulation relation such that:

$$\forall s \in \mathit{Init} \bullet \exists s_2 \in \mathit{Init}_2 \bullet (s, s_2) \in \mathcal{R}_2 \tag{6.15}$$

Define relation $\mathcal{R} \subseteq S \times (S_1 \times S_2)$ as follows:

$$\mathcal{R} = \{(s, (s_1, s_2)) \mid (s, s_1) \in \mathcal{R}_1 \wedge (s, s_2) \in \mathcal{R}_2\} \tag{6.16}$$

From 6.14, 6.15 and 6.16 we get immediately the following:

$$\forall s \in Init \centerdot \exists s_1 \in Init_1 \centerdot \exists s_2 \in Init_2 \centerdot s\mathcal{R}(s_1, s_2) \tag{6.17}$$

Now we need to show that $\mathcal{R}$ is a simulation relation. Consider any states $s \in S$, $s_1 \in S_1$ and $s_2 \in S_2$ such that:

$$s\mathcal{R}(s_1, s_2) \tag{6.18}$$

From 6.16 and 6.18 we can conclude that:

$$(s, s_1) \in \mathcal{R}_1 \bigwedge (s, s_2) \in \mathcal{R}_2 \tag{6.19}$$

Now suppose that M contains the following transition:

$$s \xrightarrow{\alpha} s' \tag{6.20}$$

Then we need to show the following:

$$\exists s_1' \in S_1 \centerdot \exists s_2' \in S_2 \centerdot (s_1, s_2) \xrightarrow{\alpha} (s_1', s_2') \bigwedge s'\mathcal{R}(s_1', s_2') \tag{6.21}$$

From 6.16 and 6.21 it is clear that we need to find an $s_1' \in S_1$ and an $s_2' \in S_2$ such that the following holds:

$$(s_1, s_2) \xrightarrow{\alpha} (s_1', s_2') \bigwedge (s', s_1') \in \mathcal{R}_1 \bigwedge (s', s_2') \in \mathcal{R}_2 \tag{6.22}$$

We will first show the existence of such an s_1'. Suppose $\alpha \in \Sigma_1$. Then from Definition 19 and 6.20 we know that $M \downharpoonleft \Sigma_1$ contains the following transition:

$$s \xrightarrow{\alpha} s' \tag{6.23}$$

The following commutative diagram explains the basic idea behind the proof.

$$\begin{tikzcd}
s \arrow[r, "\alpha"] \arrow[d, "\mathcal{R}"'] & s' \arrow[d, "\mathcal{R}"] \\
(s_1, s_2) \arrow[r, "\alpha"] \arrow[d, "\mathcal{R}_1"'] & (s_1', s_2') \arrow[d, "\mathcal{R}_1"] \\
s_1 \arrow[r, "\alpha"] & s_1'
\end{tikzcd}$$

Since $\mathcal{R}_1$ is a weak simulation relation and since $(s, s_1) \in \mathcal{R}_1$ (cf. 6.19) we have from 6.23:

$$\exists s_1' \in S_1 \centerdot s_1 \xrightarrow{\alpha} s_1' \bigwedge (s', s_1') \in \mathcal{R}_1 \tag{6.24}$$

Clearly the s_1' above meets the requirement of 6.22.

Now suppose that $\alpha \notin \Sigma_1$. Then from Definition 19 and 6.20 we know that $M \downharpoonleft \Sigma_1$ contains the following transition:

$$s \xrightarrow{\tau} s' \tag{6.25}$$

The following commutative diagram explains the basic idea behind the proof.

$$\begin{tikzcd}
s \arrow[r, "\alpha"] \arrow[d, "\mathcal{R}"'] & s' \arrow[d, "\mathcal{R}"] \\
(s_1, s_2) \arrow[r, "\alpha"] \arrow[d, "\mathcal{R}_1"'] & (s_1, s_2') \arrow[d, "\mathcal{R}_1"] \\
s_1 \arrow[r] & s_1
\end{tikzcd}$$

Since $\mathcal{R}_1$ is a weak simulation relation and since $(s, s_1) \in \mathcal{R}_1$ (cf. 6.19) and since $\tau \notin \Sigma_1$, we have from 6.25:

$$(s', s_1) \in \mathcal{R}_1 \tag{6.26}$$

Then clearly s_1 itself can be used as the s_1' that meets the requirement of 6.22. Thus we have shown the existence of an s_1' which satisfies 6.22 irrespective of whether

$\alpha \in \Sigma_1$ or not. In a completely symmetric manner we can show the existence of an s'_2 which satisfies 6.22 irrespective of whether $\alpha \in \Sigma_2$ or not. Thus we have shown that $\mathcal{R}$ is a simulation relation. This completes the proof of the reverse implication and hence of the theorem.

$\square$

Theorem 12 essentially allows us to discharge a simulation obligation by performing *weak* simulation checks between Counterexample Witness projections and components. It is easy to see that due to the associativity and commutativity of parallel composition, Theorem 12 can be extended to any finite number of LKSs. In other words, Theorem 12 still holds if we replace $\langle M_1, M_2 \rangle$ with any finite sequence of LKSs $\langle M_1, \ldots, M_n \rangle$.

Algorithm **WeakSimul**, presented in Procedure 6.1, takes as input a Counterexample Witness projection CW, a component $\mathcal{C}$ and a context γ. Recall that $[\![\mathcal{C}]\!]_\gamma$ denotes the concrete semantics of $\mathcal{C}$ with respect to γ. Algorithm **WeakSimul** returns TRUE if $CW \precsim [\![\mathcal{C}]\!]_\gamma$ and FALSE otherwise.

Given two LKSs $M_1 = (S_1, Init_1, AP_1, L_1, \Sigma_1, T_1)$ and $M_2 = (S_2, Init_2, AP_2, L_2, \Sigma_2, T_2)$, we say that a state $s_2 \in S_2$ (weakly) simulates a state $s_1 \in S_1$ iff there exists a (weak) simulation relation $\mathcal{R} \subseteq S_1 \times S_2$ such that $s_1 \mathcal{R} s_2$. Intuitively, **WeakSimul** invokes algorithm **CanSimul** (presented in Procedure 6.2) to compute the set of states of $[\![\mathcal{C}]\!]_\gamma$ that can weakly simulate the initial state of CW. Then $CW \precsim [\![\mathcal{C}]\!]_\gamma$ iff some initial state of $[\![\mathcal{C}]\!]_\gamma$ can weakly simulate the initial state of CW.

The recursive algorithm **CanSimul** takes as input a projected Counterexample Witness CW, a state s of CW, a component $\mathcal{C}$, and a context γ for $\mathcal{C}$. Recall that

Procedure 6.1 WeakSimul returns TRUE iff $CW \precsim [\![\mathcal{C}]\!]_\gamma$.

```
Algorithm WeakSimul($CW, \mathcal{C}, \gamma$)
- $CW$ : is a Counterexample Witness
- $\mathcal{C}$ : is a component, $\gamma$ : is a context for $\mathcal{C}$
let $CW = (S_1, Init_1, AP_1, L_1, \Sigma_1, T_1)$;
let $[\![\mathcal{C}]\!]_\gamma = (S_2, Init_2, AP_2, L_2, \Sigma_2, T_2)$;
  //$[\![\mathcal{C}]\!]_\gamma$ is the concrete semantics of $\mathcal{C}$ with respect to $\gamma$
$S$ := CanSimul($CW, Init_1, \mathcal{C}, \gamma$);
  //$S$ = states of $[\![\mathcal{C}]\!]_\gamma$ which can weakly simulate initial state of $CW$
return $(S \cap Init_2) \neq \emptyset$;
```

CW has a tree structure and hence no marking of the states of CW is required to avoid revisiting them. **CanSimul** computes the set of states of $[\![\mathcal{C}]\!]_\gamma$ which can weakly simulate the *sub-LKS* of CW with initial state s. It manipulates sets of states of $[\![\mathcal{C}]\!]_\gamma$ using the symbolic techniques presented in Section 3.4. In particular it uses the functions $PreImage$ and $Restrict$ to compute pre-images and restrict sets of states with respect to propositions.

Theorem 13 *Algorithms* **CanSimul** *and* **WeakSimul** *are correct.*

Proof. From the definition of weak simulation, the correctness of $Restrict$ and $PreImage$, and the fact that $\tau \notin \Sigma_{\mathcal{C}\gamma}$.

□

Note that the ability to validate a Counterexample Witness using its projections enables us to avoid exploring the state-space of $\mathcal{P}$. Not only does this compositional approach for Counterexample Witness validation help us avoid state-space explosion, it also identifies the particular component whose abstraction has to be refined in order to eliminate a spurious Counterexample Witness.

Procedure 6.2 CanSimul computes the set of states of $[\![\mathcal{C}]\!]_\gamma$ which can weakly simulate the sub-LKS of CW with initial state s.

```
**Algorithm CanSimul**$(CW, s, \mathcal{C}, \gamma)$
- $CW$ : is a Counterexample Witness, $s$ : is a state of $CW$
- $\mathcal{C}$ : is a component, $\gamma$ : is a context for $\mathcal{C}$
let $CW = (S_1, Init_1, AP_1, L_1, \Sigma_1, T_1)$;
let $[\![\mathcal{C}]\!]_\gamma = (S_2, Init_2, AP_2, L_2, \Sigma_2, T_2)$;
  //$[\![\mathcal{C}]\!]_\gamma$ is the concrete semantics of $\mathcal{C}$ with respect to $\gamma$
$S := Restrict(S_2, L_1(s))$;
  //$S$ is the subset of $S_2$ with same propositional labeling as $s$
for each $s \xrightarrow{\alpha} s' \in T_1$    //$s'$ is a successor state of $s$
    $S' :=$ CanSimul$(CW, s', \mathcal{C}, \gamma)$;    //compute result for successor
    if $(\alpha \neq \tau)$ then $S' := PreImage(S', \alpha)$;    //take non-$\tau$ pre-image
    $S := S \cap S'$;    //update result
return $S$;
```

In particular, suppose that CW is a spurious Counterexample Witness. Recall that our program consists of n components $\{\mathcal{C}_1, \ldots, \mathcal{C}_n\}$. Then according to Theorem 12 there exists a minimum $i \in \{1, \ldots, n\}$ such that the projection of CW on γ_i is not weakly simulated by the concrete semantics of $\mathcal{C}_i$. Therefore, we can eliminate CW by refining our abstraction for $\mathcal{C}_i$ to obtain a new abstraction Abs such that the projection of CW on γ_i is not weakly simulated by Abs. This refinement process is presented in the next section.

Example 15 *Figure 6.1 shows the component $\mathcal{C}$ and the* CounterexampleWitness *CW from our running example. Recall that the actions α, β, χ and δ are performed by the library routines alpha, beta, chi and delta respectively. Since there is only one component, the projection of CW is CW itself. Note that CW is not weakly simulated by $[\![\mathcal{C}]\!]_\gamma$. Intuitively this is because CW can perform actions α and δ along the two branches of its computation tree but $[\![\mathcal{C}]\!]_\gamma$ cannot. This is because for $[\![\mathcal{C}]\!]_\gamma$ to perform α and δ, variable x has to be* TRUE *while variable y has to be* FALSE*. However this is*

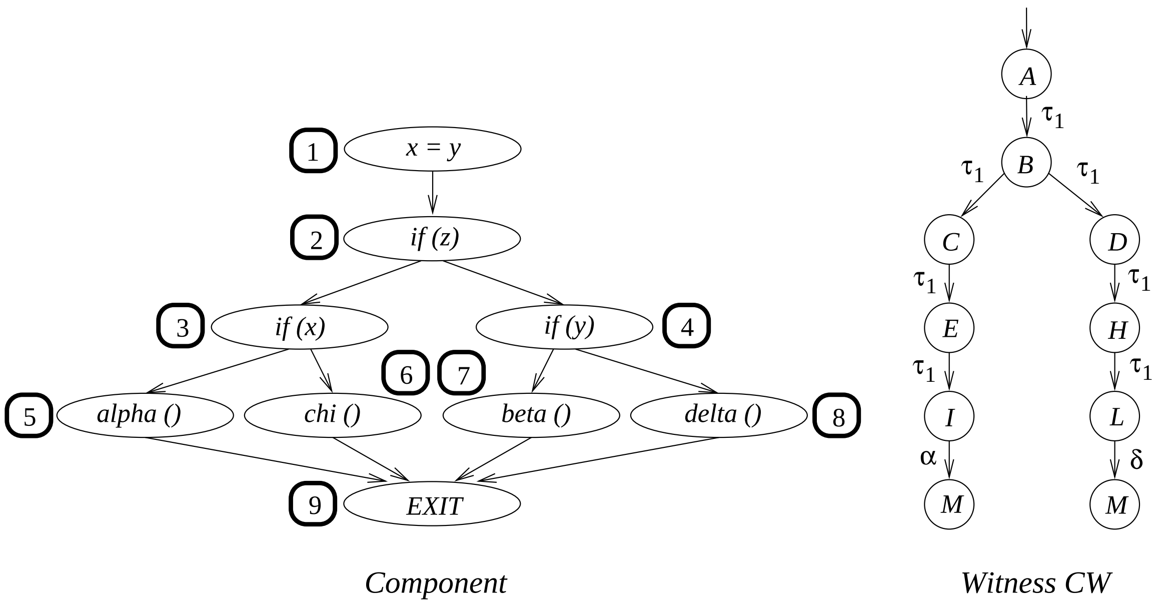


Figure 6.1: The component from Figure 4.1 and CounterexampleWitness from Figure 5.4. Note that the CounterexampleWitness is spurious.

clearly impossible due to the initial assignment of y to x. Therefore CW is a spurious counterexample.

6.2 Abstraction Refinement

Let $\mathcal{C}$ be a component, γ be a context for $\mathcal{C}$ and CW be a set of projections of Counterexample Witnesses on γ. Recall, from Section 4.5, that in our framework a predicate abstraction is determined by a predicate mapping. The predicate mapping, in turn, is obtained from a set of seed branches (cf. Section 4.5) and a context using predicate inference (cf. Section 4.5). The abstraction refinement process is encapsulated by algorithm **AbsRefine**, presented in Procedure 6.3. Essentially, it works as follows.

Recall the algorithm **PredInfer** from Procedure 4.1 which starts with a set of

Procedure 6.3 AbsRefine returns a refined abstraction for $\mathcal{C}$ that eliminates a set of spurious Counterexample Witness projections CW and ERROR on failure.

```
Algorithm AbsRefine($CW, \mathcal{C}, \gamma$)
- $CW$ : is a set of spurious Counterexample Witnesses
- $\mathcal{C}$ : is a component, $\gamma$ : is a context for $\mathcal{C}$
let $CW = \{CW_1, \ldots, CW_k\}$;
for each $B \subseteq \mathcal{B}_{\mathcal{C}}$    //$\mathcal{B}_{\mathcal{C}}$ is the set of branches in $\mathcal{C}$
    $\Pi :=$ PredInfer($\mathcal{C}, \gamma, B$);
      //$\Pi$ is set of predicates inferred from $B$
    let $\widehat{M} := [\![\mathcal{C}]\!]^{\Pi}_{\gamma} = \left(\widehat{S}, \widehat{Init}, \widehat{AP}, \widehat{L}, \widehat{\Sigma}, \widehat{T}\right)$;
      //$\widehat{M}$ is the predicate abstraction of $\mathcal{C}$ using $\Pi$
    $flag :=$ TRUE;
      //$flag$ records if $B$ can eliminate every element of $CW$
    for $i = 1$ to $k$
        let $CW_i = (S_i, Init_i, AP_i, L_i, \Sigma_i, T_i)$;
        $S :=$ AbsCanSimul($CW_i, Init_i, \widehat{M}$);
          //$S$ = states of $\widehat{M}$ which can weakly simulate initial state of $CW_i$
        if $((S \cap \widehat{Init}) \neq \emptyset)$ then $flag :=$ FALSE;
          //$CW_i \precsim \widehat{M}$ and hence $B$ cannot eliminate $CW_i$
    if $flag$ then return $\widehat{M}$;
return ERROR;
```

seed branches and populates each statement of a component with a set of predicates which can be used subsequently for predicate abstraction. We consider subsets of branches of $\mathcal{C}$ in increasing order of size. For each set B of branches we compute the predicate mapping $\Pi =$ **PredInfer**$(\mathcal{C}, \gamma, B)$ for $\mathcal{C}$. Next we compute the abstraction $\widehat{M} = [\![\mathcal{C}]\!]^{\Pi}_{\gamma}$. Let $CW_i = (S_i, Init_i, AP_i, L_i, \Sigma_i, T_i)$ for each $CW_i \in CW$. Now for each $CW_i \in CW$, we invoke algorithm **AbsCanSimul** (presented in Procedure 6.4) to compute the set of states of $\widehat{M}$ which can weakly simulate $Init_i$. If, for each $CW_i \in CW$, no initial state of $\widehat{M}$ can weakly simulate $Init_i$, then for each $CW_i \in CW$, $CW_i \not\precsim \widehat{M}$. In this case we report $\widehat{M}$ as the refined abstraction for $\mathcal{C}$ and stop. Otherwise, we move on with the next set of branches under consideration.

Procedure 6.4 AbsCanSimul computes the set of states of $\widehat{M}$ which can weakly simulate the sub-LKS of CW with initial state s.

```
Algorithm AbsCanSimul($CW, s, \widehat{M}$)
- $CW$ : is a Counterexample Witness, $s$ : is a state of $CW$
- $\widehat{M}$ is an LKS obtained by predicate abstraction
let $CW = (S, Init, AP, L, \Sigma, T)$;
let $\widehat{M} = \left(\widehat{S}, \widehat{Init}, \widehat{AP}, \widehat{L}, \widehat{\Sigma}, \widehat{T}\right)$;
$\widehat{S'} := \{\widehat{s} \in \widehat{S} \mid \widehat{L}(\widehat{s}) = L(s)\}$;
  //$\widehat{S'}$ is the subset of $\widehat{S}$ with same propositional labeling as s
for each $s \xrightarrow{\alpha} s' \in T$    //$s'$ is a successor state of $s$
    $\widehat{S''}$ := AbsCanSimul($CW, s', \widehat{M}$);    //compute result for successor
    if ($\alpha \neq \tau$) then $\widehat{S''} := \{\widehat{s} \in \widehat{S} \mid Succ(\widehat{s}, \alpha) \cap \widehat{S''} \neq \emptyset\}$; //take non-$\tau$ pre-image
    $\widehat{S'} := \widehat{S'} \cap \widehat{S''}$;    //update result
return $\widehat{S'}$;
```

Theorem 14 *Algorithm* **AbsRefine** *is correct.*

Proof. It is obvious that **AbsRefine** either returns ERROR or a refined abstraction $\widehat{M}$ such that $\forall i \in \{1, \ldots, k\} \centerdot CW_i \not\precsim \widehat{M}$.

□

Recall the component $\mathcal{C}$ and the spurious CounterexampleWitness CW from Figure 6.1. Note that both branch statements 3 (with branch condition x) and 4 (with branch condition y) are required as seeds in order to obtain a predicate abstraction that is precise enough to eliminate CW. Neither branch 3 nor branch 4 is adequate by itself. This is because the spuriousness of CW relies on the direct correlation between the truth and falsehood of variables x and y. Any abstraction must capture this correlation in order to eliminate CW. In our framework, this can only be achieved by using both 3 and 4 as seed branches for the predicate inference and subsequent predicate abstraction.

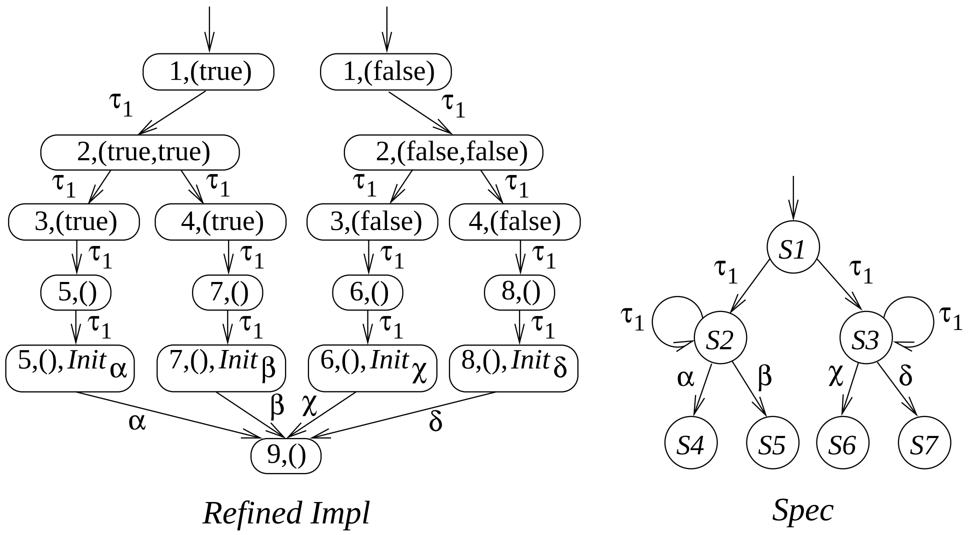


Figure 6.2: On the left is the refined abstraction of the component from Figure 6.1 using states $\{3, 4\}$ as seeds. The empty valuation $\perp$ is written as "()". On the right is the specification from Figure 5.3. Note that the refined abstraction is simulated by the specification.

Example 16 *Recall from Figure 4.3 the predicate mapping obtained by using states* $\{3, 4\}$ *as seeds. Figure 6.2 shows the refined abstraction using the resulting predicate mapping. The empty valuation* $\perp$ *is written as "()". It also shows the specification of our running example from Figure 5.3. Note that the refined abstraction is simulated by the specification.*

It is well known that simulation is preferred over trace containment because it does not require the complementation (and hence potential exponential blowup in size) of the specification. Our running example illustrates an additional advantage of simulation conformance over trace containment in the context of CEGAR-based verification. In particular, the additional *structure* (and hence information) conveyed by tree counterexamples obtained in the context of simulation conformance can aid in quicker predicate discovery and termination.

Suppose we had attempted to check trace containment on our example. We know that at least the two branches 3 and 4 are required for successful verification. Also note that these two branches cannot appear simultaneously in the same trace counterexample since they appear in disjoint fragments of the control flow of the component. Hence we would have required at least two refinement steps in order to successfully verify trace containment. In contrast, as we have already seen, verification of simulation conformance requires just a single refinement step. We will provide experimental evidence supporting this intuitive argument in Section 6.5.

Another approach to speed-up the termination of the CEGAR loop is to generate multiple counterexamples at the end of each unsuccessful verification step. The idea is that more counterexamples convey more information and will lead to quicker realization of an abstraction that is precise enough to either validate the existence of conformance or yield a non-spurious counterexample. However, manipulating a large number of counterexamples is expensive and will only provide diminishing returns beyond a certain threshold. We will also provide experimental justification of this argument in Section 6.5.

Note that our algorithm for constructing predicate mappings is restricted in the sense that it can only derive predicates from branch conditions. Therefore, in principle, we might be unable to eliminate a spurious Counterexample Witness. In the context of algorithm **AbsRefine**, this means that we could end up trying all sets of branches without finding an appropriate refined abstraction $\widehat{M}$. In such a case we return ERROR. We note that this scenario has never transpired during our experiments. Moreover, any abstraction refinement technique must necessarily suffer from this limitation since the problem we are attempting to solve is undecidable in general.

Also **AbsRefine** attempts to eliminate a set of spurious Counterexample Witness projections instead of a single projection. This will be necessary in the context of the complete CEGAR algorithm presented in the next section. In fact, **AbsRefine** iterates through the subsets of $\mathcal{B}_{\mathcal{C}}$ (the set of all branches in $\mathcal{C}$) in increasing order of size. Therefore the refined abstraction returned by **AbsRefine** corresponds to a *minimal* set of branches that can eliminate the entire set of spurious Counterexample Witness projections passed to it. This is an important feature because the size of an abstraction is, in the worst case, exponential in the number of branches used in its construction.

However, **AbsRefine** is also naive in the following sense. As we shall see shortly, the set of Counterexample Witness projections passed to **AbsRefine** by the top-level CEGAR algorithm will increase monotonically across successive invocations. Nevertheless, **AbsRefine** naively recomputes **AbsCanSimul**$(CW_i, Init_{CW_i}, \widehat{M})$ even though it might already have encountered CW_i in a previous invocation. In Chapter 7 we shall present a more sophisticated approach that avoids this redundant computation without compromising on the minimality of the number of branches used in the computation of the refined abstraction.

6.3 CEGAR for Simulation

The complete CEGAR algorithm in the context of simulation conformance, called **SimulCEGAR**, is presented in Procedure 6.5. It invokes at various stages algorithms **PredInfer**, the predicate abstraction algorithm, **SimulWitness**, **WeakSimul** and **AbsRefine**. It takes as input a program $\mathcal{P}$, a specification LKS Sp and a context Γ for $\mathcal{P}$ and outputs either "$\mathcal{P} \preccurlyeq Sp$" or "$\mathcal{P} \not\preccurlyeq Sp$" or ERROR. Intuitively **SimulCEGAR** works as follows.

Procedure 6.5 SimulCEGAR checks simulation conformance between a program $\mathcal{P}$ and a specification Sp in a context Γ.

```
Algorithm SimulCEGAR($\mathcal{P}, Sp, \Gamma$)
- $\mathcal{P}$ : is a program, $\Gamma$ : is a context for $\mathcal{P}$
- $Sp$ : is a specification LKS
let $\mathcal{P} = \langle \mathcal{C}_1, \dots, \mathcal{C}_n \rangle$ and $\Gamma = \langle \gamma_1, \dots, \gamma_n \rangle$;
for each $i \in \{1, \dots, n\}$, $\Pi_i :=$ PredInfer($\mathcal{C}_i, \gamma_i, \emptyset$) and $\widehat{M_i} := [\![\mathcal{C}_i]\!]_{\gamma_i}^{\Pi_i}$ and $CW_i := \emptyset$;
    //initialize abstractions with empty set of seed branches
forever do
    let $\widehat{M} = \widehat{M_1} \parallel \cdots \parallel \widehat{M_n}$;
      //$\widehat{M}$ is the composition of predicate abstractions
    if (SimulWitness($\widehat{M}, Sp$) = "$\widehat{M} \preccurlyeq Sp$") return "$\mathcal{P} \preccurlyeq Sp$";
      //if the property holds on $\widehat{M}$ it also holds on $\mathcal{P}$
    let $CW$ = Counterexample Witness returned by SimulWitness;
    find $i \in \{1, \dots, n\}$ such that $\neg$WeakSimul($CW \downharpoonright \gamma_i, \mathcal{C}_i, \gamma_i$);
      //check compositionally if $CW$ is spurious
    if (no such $i$ found) return "$\mathcal{P} \not\preccurlyeq Sp$";
      //$CW$ is valid and hence $\mathcal{P}$ is not simulated by $Sp$
    else $CW_i := CW_i \cup \{CW \downharpoonright \gamma_i\}$;
      //update the set of spurious Counterexample Witnesses
    if (AbsRefine($CW_i, \mathcal{C}_i, \gamma_i$) = ERROR) return ERROR;
    $\widehat{M_i}$ := AbsRefine($CW_i, \mathcal{C}_i, \gamma_i$);    //refine the abstraction and repeat
```

Let $\mathcal{P} = \langle \mathcal{C}_1, \dots, \mathcal{C}_n \rangle$. Then **SimulCEGAR** maintains a set of abstractions $\widehat{M_1}, \dots, \widehat{M_n}$ where $\widehat{M_i}$ is a predicate abstraction of $\mathcal{C}_i$ for $i \in \{1, \dots, n\}$. It also maintains a set of spurious Counterexample Witness projections $\{CW_1, \dots, CW_n\}$ which are all initialized to the empty set. Note that by Theorem 7, $\widehat{M} = \widehat{M_1} \parallel \cdots \parallel \widehat{M_n}$ is an abstraction of $\mathcal{P}$. Initially each $\widehat{M_i}$ is set to the predicate abstraction of $\mathcal{C}_i$ corresponding to an empty set of seed branches. Next **SimulCEGAR** iteratively performs the following steps:

1. **(Verify)** Invoke algorithm **SimulWitness** to check if $\widehat{M}$ is simulated by Sp. If **SimulWitness** returns "$\widehat{M} \preccurlyeq Sp$" then output "$\mathcal{P} \preccurlyeq Sp$" and exit. Otherwise

let CW be the Counterexample Witness returned by **SimulWitness**. Go to step 2.

2. **(Validate)** For $i \in \{1, \ldots, n\}$ invoke **WeakSimul**$(CW \downharpoonleft \gamma_i, \mathcal{C}_i, \gamma_i)$. If every invocation of **WeakSimul** returns TRUE then CW is a valid Counterexample Witness. In this case, output "$\mathcal{P} \not\preccurlyeq Sp$" and exit. Otherwise let i be the minimal element of $\{1, \ldots, n\}$ such that **WeakSimul**$(CW \downharpoonleft \gamma_i, \mathcal{C}_i, \gamma_i)$ returns FALSE. Go to step 3.

3. **(Refine)** Update CW_i by adding $CW \downharpoonleft \gamma_i$ to it. Invoke **AbsRefine**$(CW_i, \mathcal{C}_i, \gamma_i)$. If **AbsRefine** returns ERROR, output ERROR and stop. Otherwise set $\widehat{M_i}$ to the abstraction returned by **AbsRefine**. Repeat from step 1.

Theorem 15 *Algorithm* **SimulCEGAR** *is correct.*

Proof. When **SimulCEGAR** returns "$\mathcal{P} \preccurlyeq Sp$" its correctness follows from Theorem 7, Theorem 11 and Theorem 1. When **SimulCEGAR** returns "$\mathcal{P} \not\preccurlyeq Sp$" its correctness follows from Theorem 12, Theorem 13 and Theorem 14.

□

6.4 The MAGIC Tool

We have implemented the game semantics based refinement approach within the MAGIC [25, 80] tool. In this section we give a brief overview of how MAGIC can be used. This is essentially a copy of the tutorial of version 1.0 of MAGIC available online at `http://www.cs.cmu.edu/~chaki/magic/tutorial-1.0.html`, and should be a good starting point.

6.4.1 A Simple Implementation

The goal of MAGIC is to ascertain that an implementation conforms to a specification. The implementation is always a (possibly concurrent) C program, with each sequential component being a C procedure. Let us begin with a simple sequential implementation.

```
int my_proc(int x)
{
    int y;
    if(x == 0) {
       y = foo();
       if(y > 0) return 10;
       else return 20;
    } else {
       y = bar();
       if(y < 0) return 30;
       else return 40;
    }
}
```

6.4.2 A Simple Specification

We shall now try to construct an appropriate specification for `my_proc` and then verify it. Note that we are intentionally proceeding in the reverse direction for ease of understanding. Normally, if standard software engineering practices have been followed, the specification always comes into existence before the implementation.

So what could be a good specification for `my_proc`? In a kind of pseudo-code, one might make the following claim about `my_proc`:

1. If the first argument to `my_proc` is equal to zero:

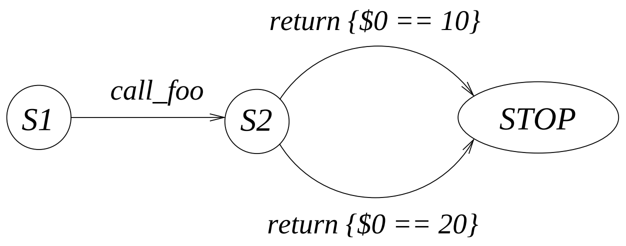
    - `my_proc` calls the library routine `foo`.

- Depending on whether the value returned by `foo` is greater than zero or not, `my_proc` returns either 10 or 20.

2. Otherwise, if the first argument to `my_proc` is not equal to zero:

   - `my_proc` calls the library routine `bar`.
   - Depending on whether the value returned by `bar` is less than zero or not, `my_proc` returns either 30 or 40.

It is clear that any specification of `my_proc` must take into account its calling context, since the behavior of `my_proc` is dependent on the value of its first argument. Further, the behavior of `my_proc` in the first case (i.e. when its first argument is equal to zero) can be expressed by the following simple LKS.



Writing down LKSs

MAGIC uses an extended FSP [79] notation to specify LKSs. The above LKS can be expressed in MAGIC's notation as follows:

```
S1 = ( call_foo -> S2 ),
S2 = ( return {$0 == 10} -> STOP | return {$0 == 20} -> STOP ).
```

Note that the name of an LKS is simply the name of its initial state. Also note that the transitions of the LKS are labeled with actions. The transition from the initial state (S1) is labeled by an action `call_foo`. This action encapsulates the externally observable event of library routine `foo` being invoked. Such actions, with only names, are also called basic actions.

Return actions

As anyone familiar with the FSP will realize, we extend the FSP notation to express a special class of actions called return actions. Return actions are of the form `return {expression}` or `return {}` where the former expresses the return of an integer value and the latter expresses returning a void value. In a return action of the form `return {expression}`, the `expression` represents a condition satisfied by the return value. The return value itself is represented by the dummy variable `$0`. For instance, the action `return {$0 < 5}`. represents the return of an integer value less than 5.

Procedure Block

We are now ready to express the fact that S1 specifies the behavior of `my_proc` when the first argument of `my_proc` is equal to zero. In MAGIC, this can be achieved by the following procedure block:

```
cproc my_proc {
        abstract { abs_1 , ($1 == 0) , S1 };
}
```

Note the keywords `cproc` and `abstract`. The block keyword `cproc` indicates that we are going to say something about a C procedure. It is followed by the name of the procedure and then by a set of statements enclosed within a pair of curly braces. Each such statement typically consists of an statement keyword followed by other terms. The procedure whose name follows `cproc` is often referred to as the scope procedure.

One such statement keyword is `abstract`.This keyword indicates that we are expressing an abstraction relation between the scope procedure and an LKS. Note the guard `($1 == 0)` where `$1` refers to the first argument. In general `$i` can be used to refer to the i-th argument of the scope procedure. Finally note that the abstraction statement has a name, `abs_1`. For procedure blocks, the abstraction names are just placeholders and have no special significance. However, soon we will discuss program blocks and for them, the names of abstraction statements will be of crucial importance.

The following LKS expresses the behavior of `my_proc` when its first argument is not equal to zero:

```
S3 = ( call_bar -> S4 ),
S4 = ( return {$0 == 30} -> STOP | return {$0 == 40} -> STOP ).
```

Thus we can have another procedure block to specify the relation between `my_proc` and S3.

```
cproc my_proc {
        abstract { abs_2 , ($1 != 0) , S3 };
}
```

In general, multiple procedure blocks can be combined into one as long as they have the same scope procedure. Also the order of statements within a procedure block is irrelevant. Thus, the above two procedure blocks together is equivalent to the following single procedure block:

```
cproc my_proc {
        abstract { abs_2 , ($1 != 0) , S3 };
        abstract { abs_1 , ($1 == 0) , S1 };
}
```

MAGIC requires that the guards of abstraction statements for any scope procedure be mutually disjoint and complete (i.e. cover all possibilities of argument valuations).

This is necessary to enable MAGIC to unambiguously identify the applicable abstraction in any given calling context of the scope procedure.

Specifying Library Routines

In order to construct a proper model for `my_proc` MAGIC must know about the behavior of the library routines called by `my_proc`. Let us assume that the actual code for `foo` and `bar` are unavailable. In such a case, MAGIC requires that the user supply appropriate abstractions for these two routines. In particular, suppose that `foo` and `bar` are respectively abstracted by the LKSs `FOO` and `BAR` described below:

```
FOO = ( call_foo -> return {$0 == -1} -> STOP ).
BAR = ( call_bar -> return {$0 == 50} -> STOP ).
```

Then the following program blocks can be used to express the relation between `foo`, `bar` and their abstractions.

```
cproc foo {
       abstract { abs_3 , (1) , FOO };
}

cproc bar {
       abstract { abs_4, (1), BAR };
}
```

Note that the guard in both abstraction statements is 1, which denotes TRUE according to C semantics. This therefore means that under all calling contexts, `foo` and `bar` are abstracted by `FOO` and `BAR` respectively. Also note that specifications and abstractions are syntactically identical. This makes sense because both abstractions and specifications are essentially asserting the same thing viz. that under a certain calling context, a procedure's behavior is subsumed by the behavior of an LKS. The only difference is that the assertion made by an abstraction can be assumed to be true while the assertion made by a specification needs to be validated. This has at least two significant consequences:

- **Verifying Incomplete Code:** In practice, one cannot assume that the actual code for each and every library routine used by a program will be available to the verifier. Hence being able to provide abstractions allows MAGIC to analyze such incomplete implementations. In effect, abstractions allow us to specify assumptions about the environment in which a program operates.

- **Compositionality:** Often programs are simply too big to be analyzed as a monolithic piece of code. Abstractions allow us to decompose such large implementations into smaller, more manageable fragments. Fragments can be verified one at a time. While verifying one fragment, the abstractions of other fragments can be used as assumptions. The fact that specifications and abstractions are identical implies that they can naturally switch from one role to the other depending on which fragment is being verified.

Program Block

It is now time to specify the entire program that we want to verify. In our case the program is sequential, i.e. it has a single component consisting of the procedure `my_proc`. The following program block expresses the relation between our program and its specification:

```
cprog my_prog = my_proc {
        abstract abs_5, {($1 == 0)}, S1;
        abstract abs_6, {($1 != 0)}, S3;
}
```

This looks a lot like a procedure block but there are some crucial differences. First, it begins with the keyword `cprog` and not `cproc`. This is followed by the name of the program (which is again a placeholder and does not serve any other purpose), an equal to sign and then a list of procedure names. Intuitively these are the names of the procedures which execute in parallel and constitute the program. In the above block this list has a single procedure name viz. `my_proc`, signifying that our program has just one component that executes `my_proc`.

Following the list of procedure names we have a sequence of statements enclosed within curly braces. Just like procedure blocks, abstraction statements are used to provide specifications. But abstraction statements for program blocks are syntactically different. They do begin with the `abstract` keyword, but the rest of it them is not enclosed within curly braces. Instead there are three components. The first is the name of the abstraction statement. This is used by MAGIC to identify the target abstraction to be validated. The second is a list of guards, one for each component of the program. Each guard in the list expresses the beginning state of the corresponding component. In the above block, the list has just one element that expresses the starting context of `my_proc`. Note that the list of guards is enclosed within curly braces. The third and final component is the name of the LKS which specifies the program.

Comments

You can use either C-style or C++ style comments in specification files.

```
/* this is a comment */
// so is this one
```

6.4.3 Running MAGIC

We are now ready to try out MAGIC. First save the C program in a file whose name must end with ".pp", say `my_proc.pp`. Next save the specifications in another file whose name ends with ".spec", for example `my_spec-1.0.spec`. Finally run MAGIC:

```
$ magic --abstraction abs_5 my_proc.pp my_spec-1.0.spec --optPred
```

MAGIC will try to validate the abstraction statement with name `abs_5`. The `--optPred` options tells MAGIC to perform counterexample guided abstraction refinement with predicate minimization. It is usually a good idea to always use this option. For details on other options that MAGIC can accept, look at the user's manual. If all goes well, MAGIC should be able to successfully verify the abstraction and produce an output that ends with something like this:

```
conformance relation exists !!
abstraction abs_5 is valid ...
Simplify process destroyed ...
terminating normally ...
```

Similarly you can try to verify `abs_6` and MAGIC should be able to do it. If you look again at `my_spec-1.0.spec` you will notice that we have added two more abstraction statements, `abs_7` and `abs_8`, to the `my_prog` block. They are similar to `abs_5` and `abs_6` except that the guard conditions have been switched. Clearly they are invalid specifications and MAGIC should be able to detect this. Try verifying `abs_7` by typing the following:

```
$ magic --abstraction abs_7 my_proc.pp my_spec-1.0.spec
  --optPred --ceShowAct
```

MAGIC should tell you that this is an invalid specification and further provide you with a counterexample. The output should look something like the following:

```
*******************************************
branch ( P0::x == 0 ) : [P0::x == 0] : TRUE
############ P0::epsilon ############
P0::y = foo (  ) : []
############ P0::epsilon ############
P0::y = foo (  ) : []
############ call_foo ############
P0::y = foo (  ) : []
############ {P0::y = [ -1 ]} ############
branch ( P0::y > 0 ) : [] : TRUE
############ P0::epsilon ############
return ( 10 ) : []
############ return { 30 } ############
*******************************************
CE dag projections analysed ...
conformance relation does not exist !!
abstraction abs_7 is invalid ...
Simplify process destroyed ...
terminating normally ...
```

6.4.4 A Concurrent Example

Let us now verify a concurrent program. Our concurrent program will be very simple. It will be two copies of `my_proc` executing in parallel. This is easy to understand because the resulting parallel program should behave exactly like a single copy of `my_proc` (since our notion of parallel composition is idempotent). All we need to do is create a new program block specifying our example. Here is a sample `my_conc-1.0.spec`. Notice that it has four abstraction statements `abs_9`, `abs_10`, `abs_11` and `abs_12` out of which the first two are valid while the last two are invalid. We can try to verify `abs_9` by the following command:

```
$ magic --abstraction abs_9 my_proc.pp my_conc-1.0.spec --optPred
```

This should succeed. Likewise MAGIC should be able to prove that `abs_10` is also valid while `abs_11` and `abs_12` are both invalid.

6.4.5 Other Keywords

In addition to `abstract`, there are several other keywords that can be used in procedure blocks for performing specific tasks. In this section we mention a few important ones.

Supplying predicates

The user can manually supply predicates to guide MAGIC's predicate abstraction. Often this is useful when MAGIC fails to discover a satisfactory set of predicates in a reasonable amount of time. Predicates are supplied in a per-procedure basis. An important feature of MAGIC is that all user-supplied predicates for a procedure proc must be syntactically equivalent to some branch condition in proc.Otherwise that predicate will be simply ignored by MAGIC. For example consider the following C procedure:

```
int proc()
{
    int x = 5;
    if(x < 10) return -1;
    else return 0;
}
```

Suppose we want to prove using MAGIC that proc is correctly specified by the following LKS:

```
PROC = ( return {$0 == -1} -> STOP ).
```

Normally we would do this by simply asking MAGIC to perform automated abstraction refinement (using the `--optPred` option). However suppose we have a good idea about which predicate MAGIC will need to complete successfully. For example, in this case `(x < 10)` is the required predicate (note that this corresponds to a branch condition in `proc`). Then we can simply tell MAGIC to use this predicate by using the `predicate` keyword. The following procedure block shows how to do this:

```
cproc proc {
       predicate (x < 10);
}
```

MAGIC will look for branch statements in proc which have a branch condition `(x < 10)`. If it finds any such branch, it will use the corresponding branch condition as a seed predicate. Otherwise it will ignore the user supplied predicate. Multiple predicates can be supplied in one statement using a comma-separated list or they can be supplied via multiple `predicate` statements. Also the order in which predicates are supplied is irrelevant. For example the two following procedure blocks each have the same effect as the procedure block above:

```
cproc proc {
       predicate (y == 10) , (w == 5) , (z +w > 20) ,
                 (x < 10) , (x+y != 5);
}

cproc proc {
       predicate (x+y != 5);
       predicate (z+w > 20) , (y == 10);
       predicate (x < 10) , (w == 5);
}
```

Inlining Procedures

Suppose procedure `foo` calls procedure `bar`. Normally MAGIC will not inline `bar` within `foo` even if the code for `bar` is available. It has to be told explicitly to do this via the `inline` keyword. Here's a procedure block that demonstrates how to do this. Once again inlining has to be done on a procedure-to-procedure basis. For example the following procedure block will not cause `bar` to be inlined within some other procedure `baz`.

```
cproc foo {
       inline bar;
}
```

6.4.6 Drawing with MAGIC

MAGIC can be used to output control flow graphs, LKSs and intermediate data structures as postscript files. This is very useful for visualization and understanding. For example, using the following command line on `draw.pp` and `draw-1.0.spec` files produces a `draw-1.0.ps` file.

```
$ magic --abstraction abs_1 draw.pp draw-1.0.spec --optPred
  --drawPredAbsLTS --drawFile draw-1.0.ps
```

Also, using the following command line on `my_proc.pp` and `my_spec-1.0.spec` yields `my_proc-1.0.ps`.

```
$ magic --abstraction abs_5 my_proc.pp my_spec-1.0.spec --optPred
  --drawPredAbsLTS --drawFile my_proc-1.0.ps
```

Please look at the user's manual for more details on command line options that control MAGIC's drawing capabilities. Also note that in order to draw its figures MAGIC requires the GRAPHVIZ package, and in particular the DOT tool. However, if you do not want to use MAGIC's drawing capabilities, there is no need to install GRAPHVIZ. At this point, you should be more or less familiar with MAGIC and ready to play around with it. Have fun !!

6.5 Experimental Results

We validated the game semantics based refinement approach experimentally using MAGIC. As the source code for our experiments we used version 0.9.6c of OpenSSL [95], an open source implementation of the SSL [109] protocol used for secure exchange of information over the Internet. In the rest of this thesis we will refer to version 0.9.6c of OpenSSL as simply OpenSSL. In particular, we used the code that implements the server side of the initial *handshake* involved in SSL. The source code consisted of about 74000 lines including comments and blank lines. In Appendix A we present our OpenSSL benchmark in greater detail.

All our experiments were carried out on an AMD Athlon 1600 XP machine with 900 MB RAM running RedHat 7.1. Our experiments were carried out to achieve two broad objectives. First, we wanted to verify the advantages of simulation conformance over trace-containment conformance. Second, we wanted to evaluate the effect of using multiple spurious Counterexample Trees for abstraction refinement in every iteration of the CEGAR loop.

As we shall see shortly, our experimental results indicate that compared to trace containment, on average, simulation leads to 6.62 times faster convergence and requires 11.79 times fewer iterations. Furthermore, refining on multiple Counterexample Trees per iteration leads to up to 25% improvement in performance. However, using more than four Counterexample Trees is counterproductive.

We manually designed a set of eleven specifications by reading the SSL documentation. Each of these specifications was required to be obeyed by any correct SSL implementation. Each specification captured critical safety requirements. For example, among other things, the first specification enforced the fact that any handshake is always initiated by the client and followed by a correct authentication by the client. Each specification combined with the OpenSSL source code yielded one benchmark for experimentation.

First, each benchmark was run twice, once using simulation and again using trace

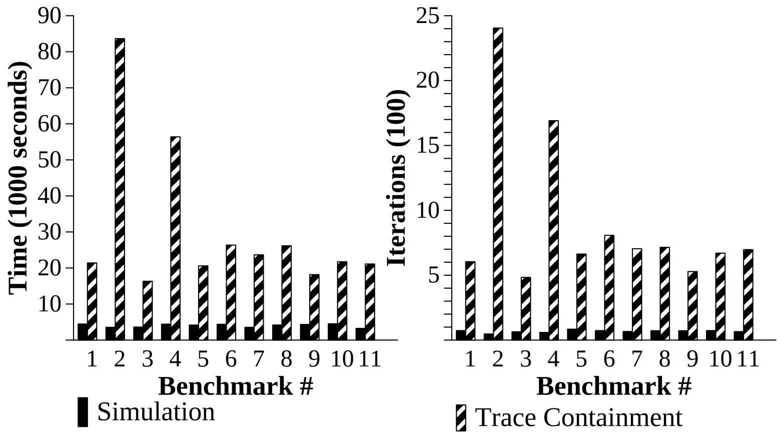


Figure 6.3: Comparison between simulation and trace containment in terms of time and number of iterations.

containment as a notion of conformance. For each run, we measured the time and number of iterations required. The resulting comparison is shown in Figures 6.3. These results indicate that simulation leads to faster (on average by 6.62 times) convergence and requires fewer (on average by 11.79 times) iterations.

Recall also that during the refinement step we use a Counterexample Tree to create a more refined predicate abstraction of some component. It is possible that in *each iteration* of the CEGAR loop, we generate not one but a set of Counterexample Trees $\widehat{CW}$. Using our benchmarks, we investigated the effect of increasing the size of $\widehat{CW}$. In particular, the measurements for total time were obtained as follows. The size of $\widehat{CW}$ was varied from 1 to 15 and for each value of $|\widehat{CW}|$, the time required to check simulation as well as trace containment for each benchmark was measured. Finally the *geometric mean* of these measurements was taken. The measurements for iterations and memory were obtained in a similar fashion.

The graphs in Figure 6.4 summarize the results we obtained. The figures indicate that it makes sense to refine on multiple counterexamples. We note that there is consistent improvement in all three metrics up to $|\widehat{CW}| = 4$. Increasing $|\widehat{CW}|$

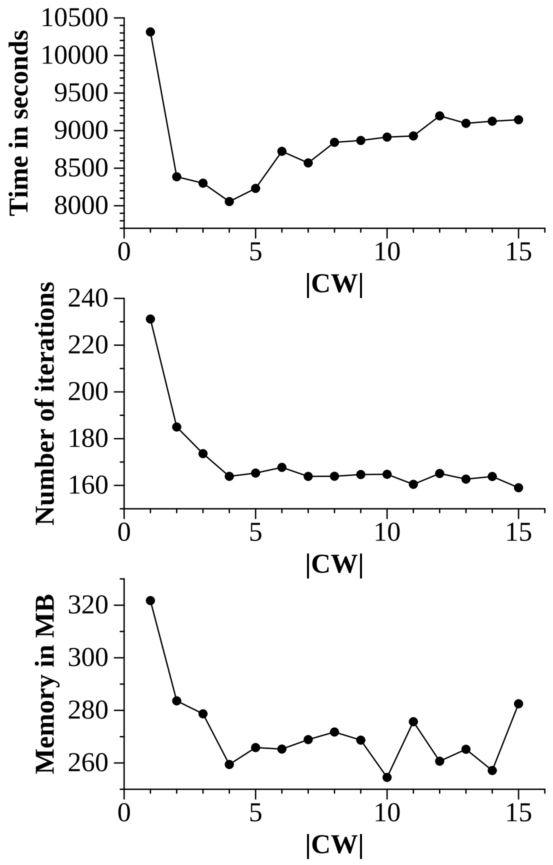


Figure 6.4: Time, iteration and memory requirements for different number of Counterexample Trees.

beyond four appears to be counterproductive. This supports our earlier intuition that manipulating a large number of counterexamples is expensive and will only provide diminishing returns beyond a certain threshold.

Chapter 7

Predicate Minimization

As we have already seen, predicate abstraction is in the worst case exponential (in both time and memory requirements) in the number of predicates involved. Therefore, a crucial requirement to make predicate abstraction effective is to use as few predicates as possible. Traditional approaches to counterexample guided predicate discovery analyze each new spurious counterexample in isolation and accumulate predicates monotonically. This may lead to larger than necessary sets of predicates, which may result in an inability to solve certain problem instances.

For example, consider a scenario where the first counterexample, CW_1, can be eliminated by either predicate $\{p_1\}$ or $\{p_2\}$, and the predicate discovery process chooses $\{p_1\}$. Now the CEGAR algorithm finds another counterexample CW_2, which can only be eliminated by the predicate $\{p_2\}$. The CEGAR algorithm now proceeds with the set of predicates $\{p_1, p_2\}$, although $\{p_2\}$ by itself is sufficient to eliminate both CW_1 and CW_2. This is clearly undesirable.

In Chapter 6 we presented a naive algorithm **AbsRefine** for finding a minimal sufficient predicate set from a given set of candidate predicates. In the above scenario, **AbsRefine** would indeed choose $\{p_2\}$ as the new set of predicates after encountering both CW_1 and CW_2. However it will perform redundant computation involving CW_1. In this chapter we will present a more sophisticated abstraction refinement algorithm called **AbsRefMin** which avoids the redundant computation without sacrificing the minimality of the result.

We have implemented **AbsRefMin** in the MAGIC tool. Our experimental results show that **AbsRefMin** can significantly reduce the number of predicates and consequently the amount of memory required in comparison to the naive **AbsRefine** algorithm as well as existing tools such as BLAST.

7.1 Related work

Predicate abstraction was introduced by Graf and Saidi in [63]. It was subsequently used with considerable success in both hardware and software verification [6, 50, 66]. The notion of CEGAR was originally introduced by Kurshan [76] (originally termed *localization*) for model checking of finite state models. Both the abstraction and refinement techniques for such systems, as employed in his and subsequent [37, 38] research efforts, are essentially different from the predicate abstraction approach we follow. For instance, abstraction in localization reduction is done by non-deterministically assigning values to selected sets of variables, while refinement corresponds to gradually returning to the original definition of these variables.

More recently the CEGAR framework has also been successfully adapted for verifying infinite state systems [102], and in particular software [7, 66]. The problem of finding small (but not minimal) sets of predicates has also been investigated in the context of hardware designs in [33]. The work most closely related to ours, however, is that of Clarke, Gupta, Kukula and Strichman [40] where the CEGAR approach is combined with integer linear programming techniques to obtain a minimal set of variables that separate sets of concrete states into different abstract states.

7.2 Pseudo-Boolean Constraints

A pseudo-Boolean (PB) formula is of the form $\sum_{i=1}^{n} c_i \cdot b_i \bowtie k$, where: (i) b_i is a Boolean variable, (ii) c_i is a rational constant for $1 \leq i \leq n$, (iii) k is a rational constant and (iv) $\bowtie$ represents one of the inequality or equality relations from the set $\{<, \leq, >, \geq, =\}$. Each such constraint can be expanded to a CNF formula (hence the name pseudo-Boolean). Hence a naive way to solve for the satisfiability of a PB formula is to translate it to a CNF formula and then use a standard CNF SAT solver [87, 104–106, 114].

Example 17 *A typical PB formula is $\phi_1 \equiv (2/3)x + 5y - (3/4)z < 6$. This formula is equivalent to the much simpler $\phi_2 \equiv 8x + 60y - 9z < 72$. Now we can convert ϕ_2 into a purely propositional form. A common way to do this is to assume each variable appearing in ϕ_2 to be a bit-vector of some fixed width w. We can then encode each variable using w Boolean propositions and use standard Boolean encodings for the arithmetic and relational operators appearing in ϕ_2 to complete the transformation.*

However the expanded CNF form of a PB formula φ can be exponential in the size of φ. The Pseudo-Boolean Solver (PBS) [2] does not perform this expansion, but rather uses an algorithm designed in the spirit of the Davis-Putnam-Loveland [51, 52] algorithm that handles these constraints directly. PBS accepts as input standard CNF formulas augmented with pseudo-Boolean constraints. Given a standard CNF formula ϕ and an objective function φ, PBS finds an optimal solution s_{opt} for ϕ. The objective function φ is usually an arithmetic expression over the variables of ϕ treated as having a value of either zero or one. The result s_{opt} is optimal in the sense that it minimizes the value of φ. PBS achieves this by repeatedly tightening the constraint over the value of φ until ϕ becomes unsatisfiable.

More precisely, PBS first finds a satisfying solution s to ϕ and calculates the value of φ according to s. Let this value be $\varphi(s)$. PBS then updates ϕ by adding a constraint that the value of φ should be less than $\varphi(s)$ and re-solves for the satisfiability of ϕ. This process is repeated until ϕ becomes unsatisfiable. The output s_{opt} is then the last satisfying solution for ϕ. Note that a possible improvement on this process is to perform a binary (rather than a linear) search over the value of φ. However, the performance of PBS was not a bottleneck in any of our experiments.

7.3 Predicate Minimization

We now describe the algorithm **AbsRefMin** (presented in Procedure 7.1) for computing a refined abstraction based on a minimal set of branches that can eliminate a set of spurious Counterexample Witness projections. Essentially **AbsRefMin** works as follows. Let $\mathcal{B}_{\mathcal{C}} = \{b_1, \ldots, b_k\}$ be the set of branches of the component $\mathcal{C}$ as usual. Suppose we introduce Boolean variables $BV = \{v_1, \ldots, v_k\}$, where each v_i corresponds to the branch b_i. The arguments to **AbsRefMin** are: (i) a spurious Counterexample Witness projection CW, (ii) a component $\mathcal{C}$, (iii) a Boolean formula ϕ over BV, and (iv) a context γ for $\mathcal{C}$.

Intuitively, ϕ captures the information about the sets of branches which can eliminate all the spurious Counterexample Witness projections encountered previously. More precisely, suppose CW' is a spurious Counterexample Witness projection seen before. Suppose that the set of sets of branches which can eliminate CW' are $\{B_1, \ldots, B_l\}$ and for $1 \leq i \leq l$, let $B_i = \{b_{i,1}, \ldots, b_{i,n_i}\}$. Let us then define the formula $\phi_{CW'}$ as follows:

$$\phi_{CW'} \equiv \bigvee_{1 \leq i \leq l} \left(\bigwedge_{1 \leq j \leq n_i} v_{i,j} \right)$$

Note that since the elements of each B_i are branches of $\mathcal{C}$, the Boolean variables in $\phi_{CW'}$ are from the set BV described earlier. Now consider any satisfying solution s to $\phi_{CW'}$ such that s assigns the variables $\{v_1, \ldots, v_m\}$ to TRUE. It should be obvious from the above definition that the set of branches $\{b_1, \ldots, b_m\}$ suffices to eliminate CW'. Now suppose that the set of all spurious Counterexample Witness projections seen previously is $\widehat{CW}$. Then the formula ϕ is defined as:

$$\phi \equiv \bigwedge_{CW' \in \widehat{CW}} \phi_{CW'}$$

Now consider any satisfying solution s to ϕ such that s assigns the variables $\{v_1, \ldots, v_m\}$ to TRUE. Once again, it should be obvious from the above definition that the set of branches $\{b_1, \ldots, b_m\}$ suffices to eliminate every element of $\widehat{CW}$.

Example 18 *Suppose $\mathcal{C}$ has four branches $\{b_1, b_2, b_3, b_4\}$. Hence there are four Boolean variables $\{v_1, v_2, v_3, v_4\}$. Consider a* Counterexample Witness *projection CW_1 such that CW_1 is eliminated by either branch b_1 alone or by branches b_2, b_3 and b_4 together. Hence the Boolean formula ϕ_{CW_1} is $(v_1) \bigvee (v_2 \wedge v_3 \wedge v_4)$.*

Again consider another Counterexample Witness *projection CW_2 such that CW_2 is eliminated by either branch b_4 alone or by branches b_1, b_2 and b_3 together. Hence the Boolean formula ϕ_{CW_2} is $(v_4) \bigvee (v_1 \wedge v_2 \wedge v_3)$.*

Let the set of Counterexample Witness *projections under consideration be* $\widehat{CW} = \{CW_1, CW_2\}$. *Then the formula* $\phi = \phi_{CW_1} \bigwedge \phi_{CW_2} = ((v_1) \vee (v_2 \wedge v_3 \wedge v_4)) \bigwedge ((v_4) \vee (v_1 \wedge v_2 \wedge v_3))$.

There are many satisfying solutions to ϕ. *For instance one solution is* $(v_1 = \text{FALSE}, v_2 = \text{TRUE}, v_3 = \text{TRUE}, v_4 = \text{TRUE})$. *This means that the set of branches* $\{b_2, b_3, b_4\}$ *is sufficient to eliminate both* CW_1 *and* CW_2.

Yet another solution is $(v_1 = \text{TRUE}, v_2 = \text{FALSE}, v_3 = \text{FALSE}, v_4 = \text{TRUE})$ *indicating that the set of branches* $\{b_1, b_4\}$ *is also sufficient to eliminate both* CW_1 *and* CW_2. *In fact this is a minimal such set.*

Algorithm **AbsRefMin** first updates ϕ by adding clauses corresponding to the new spurious Counterexample Witness projection CW. It then solves for the satisfiability of ϕ along with the pseudo-Boolean constraint $\varphi \equiv \Sigma_{i=1}^{k} v_i$. Since the solution satisfies ϕ and minimizes φ, it clearly corresponds to a minimal set of branches which can eliminate all previous spurious Counterexample Witness projections as well as CW.

Note that ϕ is updated in place so that the next invocation of **AbsRefMin** uses the updated ϕ. Also note that if ϕ is unsatisfiable, it means that there is at least one spurious Counterexample Witness projection which cannot be eliminated by any set of branches. In other words, there is some CW for which ϕ_{CW} is equivalent to FALSE. In this case **AbsRefMin** returns ERROR.

In order to compute ϕ_{CW}, **AbsRefMin** naively iterates over every subset of $\mathcal{B}_{\mathcal{C}}$. However this results in its complexity being exponential in the size of $\mathcal{B}_{\mathcal{C}}$. Below we list several ways to reduce the number of subsets attempted by **AbsRefMin**:

- Limit the cardinality or number of attempted subsets to a small constant, e.g.

Procedure 7.1 AbsRefMin returns a refined abstraction for $\mathcal{C}$ based on a minimal set of branches that eliminates a set of spurious Counterexample Witness projections and ERROR on failure. The parameter ϕ initially expresses constraints about branches which can eliminate all previous spurious Counterexample Witness projections. **AbsRefMin** also updates ϕ with the constraints for the new spurious Counterexample Witness projection CW.

```
Algorithm AbsRefMin($CW, \mathcal{C}, \phi, \gamma$)
- $CW$ : is a spurious Counterexample Witness, $\phi$ : is a Boolean formula
- $\mathcal{C}$ : is a component, $\gamma$ : is a context for $\mathcal{C}$
let $CW = (S, Init, AP, L, \Sigma, T)$;
$\phi_{CW}$ := FALSE;
for each $B \subseteq \mathcal{B}_{\mathcal{C}}$    //$\mathcal{B}_{\mathcal{C}}$ is the set of branches in $\mathcal{C}$
    $\Pi$ := PredInfer($\mathcal{C}, \gamma, B$);    //$\Pi$ is set of predicates inferred from $B$
    let $\widehat{M} := [\![\mathcal{C}]\!]^{\Pi}_{\gamma} = \left(\widehat{S}, \widehat{Init}, \widehat{AP}, \widehat{L}, \widehat{\Sigma}, \widehat{T}\right)$;
      //$\widehat{M}$ is the predicate abstraction of $\mathcal{C}$ using $\Pi$
    if (AbsCanSimul($CW, Init, \widehat{M}) \cap \widehat{Init} = \emptyset$) then $\phi_{CW} := \phi_{CW} \vee \bigwedge_{b_i \in B} v_i$;
      //$CW \not\precsim \widehat{M}$, and hence $B$ can eliminate $CW$
$\phi := \phi \wedge \phi_{CW}$;    //update $\phi$ with the constraints for $CW$
invoke PBS to solve $(\phi, \Sigma_{i=1}^{k} v_i)$;
if $\phi$ is unsatisfiable then return ERROR; //no set of branches can eliminate $CW$
else let $s_{opt}$ = solution returned by PBS;
let $\{v_1, \ldots, v_m\}$ = variables assigned TRUE by $s_{opt}$ and $\overline{B} = \{b_1, \ldots, b_m\}$;
$\Pi$ := PredInfer($\mathcal{C}, \gamma, \overline{B}$); //$\Pi$ is the set of predicates inferred from $\overline{B}$
return $[\![\mathcal{C}]\!]^{\Pi}_{\gamma}$; //return the predicate abstraction of $\mathcal{C}$ using $\Pi$
```

5, assuming that most Counterexample Witness projections can be eliminated by a small set of branches.

- Stop after reaching a certain size of subsets if any eliminating solutions have been found.

- Break up the control flow graph of $\mathcal{C}$ into blocks and only consider subsets of branches within blocks (keeping subsets in other blocks fixed).

- Use data flow analysis to only consider subsets of related branches.

- For any CW, if a set p eliminates CW, ignore all supersets of p with respect to CW (as we are seeking a minimal solution).

In our experiments we used some of the above techniques. Details are presented in Section 7.5. In conclusion, we note that other techniques for solving this optimization problem are also possible, including minimal hitting sets and logic minimization. The PBS step, however, has not been a bottleneck in any of our experiments.

7.4 CEGAR with Predicate Minimization

In this section we present the complete CEGAR algorithm for simulation conformance that uses **AbsRefMin** instead of **AbsRefine** for abstraction refinement. The algorithm is called **SimulCEGARMin** and is presented in Procedure 7.2. It is similar to **SimulCEGAR** except that instead of maintaining the set of spurious Counterexample Witness projections CW_i for each component $\mathcal{C}_i$ it maintains the formula ϕ_i. The proof of its correctness is also similar to that of **SimulCEGAR**.

7.5 Experimental Results

We implemented our technique inside MAGIC and experimented with a variety of benchmarks. Each benchmark consisted of an implementation (a C program) and a specification (provided separately as an LKS). All of the experiments were carried out on an AMD Athlon 1600 XP machine with 900 MB RAM running RedHat 7.1. In this section we describe our results in the context of checking the effectiveness of predicate minimization. We also present results comparing our predicate minimization scheme with a greedy predicate minimization strategy implemented on top of MAGIC.

Procedure 7.2 SimulCEGARMin checks simulation conformance between a program $\mathcal{P}$ and a specification Sp in a context Γ.

```
**Algorithm SimulCEGARMin**$(\mathcal{P}, Sp, \Gamma)$
- $\mathcal{P}$ : is a program, $\Gamma$ : is a context for $\mathcal{P}$
- $Sp$ : is a specification LKS
**let** $\mathcal{P} = \langle \mathcal{C}_1, \dots, \mathcal{C}_n \rangle$ **and** $\Gamma = \langle \gamma_1, \dots, \gamma_n \rangle$;
**for each** $i \in \{1, \dots, n\}$
    $\Pi_i :=$ **PredInfer**$(\mathcal{C}_i, \gamma_i, \emptyset)$ **and** $\widehat{M_i} := [\![\mathcal{C}_i]\!]_{\gamma_i}^{\Pi_i}$ **and** $\phi_i :=$ TRUE;
      //$\widehat{M_i}$ = initial predicate abstractions of $\mathcal{C}_i$ with empty set of seed branches
**forever do**
    **let** $\widehat{M} = \widehat{M_1} \parallel \cdots \parallel \widehat{M_n}$;
      //$\widehat{M}$ is the composition of predicate abstractions
    **if** (**SimulWitness**$(\widehat{M}, Sp)$ = "$\widehat{M} \preccurlyeq Sp$") **return** "$\mathcal{P} \preccurlyeq Sp$";
      //if the property holds on $\widehat{M}$ it also holds on $\mathcal{P}$
    **let** $CW$ = Counterexample Witness returned by **SimulWitness**;
    **find** $i \in \{1, \dots, n\}$ **such that** $\neg$**WeakSimul**$(CW \downharpoonleft \gamma_i, \mathcal{C}_i, \gamma_i)$;
      //check compositionally if $CW$ is spurious
    **if (no such** $i$ **found) return** "$\mathcal{P} \not\preccurlyeq Sp$";
      //$CW$ is valid and hence $\mathcal{P}$ is not simulated by $Sp$
    **if** (**AbsRefMin**$(CW \downharpoonleft \gamma_i, \mathcal{C}_i, \phi_i, \gamma_i)$ = ERROR) **return** ERROR;
      //no set of branches can eliminate $CW \downharpoonleft \gamma_i$
    $\widehat{M_i} :=$ **AbsRefMin**$(CW \downharpoonleft \gamma_i, \mathcal{C}_i, \phi_i, \gamma_i)$;    //refine the abstraction and repeat
```

7.5.1 The Greedy Approach

In each iteration, this greedy strategy first adds predicates sufficient to eliminate the spurious Counterexample Witness to its candidate branch set B. Then it attempts to reduce the size of the resulting B by using the algorithm **GreedyMin** described in Procedure 7.3. The advantage of this approach is that it requires only a small overhead (polynomial) compared to **AbsRefMin**, but on the other hand it does not guarantee an optimal result. Further, we performed experiments with the BLAST [66] tool. BLAST also takes C programs as input, and uses a variation of the standard CEGAR loop based on *lazy abstraction*, but without minimization. Lazy abstraction

Procedure 7.3 GreedyMin returns a greedily computed refined abstraction for $\mathcal{C}$ that eliminates every spurious Counterexample Witness projection in CW.

```
Algorithm GreedyMin($CW, \mathcal{C}, \gamma$)
- $CW$ : is a set of Counterexample Witnesses
- $\mathcal{C}$ : is a component, $\gamma$ : is a context for $\mathcal{C}$
$B := \mathcal{B}_{\mathcal{C}}$;   //start with the set of all branches in $\mathcal{C}$
//repeatedly try to remove elements from $B$ while maintaining the invariant that
//every spurious Counterexample Witness projection in $CW$ can still be eliminated
Loop:
    Create random ordering $\{b_1, \ldots, b_k\}$ of $B$;
    for $i = 1$ to $k$
        $B' := B \setminus \{b_i\}$;   //check if $b_i$ is redundant
        if $B'$ can eliminate all elements of $CW$
            $B := B'$;   //$b_i$ is redundant and can be eliminated
            goto Loop;
$\Pi :=$ PredInfer($\mathcal{C}, \gamma, B$);   //$\Pi$ is set of predicates inferred from $B$
return $[\![\mathcal{C}]\!]_{\gamma}^{\Pi}$;   //return the predicate abstraction of $\mathcal{C}$ using $\Pi$
```

refines an abstract model while allowing different degrees of abstraction in different parts of a program, without requiring recomputation of the entire abstract model in each iteration. Laziness and predicate minimization are, for the most part, orthogonal techniques. In principle a combination of the two might produce better results than either in isolation.

7.5.2 Benchmarks

We used two kinds of benchmarks. A small set of relatively simple benchmarks were derived from the examples supplied with the BLAST distribution and regression tests for MAGIC. The difficult benchmarks were derived from the C source code of OpenSSL. A critical component of this protocol is the initial *handshake* between a server and a client. We verified different properties of the main routines that implement the

	MAGIC + GREEDY				MAGIC + MINIMIZE			
Program	Time	Iter	Pred	Mem	Time	Iter	Pred	Mem
funcall-nested	6	2	10/9/1	×	**5**	2	10/9/1	×
fun_lock	**5**	5	8/3/3	×	6	4	8/3/3	×
driver.c	5	5	6/2/4	×	5	5	6/2/4	×
read.c	6	3	15/5/1	×	**5**	2	15/5/1	×
socket-y-01	**5**	3	12/4/2	×	6	3	12/4/2	×
opttest.c	**150**	5	4/4/4	63	247	25	4/4/4	63
ssl-srvr-1	*	103	16/3/5	51	**226**	14	5/4/2	38
ssl-srvr-2	2106	62	8/4/3	34	**216**	14	5/4/2	38
ssl-srvr-3	*	100	22/3/7	53	**200**	12	5/4/2	38
ssl-srvr-4	8465	69	14/4/5	56	**170**	9	5/4/2	38
ssl-srvr-5	*	117	23/5/9	56	**205**	13	5/4/2	36
ssl-srvr-6	*	84	22/4/8	337	**359**	14	8/4/3	89
ssl-srvr-7	*	99	19/3/6	62	**196**	11	5/4/2 S	38
ssl-srvr-8	*	97	19/4/7	142	**211**	10	8/4/3	40
ssl-srvr-9	8133	99	11/4/4	69	**316**	20	11/4/4	38
ssl-srvr-10	*	97	12/3/4	77	**241**	14	8/4/3	38
ssl-srvr-11	*	87	26/4/9	65	**356**	24	8/4/3	38
ssl-srvr-12	*	122	23/4/8	180	**301**	17	8/4/3	42
ssl-srvr-13	*	106	19/4/7	69	**436**	29	11/4/4	38
ssl-srvr-14	*	115	18/3/6	254	**406**	20	8/4/3	52
ssl-srvr-15	2112	37	8/4/3	118	**179**	7	8/4/3	40
ssl-srvr-16	*	103	22/3/7	405	**356**	17	8/4/3	58
ssl-clnt-1	225	27	5/4/2	20	**156**	12	5/4/2	31
ssl-clnt-2	1393	63	5/4/2	23	**185**	18	5/4/2	29
ssl-clnt-3	*	136	29/4/10	28	**195**	21	5/4/2	29
ssl-clnt-4	**152**	29	5/4/2	20	191	19	5/4/2	29
TOTAL	163163	1775	381/102 /129	2182	**5375**	356	191/107 /67	880
AVERAGE	6276	68	15/4/5	104	**207**	14	7/4/3	42

Table 7.1: Comparison of MAGIC with the greedy approach. '*' indicates run-time longer than 3 hours. '×' indicates negligible values. Best results are emphasized.

handshake. The names of benchmarks that are derived from the server routine and client routine begin with *ssl-srvr* and *ssl-clnt* respectively. In all our benchmarks, the properties are satisfied by the implementation. Note that all these benchmarks involved purely sequential C code.

7.5.3 Results Summary

Table 7.1 summarizes the comparison of our predicate minimization strategy with the greedy approach. Time consumptions are given in seconds. For predicate minimization, instead of solving the full optimization problem, we simplified the

	MAGIC				MAGIC + MINIMIZE			
Program	Time	Iter	Pred	Mem	Time	Iter	Pred	Mem
funcall-nested	5	2	10/9/1	×	5	2	10/9/1	×
fun_lock	**5**	4	8/3/3	×	6	4	8/3/3	×
driver.c	6	5	6/2/4	×	**5**	5	6/2/4	×
read.c	5	2	15/5/2	×	5	2	15/5/1	×
socket-y-01	**5**	3	12/4/2	×	6	3	12/4/2	×
opttest.c	**145**	5	7/7/8	63	247	25	4/4/4	63
ssl-srvr-1	250	12	56/5/22	43	**226**	14	5/4/2	38
ssl-srvr-2	752	16	72/6/30	72	**216**	14	5/4/2	38
ssl-srvr-3	331	12	56/5/22	47	**200**	12	5/4/2	38
ssl-srvr-4	677	14	63/6/26	72	**170**	9	5/4/2	38
ssl-srvr-5	**71**	5	22/4/8	24	205	13	5/4/2	36
ssl-srvr-6	11840	23	105/11/44	1187	**359**	14	8/4/3	89
ssl-srvr-7	2575	20	94/7/38	192	**196**	11	5/4/2 S	38
ssl-srvr-8	**130**	8	32/5/14	58	211	10	8/4/3	40
ssl-srvr-9	2621	15	65/8/28	183	**316**	20	11/4/4	38
ssl-srvr-10	561	16	75/6/30	73	**241**	14	8/4/3	38
ssl-srvr-11	4014	19	89/8/36	287	**356**	24	8/4/3	38
ssl-srvr-12	7627	22	102/9/42	536	**301**	17	8/4/3	42
ssl-srvr-13	3127	17	75/9/32	498	**436**	29	11/4/4	38
ssl-srvr-14	7317	22	102/9/42	721	**406**	20	8/4/3	52
ssl-srvr-15	615	15	81/28/5	188	**179**	7	8/4/3	40
ssl-srvr-16	3413	21	98/8/40	557	**356**	17	8/4/3	58
ssl-clnt-1	**110**	10	43/4/18	25	156	12	5/4/2	31
ssl-clnt-2	**156**	11	53/5/20	31	185	18	5/4/2	29
ssl-clnt-3	421	13	52/7/24	58	**195**	21	5/4/2	29
ssl-clnt-4	**125**	10	35/5/18	27	191	19	5/4/2	29
TOTAL	46904	322	1428/185 /559	4942	**5375**	356	191/107 /67	880
AVERAGE	1804	12	55/7/22	235	**207**	14	7/4/3	42

Table 7.2: Results for MAGIC with and without minimization. '*' indicates run-time longer than 3 hours. '×' indicates negligible values. Best results are emphasized.

problem as described in section 7.3. In particular, for each trace we only considered the first 1,000 combinations and only generated 20 eliminating combinations. The combinations were considered in increasing order of size. After all combinations of a particular size had been tried, we checked whether at least one eliminating combination had been found. If so, no further combinations were tried. In the smaller examples we observed no loss of optimality due to these restrictions. We also studied the effect of altering these restrictions on the larger benchmarks and we report our findings later.

Table 7.2 shows the improvement observed in MAGIC upon using predicate minimization while Table 7.3 shows the comparison between predicate minimization

	BLAST				MAGIC + MINIMIZE			
Program	Time	Iter	Pred	Mem	Time	Iter	Pred	Mem
funcall-nested	**1**	3	13/10	×	5	2	10/9/1	×
fun_lock	**5**	7	7/7	×	6	4	8/3/3	×
driver.c	**1**	4	3/2	×	5	5	6/2/4	×
read.c	6	11	20/11	×	**5**	2	15/5/1	×
socket-y-01	**5**	13	16/6	×	6	3	12/4/2	×
opttest.c	7499	38	37/37	231	**247**	25	4/4/4	63
ssl-srvr-1	2398	16	33/8	175	**226**	14	5/4/2	38
ssl-srvr-2	691	13	68/8	60	**216**	14	5/4/2	38
ssl-srvr-3	1162	14	32/7	103	**200**	12	5/4/2	38
ssl-srvr-4	284	11	27/5	44	**170**	9	5/4/2	38
ssl-srvr-5	1804	19	52/5	71	**205**	13	5/4/2	36
ssl-srvr-6	*	39	90/10	805	**359**	14	8/4/3	89
ssl-srvr-7	359	11	76/9	37	**196**	11	5/4/2 S	38
ssl-srvr-8	*	25	35/5	266	**211**	10	8/4/3	40
ssl-srvr-9	337	10	76/9	36	**316**	20	11/4/4	38
ssl-srvr-10	8289	20	35/8	148	**241**	14	8/4/3	38
ssl-srvr-11	547	11	78/11	51	**356**	24	8/4/3	38
ssl-srvr-12	2434	21	80/8	120	**301**	17	8/4/3	42
ssl-srvr-13	608	12	79/12	54	**436**	29	11/4/4	38
ssl-srvr-14	10444	27	84/10	278	**406**	20	8/4/3	52
ssl-srvr-15	*	31	38/5	436	**179**	7	8/4/3	40
ssl-srvr-16	*	33	87/10	480	**356**	17	8/4/3	58
ssl-clnt-1	348	16	28/5	43	**156**	12	5/4/2	31
ssl-clnt-2	523	15	28/4	52	**185**	18	5/4/2	29
ssl-clnt-3	469	14	29/5	49	**195**	21	5/4/2	29
ssl-clnt-4	380	13	27/4	45	**191**	19	5/4/2	29
TOTAL	81794	447	1178/221	3584	**5375**	356	191/107 /67	880
AVERAGE	3146	17	45/9	171	**207**	14	7/4/3	42

Table 7.3: Results for BLAST and MAGIC with predicate minimization. '*' indicates run-time longer than 3 hours. '×' indicates negligible values. Best results are emphasized.

and BLAST. Once again, time consumptions are reported in seconds. The column Iter reports the number of iterations through the CEGAR loop necessary to complete the proof. Predicates are listed differently for the two tools. For BLAST, the first number is the total number of predicates discovered and used and the second number is the number of predicates active at any one point in the program (due to lazy abstraction this may be smaller). In order to force termination we imposed a limit of three hours on the running time. We denote by '*' in the Time column examples that could not be solved in this time limit. In these cases the other columns indicate relevant measurements made at the point of forceful termination.

For MAGIC, the first number is the total number of expressions used to prove the property. The number of predicates (the second number) may be smaller, as MAGIC combines multiple mutually exclusive expressions (e.g., $x == 1$, $x < 1$, and $x > 1$) into a single, possibly non-binary predicate, having a number of values equal to the number of expressions (plus one, if the expressions do not cover all possibilities). The final number for MAGIC is the size of the final set of branches. For experiments in which memory usage was large enough to be a measure of state-space size rather than overhead, we also report memory usage (in megabytes).

For the smaller benchmarks, the various abstraction refinement strategies do not differ markedly. However, for our larger examples derived from OpenSSL, the refinement strategy is of considerable importance. Predicate minimization, in general, reduced verification time (though there were a few exceptions to this rule, the average running time was considerably lower than for the other techniques, even with the cutoff on the running time). Moreover, predicate minimization reduced the memory needed for verification, which is an even more important bottleneck. Given that the memory was cutoff in some cases for other techniques before verification was complete, the results are even more compelling.

The greedy approach kept memory use fairly low, but almost always failed to find near-optimal predicate sets and converged much more slowly than the usual monotonic refinement or predicate minimization approaches. Further, it is not clear how much final memory usage would be improved by the greedy strategy if it were allowed to run to completion. Another major drawback of the greedy approach is its unpredictability. We observed that on any particular example, the greedy strategy might or might not complete within the time limit in different executions. Clearly, the order in which this strategy tries to eliminate predicates in each iteration is very critical to its success. Given that the strategy performs poorly on most of our benchmarks using a random ordering, more sophisticated ordering techniques may perform better.

		ssl-srvr-4						ssl-srvr-15					
ELM	SUB	Ti	It	Br	M	T	G	Ti	It	Br	M	T	G
50	250	656	8	2	64	34	1	1170	15	3	72	86	1
100	250	656	8	2	64	34	1	1169	15	3	72	86	1
150	250	657	8	2	64	34	1	1169	15	3	72	86	1
200	250	656	8	2	64	34	1	1170	15	3	72	86	1
250	250	656	8	2	64	34	1	1168	15	3	72	86	1

		ssl-clnt-1					
ELM	SUB	Ti	It	Br	M	T	G
50	250	1089	13	2	67	66	1
100	250	1089	13	2	67	66	1
150	250	1090	13	2	67	66	1
200	250	1089	13	2	67	66	1
250	250	1090	13	2	67	66	1

Table 7.4: Results for optimality. ELM = MAXELM, SUB = MAXSUB, Ti = Time in seconds, It = number of iterations, Br = number of branches, M = Memory, T = total number of eliminating subsets generated, and G = maximum size of any eliminating subset generated.

7.5.4 Optimality

We experimented with two of the parameters that affect the optimality of our predicate minimization algorithm: (i) the maximum number of examined subsets (MAXSUB) and (ii) the maximum number of eliminating subsets generated (MAXELM) (that is, the procedure stops the search if MAXELM eliminating subsets were found, even if less than MAXSUB combinations were tried). We first kept MAXSUB fixed and took measurements for different values of MAXELM on a subset of our benchmarks viz. ssl-srvr-4, ssl-srvr-15 and ssl-clnt-1. Our results, shown in Table 7.4, clearly indicate that the optimality is practically unaffected by the value of MAXELM.

Next we experimented with different values of MAXSUB (the value of MAXELM was set equal to MAXSUB). The results we obtained are summarized in Table 7.5. It appears that, at least for our benchmarks, increasing MAXSUB leads only to increased

	ssl-srvr-4					ssl-srvr-15				
SUB	Time	It	Br	Mem	T/M/G	Time	It	Br	Mem	T/M/G
100	262	8	2	44	34/2/1	396	12	3	50	62/2/1
200	474	7	2	57	27/2/1	917	14	3	65	81/2/1
400	1039	9	2	71	38/2/1	1110	8	3	76	45/2/1
800	2182	7	2	165	25/2/1	2797	9	3	148	51/2/1
1600	6718	9	2	410	35/3/1	10361	11	3	410	76/3/1
3200	13656	9	2	461	40/3/1	14780	9	3	436	50/3/1
6400	26203	9	2	947	31/3/1	33781	10	3	792	51/3/1

	ssl-clnt-1				
SUB	Time	It	Br	Mem	T/M/G
100	310	11	2	40	58/2/1
200	683	12	2	51	63/2/1
400	2731	13	2	208	67/3/1
800	5843	14	2	296	75/3/1
1600	13169	12	2	633	61/3/1
3200	36155	12	2	1155	67/4/1
6400	> 57528	4	1	2110	22/4/1

Table 7.5: Results for optimality. SUB = MAXSUB, Time is in seconds, It = number of iterations, Br = number of branches, T = total number of eliminating subsets generated, M = maximum size of subsets tried, and G = maximum size of eliminating subsets generated.

execution time without reduced memory consumption or number of predicates. The additional number of combinations attempted or constraints allowed does not lead to improved optimality. The most probable reason is that, as shown by our results, even though we are trying more combinations, the actual number or maximum size of eliminating combinations generated does not increase significantly. Indeed, if this is a feature of most real-life programs, it would allow us, in most cases, to achieve near optimality by trying out only a small number of combinations or only combinations of small size.

Chapter 8

State-Event Temporal Logic

In this chapter, we present an expressive linear temporal logic called SE-LTL for specifying state-event based properties. We also discuss a compositional CEGAR framework for verifying concurrent software systems against SE-LTL formulas. Control systems ranging from smart cards to automated flight controllers are increasingly being incorporated within complex software systems. In many instances, errors in such systems can have catastrophic consequences, hence the urgent need to be able to ensure and guarantee their correctness. In this endeavor, the well-known methodology of *model checking* [32, 35, 39, 99] holds much promise. Although most of its early applications dealt with hardware and communication protocols, model checking is increasingly being used to verify software systems [5, 6, 13, 23, 24, 44, 65, 66, 98, 107, 111, 112] as well.

Unfortunately, applying model checking to software is complicated by several factors, ranging from the difficulty to model computer programs—due to the complexity of programming languages as compared to hardware description languages—to difficulties in specifying meaningful properties of software using the usual temporal logical formalisms of model checking. A third reason is the state space explosion problem, whereby the complexity of verifying an implementation against a specification becomes prohibitive.

The most common instantiations of model checking to date have focused on finite-state models and either branching-time (CTL [32]) or linear (LTL [78]) temporal logics. To apply model checking to software, it is necessary to specify (often complex) properties on the finite-state abstracted models of computer programs. The difficulties in doing so are even more pronounced when reasoning about *modular* software, such as concurrent or component-based sequential programs. Indeed, in modular programs, communication among modules proceeds via actions (or events), which can represent function calls, requests and acknowledgments, etc. Moreover, such communication is commonly data-dependent. Software behavioral claims, therefore, are often specifications defined over combinations of program actions and data valuations.

Existing modeling techniques usually represent finite-state machines as finite annotated directed graphs, using either *state-based* or *event-based* formalisms. It is well-known that theoretically the two frameworks are interchangeable. For instance, an action can be encoded as a change in a state variable, and likewise one can equip a state with different actions to reflect different values of its internal variables. However, converting from one representation to the other often leads to a significant enlargement of the state space. Moreover, neither approach on its own is practical when it comes to modular software, in which actions are often data-dependent: considerable domain expertise is then required to annotate the program and to specify proper claims.

This chapter, therefore, presents a framework in which both state-based and action-based properties can be expressed, combined, and verified. The modeling framework consists of *labeled Kripke structures* (LKS), which are directed graphs in which states are labeled with atomic propositions and transitions are labeled with actions. The specification logic is a *state/event* derivative of LTL. This allows us to represent both software implementations and specifications directly without any program annotations or privileged insights into program execution. We further show that an efficient model checking algorithm can be applied to help reason about state/event-based systems. Significantly, our model checking algorithm operates directly on the LKS models and is thus able to avoid the extra cost involved in translating state/event systems into systems that are based purely either states or events. We have implemented our approach within the C verification tool MAGIC [23, 24, 80], and report promising results in the examples which we have tackled.

The state/event-based formalism presented here is suitable for both sequential and concurrent systems, and is also amenable to the compositional abstraction refinement procedures [23] presented earlier. These procedures are embedded within a CEGAR framework [37], one of the core features of MAGIC. CEGAR lets us investigate the validity of a given specification through a sequence of increasingly refined abstractions of our system, until the property is either established or a real counterexample is found. Moreover, thanks to compositionality, the abstraction, counterexample validation, and refinement steps can all be carried out component-wise, thereby alleviating the need to build the full state space of the distributed system.

We illustrate our state/event paradigm with a current surge protector example, and conduct further experiments with the source code for OpenSSL and μC/OS-II (a real-time operating system for embedded applications). In the case of the latter, we discovered several bugs, one of which was unknown to the developers of μC/OS-II. We contrast our approach with equivalent pure state-based and event-based alternatives, and demonstrate that the state/event methodology yields significant gains in state space size and verification time.

This chapter is organized as follows. In Section 8.1, we review and discuss related work. Section 8.2 presents the basic definitions and results needed for the presentation of our compositional CEGAR verification algorithm. In Section 8.3, we present our state/event specification formalism, based on linear temporal logic. We review standard automata-theoretic model checking techniques, and show how these can be adapted to the verification task at hand.

In Section 8.4, we illustrate these ideas by modeling a simple surge protector. We also contrast our approach with pure state-based and event-based alternatives, and show that both the resulting implementations and specifications are significantly more cumbersome. We then use MAGIC to check these specifications, and discover that the non-state/event formalisms incur significant time and space penalties during verification.

Section 8.5 details our SE-LTL verification algorithm for C programs while Section 8.6 presents the complete compositional CEGAR scheme for SE-LTL verification of concurrent C programs. Finally, in Section 8.7, we report on case studies in which we checked specifications on the source code for OpenSSL and μC/OS-II, which led us to the discovery of a bug in the latter.

8.1 Related Work

Counterexample guided abstraction refinement [37, 76], or CEGAR, is an iterative procedure whereby spurious counterexamples to a specification are repeatedly eliminated through incremental refinements of a conservative abstraction of the system. CEGAR has been used, among others, in [90] (in non-automated form), and [6, 23, 29, 40, 66, 77, 98].

Compositionality, which features centrally in our work, is broadly concerned with the preservation of properties under substitution of components in concurrent systems. It has been extensively studied, among others, in process algebra (e.g., [69, 85, 100]), in temporal logic model checking [64], and in the form of assume-guarantee reasoning [41, 67, 84]. The combination of CEGAR and compositional reasoning is a relatively new approach. In [10], a compositional framework for (non-automated) CEGAR over data-based abstractions is presented. This approach differs from ours in that communication takes place through shared variables (rather than blocking message-passing), and abstractions are refined by eliminating spurious transitions, rather than by splitting abstract states.

The idea of combining state-based and event-based formalisms is certainly not new [14]. De Nicola and Vaandrager [94], for instance, introduce *doubly labeled transition systems*, which are very similar to our LKSs. From the point of view of expressiveness, our state/event version of LTL is also subsumed by the modal mu-calculus [16, 74, 97], via a translation of LTL formulas into Büchi automata. However all these approaches are restricted to finite state systems.

Kindler and Vesper [73] propose a state/event-based temporal logic for Petri nets. They motivate their approach by arguing, as we do, that pure state-based or event-based formalisms lack expressiveness in important respects. Huth et. al. [72] also propose a state/event framework, and define rich notions of abstraction and refinement. In addition, they provide *may* and *must* modalities for transitions, and show how to perform efficient three-valued verification on such structures. They do not, however, provide an automated CEGAR framework, and it is not clear whether they have implemented and tested their approach. Giannakopoulou and Magee [62] define *fluent* propositions within a labeled transition system (LTS – essentially an LKS with an empty set of atomic propositions) context to express action-based linear-time properties. A fluent proposition is a property that holds after it is initiated by an action and ceases to hold when terminated by another action. This work exploits partial-order reduction techniques and has been implemented in the LTSA tool.

In a comparatively early paper, De Nicola et. al. [93] propose a process algebraic framework with an action-based version of CTL as specification formalism. Verification then proceeds by first translating the underlying LTSs of processes into Kripke structures and the action-based CTL specifications into equivalent state-based CTL formulas. At that point, a model checker is used to establish or refute the property. Dill [55, 56] defines *trace structures* as algebraic objects to model both hardware circuits and their specifications. Trace structures can handle equally well states or events, although usually not both at the same time. Dill's approach to verification is based on abstractions and compositional reasoning, albeit without an iterative counterexample-driven refinement loop.

In general, events (input signals) in circuits can be encoded via changes in state variables. Browne makes use of this idea in [18], which features a CTL* specification formalism. Browne's framework also features abstractions and compositional reasoning, in a manner similar to Dill's. Burch [20] extends the idea of trace structures into a full-blown theory of *trace algebra*. The focus here however is the modeling of discrete and continuous time, and the relationship between these two paradigms. This work also exploits abstractions and compositionality, however once again without automated counterexample guided refinements. Finally, Bultan [19] proposes an intermediate specification language lying between high-level Statechart-like formalisms and transition systems. Actions are encoded as changes in state variables in a framework which also focuses on exploiting compositionality in model checking.

8.2 Preliminaries

Recall, from Definition 1, that an LKS is a 6-tuple $(S, Init, AP, L, \Sigma, T)$. In this section we present a few preliminary definitions that will be used in the rest of the chapter.

Definition 20 (Infinite Path and Trace) *Let M be an LKS. An infinite path of M is an infinite sequence $\langle s_0, \alpha_0, s_1, \alpha_1, \ldots \rangle$ such that: (i) $s_0 \in Init_M$ and (ii) $\forall i \geq 0 \centerdot s_i \xrightarrow{\alpha_i}_M s_{i+1}$. In such a case, the infinite sequence $\langle L_M(s_0), \alpha_0, L_M(s_1), \alpha_1, \ldots \rangle$ is called an infinite trace of M.*

In the rest of this chapter we will only restrict our attention to infinite paths and traces. We will also assume that the transition relation of every LKS is total, i.e., every state has at least one outgoing transition. The notion of paths leads naturally to that of languages.

Definition 21 (Language) *Let M be an LKS. The language of M, denoted by $\mathcal{L}(M)$, is defined as: $\mathcal{L}(M) = \{\pi \mid \pi$ is a path of $M\}$.*

8.3 The Logic SE-LTL

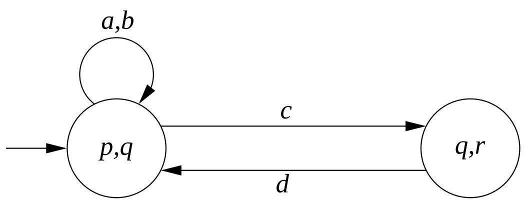
We now present a logic enabling us to refer easily to both states and events when constructing specifications. Given an LKS M, we consider linear temporal logic *state/event formulas* over the sets AP_M and Σ_M. Suppose p ranges over AP_M and α ranges over Σ_M. Then the syntax of SE-LTL can be defined inductively as follows:

$$\phi ::= p \mid \alpha \mid \neg\phi \mid \phi \wedge \phi \mid \mathbf{X}\phi \mid \mathbf{G}\phi \mid \mathbf{F}\phi \mid \phi \,\mathbf{U}\, \phi$$

We write SE-LTL to denote the resulting logic, and in particular to distinguish it from (standard) LTL. Let $\pi = \langle s_0, \alpha_0, s_1, \alpha_1, \ldots \rangle$ be a path of M. For $i \geq 0$, let π^i denote the suffix of π starting in state s_i. We then inductively define path-satisfaction of SE-LTL formulas as follows:

$$\begin{array}{rll}
\pi \models p & \text{iff} & p \in L_M(s_0) \\
\pi \models \alpha & \text{iff} & \alpha = \alpha_0 \\
\pi \models \neg\phi & \text{iff} & \pi \not\models \phi \\
\pi \models \phi_1 \wedge \phi_2 & \text{iff} & \pi \models \phi_1 \text{ and } \pi \models \phi_2 \\
\pi \models \mathbf{X}\phi & \text{iff} & \pi^1 \models \phi \\
\pi \models \mathbf{G}\phi & \text{iff} & \forall i \geqslant 0 \,.\, \pi^i \models \phi \\
\pi \models \mathbf{F}\phi & \text{iff} & \exists i \geqslant 0 \,.\, \pi^i \models \phi \\
\pi \models \phi_1 \,\mathbf{U}\, \phi_2 & \text{iff} & \exists i \geqslant 0 \,.\, \pi^i \models \phi_2 \text{ and } \forall 0 \leqslant j < i \,.\, \pi^j \models \phi_1
\end{array}$$

We then let $M \models \phi$ iff, for every path $\pi \in \mathcal{L}(M)$, $\pi \models \phi$. We also use the derived (weak until) $\mathbf{W}$ operator: $\phi_1 \,\mathbf{W}\, \phi_2 \equiv (\mathbf{G}\phi_1) \vee (\phi_1 \,\mathbf{U}\, \phi_2)$. As a simple example, consider the following LKS M. It has two states, the leftmost of which is the sole initial state. Its set of atomic state propositions is $\{p, q, r\}$; the first state is labeled with $\{p, q\}$ and the second with $\{q, r\}$. M's transitions are similarly labeled with sets of events drawn from the alphabet $\{a, b, c, d\}$.



As the reader may easily verify, $M \models \mathbf{G}(c \implies \mathbf{F}r)$ but $M \not\models \mathbf{G}(b \implies \mathbf{F}r)$. Note also that $M \models \mathbf{G}(d \implies \mathbf{F}r)$, but $M \not\models \mathbf{G}(d \implies \mathbf{XF}r)$.

8.3.1 Automata-based Verification

We aim to reduce SE-LTL verification problems to standard automata-theoretic techniques for LTL. Note that a standard—but unsatisfactory—way of achieving this is to explicitly encode actions through changes in (additional) state variables, and then proceed with LTL verification. Unfortunately, this technique usually leads to a significant blow-up in the state space, and consequently yields much larger verification times. The approach we present here, on the other hand, does *not* alter the size of the LKS, and is therefore considerably more efficient. We first recall some basic results about LTL, Kripke structures, and automata-based verification.

A Kripke structure is simply an LKS minus the alphabet and the transition-labeling function; as for LKSs, the transition relation of a Kripke structure is required to be total. An LTL formula is an SE-LTL formula which makes no use of events as atomic propositions.

Definition 22 (Kripke Structure) *A Kripke Structure (KS for short) is a 5-tuple* $(S, Init, AP, L, T)$ *where: (i)* S *is a non-empty set of states, (ii)* $Init \subseteq S$ *is a set* *of initial states, (iii) AP is a set of atomic propositions, (iv) $L : S \rightarrow 2^{AP}$ is a propositional labeling function that maps every state to a set of atomic propositions that are true in that state, and (v) $T \subseteq S \times S$ is a total transition relation.*

The notion of paths, traces and languages for Kripke Structures is analogous to those of Labeled Kripke Structures.

Definition 23 (Path and Trace) *Let M be a KS. A path of M is an infinite sequence $\langle s_0, s_1, \dots \rangle$ such that: (i) $s_0 \in Init_M$ and (ii) $\forall i \geq 0$, $s_i \longrightarrow_M s_{i+1}$. In such a case, the infinite sequence $\langle L_M(s_0), L_M(s_1), \dots \rangle$ is called a trace of M.*

Definition 24 (Language) *Let M be a KS. The language of M, denoted by $\mathcal{L}(M)$, is defined as: $\mathcal{L}(M) = \{\pi \mid \pi$ is a path of $M\}$.*

8.3.2 The Logic LTL

Given a KS M, we consider linear temporal logic (LTL) formulas over the set AP_M. Suppose p ranges over AP_M. Then the syntax of LTL can be defined inductively as follows:

$$\phi ::= p \mid \neg\phi \mid \phi \wedge \phi \mid \mathbf{X}\phi \mid \mathbf{G}\phi \mid \mathbf{F}\phi \mid \phi \ \mathbf{U} \ \phi$$

Let $\pi = \langle s_0, s_1, \ldots \rangle$ be a path of M. For $i \geq 0$, let π^i denote the suffix of π starting in state s_i. We then inductively define path-satisfaction of LTL formulas as follows:

$$\begin{aligned}
\pi \models p &\quad \text{iff} \quad p \in L_M(s_0) \\
\pi \models \neg\phi &\quad \text{iff} \quad \pi \not\models \phi \\
\pi \models \phi_1 \wedge \phi_2 &\quad \text{iff} \quad \pi \models \phi_1 \text{ and } \pi \models \phi_2 \\
\pi \models \mathbf{X}\phi &\quad \text{iff} \quad \pi^1 \models \phi \\
\pi \models \mathbf{G}\phi &\quad \text{iff} \quad \forall i \geqslant 0 \,\centerdot\, \pi^i \models \phi \\
\pi \models \mathbf{F}\phi &\quad \text{iff} \quad \exists i \geqslant 0 \,\centerdot\, \pi^i \models \phi \\
\pi \models \phi_1 \,\mathbf{U}\, \phi_2 &\quad \text{iff} \quad \exists i \geqslant 0 \,\centerdot\, \pi^i \models \phi_2 \text{ and } \forall 0 \leqslant j < i \,\centerdot\, \pi^j \models \phi_1
\end{aligned}$$

We then let $M \models \phi$ iff, for every path $\pi \in \mathcal{L}(M)$, $\pi \models \phi$. We also use the derived (weak until) $\mathbf{W}$ operator: $\phi_1 \;\mathbf{W}\; \phi_2 \equiv (\mathbf{G}\phi_1) \vee (\phi_1 \;\mathbf{U}\; \phi_2)$. We now define a Büchi automaton formally.

Definition 25 (Büchi Automaton) *A Büchi Automaton (BA for short) is a 6-tuple $(S, Init, AP, L, T, Acc)$ where: (i) S is a finite set of states, (ii) $Init \subseteq S$ is a set of initial states, (iii) AP is a finite set of atomic state propositions, (iv) $L : S \to 2^{2^{AP}}$ is a state-labeling function, (v) $T \subseteq S \times S$ is a transition relation, and (vi) $Acc \subseteq S$ is a set of accepting states.*

Note that the transition relation is not required to be total, and is moreover unlabeled. Note also that the states of a Büchi automaton are labeled with arbitrary functions over the set of atomic propositions AP. The idea is that each state of the BA corresponds to a *set* of valuations to AP. For example, suppose $AP = \{p_1, p_2, p_3\}$ and suppose that a state s of the BA is labeled with the function $p_1 \wedge \neg p_2$. Then the state s can *match* any state of a Kripke structure where p_1 is TRUE and p_2 is

FALSE (the value of p_3 can be either TRUE or FALSE). Thus a labeling of the state of a BA is a *constraint* and not an actual valuation to all the propositions. The ability to specify constraints enables us to construct Büchi automata with fewer number of states. The notion of paths and languages for Büchi automata are quite natural and we define them formally next.

Definition 26 (Path and Language) *Let B be a BA. A path of B is an infinite sequence $\langle s_0, s_1, \dots \rangle$ such that: (i) $s_0 \in Init_B$ and (ii) $\forall i \geq 0$, $s_i \longrightarrow_B s_{i+1}$. For a path π of B we denote by $Inf(\pi) \subseteq S_B$ the set of states of B which occur infinitely often in π. Then the language of B, denoted by $\mathcal{L}(B)$, is defined as: $\mathcal{L}(B) = \{\pi \mid \pi$ is a path of $B \wedge Inf(\pi) \cap Acc_B \neq \emptyset\}$.*

8.3.3 Product Automaton

Let M be a Kripke structure and B be a Büchi automaton such that $AP_M = AP_B$. We define the *standard* product $M \times B$ as a Büchi automata such that:

- $S_{M \times B} = \{(s, b) \in S_M \times S_B \mid L_M(s) \in L_B(b)\}$
- $Init_{M \times B} = \{(s, b) \in S_{M \times B} \mid s \in Init_M \wedge b \in Init_B\}$
- $AP_{M \times B} = AP_B$ and $\forall (s, b) \in S_{M \times B} \centerdot L_{M \times B}(s, b) = L_B(b)$
- $\forall (s, b) \in S_{M \times B} \centerdot \forall (s', b') \in S_{M \times B} \centerdot (s, b) \longrightarrow_{M \times B} (s', b')$ iff $s \longrightarrow_M s'$ and $b \longrightarrow_B b'$
- $Acc_{M \times B} = \{(s, b) \in S_{M \times B} \mid b \in Acc_B\}$

The non-symmetrical standard product $M \times B$ accepts exactly those paths of M which are *consistent* with B. Its main technical use lies in the following result of Gerth et. al. [61]:

Theorem 16 *Given a Kripke structure M and* LTL *formula ϕ, there is a Büchi automaton $B_{\neg\phi}$ such that $M \models \phi \iff \mathcal{L}(M \times B_{\neg\phi}) = \emptyset$.*

Theorem 16 is the core result that enables efficient LTL model checking. Given a Kripke structure M and an LTL formula ϕ we first construct the Büchi automaton $B_{\neg\phi}$. We then check if $\mathcal{L}(M \times B_{\neg\phi}) = \emptyset$ using a highly optimized double-depth-first-search algorithm [46, 71]. Finally, if $\mathcal{L}(M \times B_{\neg\phi}) = \emptyset$, we conclude that $M \models \phi$. Otherwise we conclude that $M \not\models \phi$.

An efficient tool to convert LTL formulas into optimized Büchi automata is Somenzi and Bloem's WRING [108, 113]. We now turn our attention back to labeled Kripke structures. Recall that SE-LTL formulas allow events in Σ_M to stand for atomic propositions. Therefore, given an SE-LTL formula ϕ over AP_M and Σ_M, we can interpret ϕ as an LTL formula over $AP_M \cup \Sigma_M$; let us denote the latter formula by $\phi^\flat$. $\phi^\flat$ is therefore syntactically identical to ϕ, but differs from ϕ in its semantic interpretation.

8.3.4 State/Event Product

We now define the *state/event product* of a labeled Kripke structure with a Büchi automaton. Let M be an LKS, and B be a Büchi automaton such that $AP_B = AP_M \cup \Sigma_M$. The state/event product $M \otimes B$ is a Büchi automaton that satisfies the following conditions:

- $S_{M\otimes B} = \{(s, b) \in S \times S_B \mid \exists \alpha \in \Sigma_M \centerdot L_M(s) \cup \{\alpha\} \in L_B(b)\}$.
- $Init_{M\otimes B} = \{(s, b) \in S_{M\otimes B} \mid s \in Init_M \wedge b \in Init_B\}$
- $AP_{M\otimes B} = AP_B$ and $\forall (s, b) \in S_{M\otimes B} \centerdot L_{M\otimes B}(s, b) = L_B(b)$

- $\forall (s,b) \in S_{M \otimes B} \centerdot \forall (s',b') \in S_{M \otimes B} \centerdot (s,b) \longrightarrow_{M \otimes B} (s',b')$ iff

$$\exists \alpha \in \Sigma_M \centerdot s \xrightarrow{\alpha}_M s' \wedge b \longrightarrow_B b' \wedge (L_M(s) \cup \{\alpha\}) \in L_B(b)$$

- $Acc_{M \otimes B} = \{(s,b) \in S_{M \otimes B} \mid b \in Acc_B\}$

The usefulness of a state/event product is captured by the following theorem. Note that the state/event product does *not* require an enlargement of the LKS M, even though we consider below just such an enlargement in the course of the proof of Theorem 17.

Theorem 17 *For any LKS M and* SE-LTL *formula ϕ, the following holds:*

$$M \models \phi \iff \mathcal{L}(M \otimes B_{\neg \phi^\flat}) = \emptyset$$

Proof. Observe that a state of M can have several differently-labeled transitions emanating from it. However, by duplicating states (and transitions) as necessary, we can transform M into another LKS M' having the following two properties: (i) $\mathcal{L}(M') = \mathcal{L}(M)$, and (ii) for every state s of M', the transitions emanating from s are all labeled with the same action. As a result, the validity of an SE-LTL atomic event proposition α in a given state of M' does not depend on the particular path to be taken from that state, and can therefore be recorded as a propositional state variable of the state itself. Formally, this gives rise to a Kripke structure M'' over atomic state propositions $AP_M \cup \Sigma_M$. We now claim that:

$$\mathcal{L}(M \otimes B_{\neg \phi^\flat}) = \emptyset \iff \mathcal{L}(M'' \times B_{\neg \phi^\flat}) = \emptyset. \tag{8.1}$$

To see this, notice first that there is a bijection between $\mu : \mathcal{L}(M) \to \mathcal{L}(M'')$. Next, observe that any path in $\mathcal{L}(M \otimes B_{\neg \phi^\flat})$ can be decomposed as a pair (π, β), where $\pi \in \mathcal{L}(M)$ and $\beta \in \mathcal{L}(B_{\neg \phi^\flat})$; likewise, any path in $\mathcal{L}(M'' \times B_{\neg \phi^\flat})$ can be decomposed as a pair (π'', β), where $\pi'' \in \mathcal{L}(M'')$ and $\beta \in \mathcal{L}(B_{\neg\phi^\flat})$. A straightforward inspection of the relevant definitions then reveals that $(\pi, \beta) \in \mathcal{L}(M \otimes B_{\neg\phi^\flat})$ iff $(\mu(\pi), \beta) \in \mathcal{L}(M'' \times B_{\neg\phi^\flat})$, which establishes our claim.

Finally, we clearly have $M \models \phi$ iff $M' \models \phi$ iff $M'' \models \phi^\flat$. Combining this with Theorem 16 and Equation 8.1 above, we get $M \models \phi \iff \mathcal{L}(M \otimes B_{\neg\phi^\flat}) = \emptyset$, as required.

□

The significance of Theorem 17 is that it enables us to make use of the highly optimized algorithms [46, 71] and tools [108] available for verifying LTL formulas on Kripke structures to verify SE-LTL specifications on *labeled* Kripke structures, at no additional cost.

8.3.5 SE-LTL **Counterexamples**

In case an LKS M does not satisfy an SE-LTL formula ϕ we will need to construct a counterexample so as to perform abstraction refinement. In this section, we present the notion of a counterexamples to an SE-LTL formula formally. We begin with a few well-known results.

Theorem 18 *Let M_1 and M_2 be two LKSs and ϕ be an* SE-LTL *formula. Then the following statement holds:*

$$M_1 \preccurlyeq M_2 \wedge M_2 \models \phi \implies M_1 \models \phi$$

Proof. Follows directly from the following two facts:

$$M_1 \preccurlyeq M_2 \implies \mathcal{L}(M_1) \subseteq \mathcal{L}(M_2)$$

$$M_2 \models \phi \implies \forall \pi \in \mathcal{L}(M_2) \centerdot \pi \models \phi$$

□

Theorem 19 *Let M_1 and M_2 be two LKSs and ϕ be an* SE-LTL *formula. Then the following statement holds:*

$$M_1 \preccurlyeq M_2 \wedge M_1 \not\models \phi \implies M_2 \not\models \phi$$

Proof. Directly from Theorem 18.

□

Theorem 19 leads to the notion of a witness for the non-entailment of a SE-LTL formula ϕ by an LKS M_2. It essentially says that an LKS M_1 is a witness for $M_2 \not\models \phi$ iff $M_1 \preccurlyeq M_2$ and $M_1 \not\models \phi$. Alternately such a witness M_1 can be viewed as a counterexample to $M_2 \models \phi$. In the rest of this chapter we will write Lasso to mean a counterexample for SE-LTL since such counterexamples are shaped like a lasso. We now present an algorithm for constructing Lassos.

Let M be an LKS and ϕ be an SE-LTL formula such that $M \not\models \phi$. From Theorem 17 we know that $\mathcal{L}(M \otimes B_{\neg\phi^\flat}) \neq \emptyset$. Let $\pi_\otimes = \langle (s_0, b_0), (s_1, b_1), \dots \rangle$ be an arbitrary element of $\mathcal{L}(M \otimes B_{\neg\phi^\flat})$. Clearly there exists a sequence of actions $\langle \alpha_0, \alpha_1, \dots \rangle$ such that the following two conditions hold: (i) $\langle s_0, \alpha_0, s_1, \alpha_1, \dots \rangle \in \mathcal{L}(M)$, and (ii) $\forall i \geq 0 \centerdot L_M(s_i) \cup \{\alpha_i\} \in L_{B_{\neg\phi^\flat}}(b_i)$. Let us denote the sequence $\langle s_0, \alpha_0, s_1, \alpha_1, \dots \rangle$ by π. Since M has a finite number of states and a finite alphabet (recall that in general M will be obtained by predicate abstraction of a program), π induces an LKS CE such that:

- $S_{CE} = \{s_i \mid i \geq 0\}$ $\quad Init_{CE} = \{s_0\}$ $\quad AP_{CE} = AP_M$ $\quad \Sigma_{CE} = \Sigma_M$

- $\forall s \in S_{CE} \centerdot L_{CE}(s) = L_M(s) \quad T_{CE} = \{(s_i, \alpha_i, s_{i+1}) \mid i \geq 0\}$

The significance of CE is captured by the following result which essentially states that CE is a Lasso for $M \not\models \phi$.

Theorem 20 *Let M be an LKS and ϕ be an* SE-LTL *formula. Suppose that $M \not\models \phi$ and let CE be the LKS as described above. Then the following holds:*

$$CE \preccurlyeq M \wedge CE \not\models \phi$$

Proof. To prove that $CE \preccurlyeq M$ we show that the relation $\mathcal{R} = \{(s, s) \mid s \in S_{CE}\}$ satisfies the following two conditions: (i) $\mathcal{R}$ is a simulation relation, and (ii) $\forall s_1 \in Init_{CE} \centerdot \exists s_2 \in Init_M \centerdot (s_1, s_2) \in \mathcal{R}$.

Recall that $\pi_\otimes = \langle (s_0, b_0), (s_1, b_1), \ldots \rangle$ was the arbitrary element of $\mathcal{L}(M \otimes B_{\neg\phi^\flat})$ used to construct CE. To prove that $CE \not\models \phi$ we show that $\pi_\otimes \in \mathcal{L}(CE \otimes B_{\neg\phi^\flat})$ and hence $\mathcal{L}(CE \otimes B_{\neg\phi^\flat}) \neq \emptyset$.

□

Let us denote by **ModelCheck** an algorithm which takes as input an LKS M and an SE-LTL formula ϕ. The algorithm checks for the emptiness of $\mathcal{L}(M \otimes B_{\neg\phi^\flat})$. If $\mathcal{L}(M \otimes B_{\neg\phi^\flat})$ is empty, it returns "$M \models \phi$". Otherwise it computes (as described above) and returns a Lasso CE for $M \not\models \phi$.

Theorem 21 *Algorithm* **ModelCheck** *is correct.*

Proof. Follows from Theorem 17 and Theorem 20.

□

8.4 A Surge Protector

We describe a safety-critical current surge protector in order to illustrate the advantages of state/event-based implementations and specifications over both the pure state-based and the pure event-based approaches. The surge protector is meant at all times to disallow changes in current beyond a varying threshold. The labeled Kripke structure in Figure 8.1 captures the main functional aspects of such a protector in which the possible values of the current and threshold are 0, 1, and 2. The threshold value is stored in the variable m and the value of the current is stored in variable c. Changes in threshold and current values are respectively communicated via the events $m0$, $m1$, $m2$, and $c0$, $c1$, $c2$.

Note, for instance, that when $m = 1$ the protector accepts changes in current to values 0 and 1, but not 2 (in practice, an attempt to hike the current up to 2 should trigger, say, a fuse and a jump to an emergency state, behaviors which are here abstracted away). The reader may object that we have only allowed for Boolean variables in our definition of labeled Kripke structures; it is however trivial to implement more complex types, such as bounded integers, as boolean encodings, and we have therefore elided such details here.

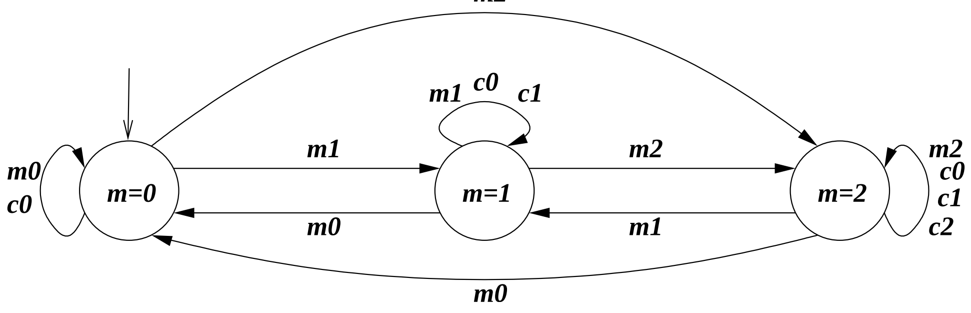


Figure 8.1: The LKS of a surge protector

The required specification is neatly captured as the following SE-LTL formula:

$$\phi_{\text{se}} = \mathbf{G}((c2 \implies m = 2) \wedge (c1 \implies (m = 1 \vee m = 2))).$$

By way of comparison, Figure 8.2 represents the (event-free) Kripke structure that captures the same behavior as the LKS of Figure 8.1. In this pure state-based formalism, nine states are required to capture all the reachable combinations of threshold ($m = i$) and last current changes ($c = j$) values. Note that the surge protector does not guarantee $c \leq m$. Indeed states where $c > m$ (e.g., $m = 1$ and $c = 2$) are reachable since the value of m can be decreased while keeping the value of c unchanged.

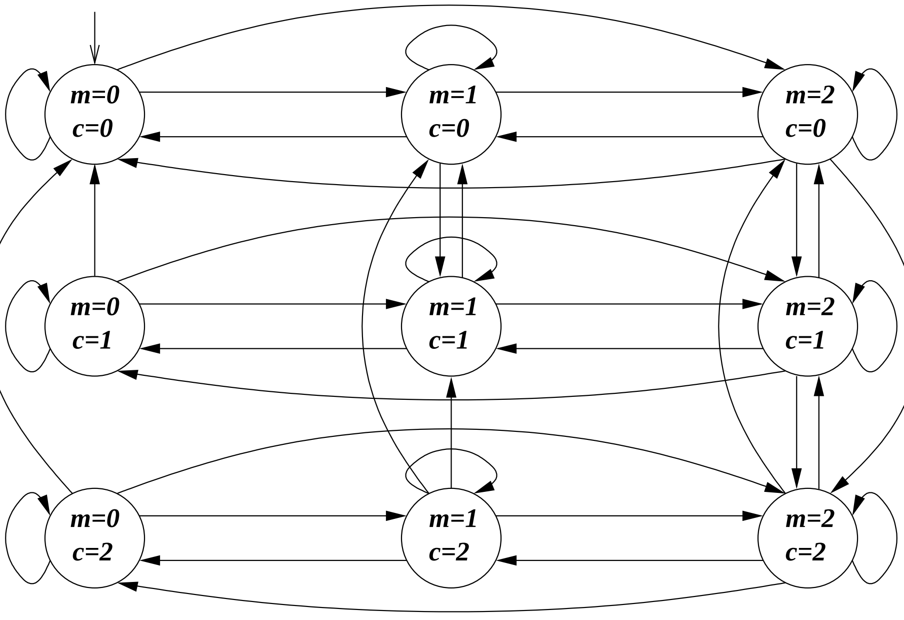


Figure 8.2: The Kripke structure of a surge protector

The data (9 states and 39 transitions) compares unfavorably with that of the LKS in Figure 8.1 (3 states and 9 transitions). Moreover, as the allowable current ranges increase, the number of states of the LKS will grow linearly, as opposed to quadratically for the Kripke structure. The number of transitions of both will grow quadratically, but with a roughly four-fold larger factor for the Kripke structure. These observations highlight the advantages of a state/event approach, which of course will be more or less pronounced depending on the type of system under consideration.

Another advantage of the state/event approach is witnessed when one tries to write down specifications. In this instance, the specification we require is

$$\begin{aligned}\phi_{\mathrm{s}} = \mathbf{G}(((c = 0 \vee c = 2) \wedge \mathbf{X}(c = 1)) \implies (m = 1 \vee m = 2)) \wedge \\ \mathbf{G}(((c = 0 \vee c = 1) \wedge \mathbf{X}(c = 2)) \implies m = 2),\end{aligned}$$

which is arguably significantly more complex than ϕ_{se}. The pure event-based specification ϕ_{e} capturing the same requirement is also clearly more complex than ϕ_{se}:

$$\begin{aligned}\phi_{\mathrm{e}} = \mathbf{G}(m0 &\implies ((\neg c1)\ \mathbf{W}\ (m1 \vee m2))) \wedge \\ \mathbf{G}(m0 &\implies ((\neg c2)\ \mathbf{W}\ m2)) \wedge \\ \mathbf{G}(m1 &\implies ((\neg c2)\ \mathbf{W}\ m2)).\end{aligned}$$

The greater simplicity of the implementation and specification associated with the state/event formalism is not purely a matter of aesthetics, or even a safeguard against subtle mistakes; experiments also suggest that the state/event formulation yields significant gains in both time and memory during verification. We implemented three parameterized instances of the surge protector as simple C programs, in one case allowing message passing (representing the LKS), and in the other relying solely on local variables (representing the Kripke structure). We also wrote corresponding specifications respectively as SE-LTL and LTL formulas (as above) and converted these into Büchi automata using the tool WRING [113]. Table 8.1 records the number of Büchi states and transitions associated with the specification, as well as the time

Range	Pure State				Pure Event				State/Event			
	St	Tr	B-T	T-T	St	Tr	B-T	T-T	St	Tr	B-T	T-T
2	4	5	253	383	6	10	245	320	3	4	184	252
3	8	12	270	545	12	23	560	674	4	6	298	407
4	14	23	492	1141	20	41	1597	1770	5	8	243	391
5	22	38	1056	2326	30	64	3795	4104	6	10	306	497
6	32	57	2428	4818	42	92	12077	12660	7	12	614	962
7	44	80	6249	10358	56	125	54208	55064	8	14	930	1321
8	58	107	17503	24603	72	163	372784	374166	9	16	2622	3133
9	74	138	55950	67553	$*$	$*$	$*$	$*$	10	18	8750	9488
10	92	173	195718	213969	$*$	$*$	$*$	$*$	11	20	33556	34503
11	$*$	$*$	$*$	$*$	$*$	$*$	$*$	$*$	12	22	135252	136500
12	$*$	$*$	$*$	$*$	$*$	$*$	$*$	$*$	13	24	534914	536451
13	$*$	$*$	$*$	$*$	$*$	$*$	$*$	$*$	$*$	$*$	$*$	$*$

Table 8.1: Comparison of pure state-based, pure event-based and state/event-based formalisms. Values of c and m range between 0 and **Range**. **St** and **Tr** respectively denote the number of states and transitions of the Büchi automaton corresponding to the specification. **B-T** is the Büchi construction time and **T-T** is the total verification time. All times are reported in milliseconds. A $*$ indicates that the Büchi automaton construction did not terminate in 10 minutes.

taken by MAGIC to construct the Büchi automaton and confirm that the corresponding implementation indeed meets the specification.

A careful inspection of the table in Table 8.1 reveals several consistent trends. First, the number of Büchi states increases quadratically with the value of *Range* for both the pure state-based and pure event-based formalisms. In contrast, the increase is only linear when both states and events are used. We notice a similar pattern among the number of transitions in the Büchi automata. The rapid increase in the sizes of Büchi automata will naturally contribute to increased model checking time. However, we notice that the major portion of the total verification time is required to construct the Büchi automaton. While this time increases rapidly in all three formalisms, the growth is observed to be most benign for the state/event scenario. The net result is clearly evident from Table 8.1. Using both states and events allows us to push the limits of c and m beyond what is possible by using either states or events alone.

8.5 SE-LTL Verification of C Programs

In this section we present a compositional CEGAR framework for verifying C programs against SE-LTL specifications. Our framework for SE-LTL will have the same general structure as that for simulation presented in earlier chapters. The crucial difference from simulation arises due to the difference in the structure of the counterexamples. Recall that a Counterexample Witness for simulation always has a tree-like structure and hence is acyclic. In contrast a Lasso always has an infinite path and hence must also necessarily have a cycle. Thus we have to modify our algorithms for counterexample validation and abstraction refinement to take into account the cyclic structure present in Lassos.

8.5.1 Compositional Lasso Validation

The notion of projections for Lassos is exactly the same as that for Counterexample Witnesses since projections are defined on LKSs and are unaffected by the presence or absence of cycles. The algorithm for validating Lasso projections is called **WeakSimLasso** and is presented in Procedure 8.1. **WeakSimLasso** takes as input a projected Lasso CE, a component $\mathcal{C}$, and a context γ for $\mathcal{C}$. It returns TRUE if $CE \precsim [\![\mathcal{C}]\!]_\gamma$ and FALSE otherwise. **WeakSimLasso** manipulates sets of states of $[\![\mathcal{C}]\!]_\gamma$ using the symbolic techniques presented in Section 3.4. In particular it uses the functions $PreImage$ and $Restrict$ to compute pre-images and restrict sets of states with respect to propositions.

WeakSimLasso iteratively computes for each state s of CE, the set of states $Sim(s)$ of $[\![\mathcal{C}]\!]_\gamma$ which can weakly simulate s. It returns TRUE if $Sim(Init_{CE}) \cap Init_{[\![\mathcal{C}]\!]_\gamma} \neq \emptyset$, otherwise it returns FALSE. Note that the process of computing Sim

Procedure 8.1 WeakSimLasso returns TRUE iff $CE \precsim [\![\mathcal{C}]\!]_\gamma$.

```
Algorithm WeakSimLasso($CE$, $\mathcal{C}$, $\gamma$)
- $CE$ : is a Lasso, $\mathcal{C}$ : is a component, $\gamma$ : is a context for $\mathcal{C}$
let $CE = (S_1, Init_1, AP_1, L_1, \Sigma_1, T_1)$;
let $[\![\mathcal{C}]\!]_\gamma = (S_2, Init_2, AP_2, L_2, \Sigma_2, T_2)$;
  //$[\![\mathcal{C}]\!]_\gamma$ is the concrete semantics of $\mathcal{C}$ with respect to $\gamma$
for each $s \in S_1$, $Sim(s) := Restrict(S_2, L_1(s))$;
  //$Sim$ is a map from $S_1$ to $2^{S_2}$
  //intuitively, $Sim(s)$ contains the states of $[\![\mathcal{C}]\!]_\gamma$ which can weakly simulate $s$
  //initially, $Sim(s)$ = subset of $S_2$ with same propositional labeling as $s$
forever do
    if $Sim(Init_1) \cap Init_2 = \emptyset$ then return FALSE;
    $OldSim := Sim$;
    for each $s \xrightarrow{\alpha} s' \in T_1$    //$s'$ is a successor state of $s$
        if $(\alpha = \tau)$ then $Sim(s) := Sim(s) \cap Sim(s')$;
        else $Sim(s) := Sim(s) \cap PreImage(Sim(s'), \alpha)$;
    if $Sim = OldSim$ then return TRUE;
```

essentially involves the computation of a greatest fixed point and might not terminate in general. Hence **WeakSimLasso** is really a semi-algorithm. But this situation can hardly be improved since checking if $CE \precsim [\![\mathcal{C}]\!]_\gamma$ is an undecidable problem in general.

Theorem 22 *Algorithm* **WeakSimLasso** *is correct.*

Proof. From the definition of weak simulation, the correctness of *Restrict* and *PreImage*, and the fact that $\tau \notin \Sigma_{\mathcal{C}\gamma}$.

□

8.5.2 Abstraction Refinement

Algorithm **AbsRefSELTL**, presented in Procedure 8.2 refines an abstraction on the basis of a spurious Lasso projection. It is very similar to algorithm **AbsRefMin**

(Procedure 7.1) and we will not explain it further here. In order to check if a set of branches B can eliminate a spurious Lasso CE, it first creates a predicate abstraction $\widehat{M}$ using B as the set of seed branches. It then invokes procedure **AbsSimLasso** to check if $\widehat{M}$ can weakly simulate CE. As we already know, B can eliminate CE iff $\widehat{M}$ cannot weakly simulate CE.

Note that, as in the case for simulation conformance, our algorithm for constructing predicate mappings can only derive predicates from branch conditions. Therefore, in principle, we might be unable to eliminate a spurious Lasso. In the context of algorithm **AbsRefSELTL**, this means that we could end up trying all sets of branches without finding an appropriate refined abstraction $\widehat{M}$. In such a case we return ERROR.

Procedure 8.2 AbsRefSELTL returns a refined abstraction for $\mathcal{C}$ that eliminates a spurious Lasso projection CE and ERROR on failure. The parameter ϕ initially expresses constraints about branches which can eliminate all previous spurious Lasso projections. **AbsRefSELTL** also updates ϕ with the constraints for the new spurious Lasso projection CE.

```
Algorithm AbsRefSELTL($CE, \mathcal{C}, \phi, \gamma$)
- $CE$ : is a spurious Lasso, $\phi$ : is a Boolean formula
- $\mathcal{C}$ : is a component, $\gamma$ : is a context for $\mathcal{C}$
$\phi_{CE}$ := FALSE;
for each $B \subseteq \mathcal{B}_{\mathcal{C}}$    //$\mathcal{B}_{\mathcal{C}}$ is the set of branches in $\mathcal{C}$
    $\Pi$ := PredInfer($\mathcal{C}, \gamma, B$);    //$\Pi$ is set of predicates inferred from $B$
    $\widehat{M} := [\![\mathcal{C}]\!]_{\gamma}^{\Pi}$;    //$\widehat{M}$ is the predicate abstraction of $\mathcal{C}$ using $\Pi$
    if $\neg$AbsSimLasso($CE, \widehat{M}$) then $\phi_{CE} := \phi_{CE} \vee \bigwedge_{b_i \in B} v_i$;
      //$\widehat{M}$ cannot weakly simulate $CE$, hence $B$ can eliminate $CE$
$\phi := \phi \wedge \phi_{CE}$;    //update $\phi$ with the constraints for $CE$
invoke PBS to solve $(\phi, \Sigma_{i=1}^{k} v_i)$;
if $\phi$ is unsatisfiable then return ERROR; //no set of branches can eliminate $CE$
else let $s$ = solution returned by PBS;
let $\{v_1, \ldots, v_m\}$ = variables assigned TRUE by $s_{opt}$ and $\overline{B} = \{b_1, \ldots, b_m\}$;
$\Pi$ := PredInfer($\mathcal{C}, \gamma, \overline{B}$); //$\Pi$ is the set of predicates inferred from $\overline{B}$
return $[\![\mathcal{C}]\!]_{\gamma}^{\Pi}$; //return the predicate abstraction of $\mathcal{C}$ using $\Pi$
```

Procedure 8.3 AbsSimLasso returns TRUE iff $\widehat{M}$ can weakly simulate CE.

```
Algorithm AbsSimLasso($CE, \widehat{M}$)
- $CE$ : is a Lasso, $\widehat{M}$ : is an LKS obtained by predicate abstraction
let $CE = (S, Init, AP, L, \Sigma, T)$;
let $\widehat{M} = \left(\widehat{S}, \widehat{Init}, \widehat{AP}, \widehat{L}, \widehat{\Sigma}, \widehat{T}\right)$;
for each $s \in S$, $Sim(s) := \{\widehat{s} \in \widehat{S} \mid \widehat{L}(\widehat{s}) = L(s)\}$;
  //Sim is a map from S to $2^{\widehat{S}}$
  //intuitively, Sim(s) contains the states of $\widehat{M}$ which can weakly simulate s
  //initially, Sim(s) = subset of $\widehat{S}$ with same propositional labeling as s
forever do
    if $Sim(Init) \cap \widehat{Init} = \emptyset$ then return FALSE;
    $OldSim := Sim$; //save old map for subsequent fixed point detection
    for each $s \xrightarrow{\alpha} s' \in T$    //s' is a successor state of s
        if $(\alpha = \tau)$ then $Sim(s) := Sim(s) \cap Sim(s')$;
        else $Sim(s) := \{\widehat{s} \in Sim(s) \mid Succ(\widehat{s}, \alpha) \cap Sim(s') \neq \emptyset\}$;
    if $Sim = OldSim$ then return TRUE;
      //fixed point reached, hence $\widehat{M}$ weakly simulates CE
```

Theorem 23 *Algorithm* **AbsRefSELTL** *is correct.*

Proof. It is obvious that **AbsRefSELTL** either returns ERROR or a refined abstraction $\widehat{M}$ such that for all spurious Lasso projections CE seen so far, $CE \not\precsim \widehat{M}$.

□

8.6 CEGAR for SE-LTL

The complete CEGAR algorithm in the context of SE-LTL conformance, called **SELTL-CEGAR**, is presented in Procedure 8.4. It invokes at various stages algorithms **PredInfer**, the predicate abstraction algorithm, **ModelCheck**, **WeakSimLasso** and **AbsRefSELTL**. It takes as input a program $\mathcal{P}$, an SE-LTL specification LKS ϕ and a context Γ for $\mathcal{P}$ and outputs either "$\mathcal{P} \models \phi$" or "$\mathcal{P} \not\models \phi$" or ERROR. Intuitively **SELTL-CEGAR** works as follows.

Procedure 8.4 SELTL-CEGAR checks entailment between a program $\mathcal{P}$ and an SE-LTL specification ϕ in a context Γ.

```
Algorithm SELTL-CEGAR($\mathcal{P}, \phi, \Gamma$)
- $\mathcal{P}$ : is a program, $\Gamma$ : is a context for $\mathcal{P}$, $\phi$ : is an SE-LTL formula
let $\mathcal{P} = \langle \mathcal{C}_1, \dots, \mathcal{C}_n \rangle$ and $\Gamma = \langle \gamma_1, \dots, \gamma_n \rangle$;
for each $i \in \{1, \dots, n\}$
    $\Pi_i :=$ PredInfer($\mathcal{C}_i, \gamma_i, \emptyset$) and $\widehat{M_i} := [\![\mathcal{C}_i]\!]^{\Pi_i}_{\gamma_i}$ and $\phi_i = $ TRUE;
      //$\widehat{M_i}$ = initial predicate abstractions of $\mathcal{C}_i$ with empty set of seed branches
forever do
    let $\widehat{M} = \widehat{M_1} \parallel \cdots \parallel \widehat{M_n}$;
      //$\widehat{M}$ is the composition of predicate abstractions
    if (ModelCheck($\widehat{M}, \phi$) = "$\widehat{M} \models \phi$") return "$\mathcal{P} \models \phi$";
      //if $\widehat{M}$ satisfies $\phi$ then so does $\mathcal{P}$
    let $CE$ = Lasso returned by ModelCheck;
    find $i \in \{1, \dots, n\}$ such that $\neg$WeakSimLasso($CE \downharpoonleft \gamma_i, \mathcal{C}_i, \gamma_i$);
      //check compositionally if $CE$ is spurious
    if (no such $i$ found) return "$\mathcal{P} \not\models \phi$"; //$CE$ is valid and hence $\mathcal{P} \not\models \phi$
    if (AbsRefSELTL($CE \downharpoonleft \gamma_i, \mathcal{C}_i, \phi_i, \gamma_i$) = ERROR) return ERROR;
      //no set of branches can eliminate $CE \downharpoonleft \gamma_i$
    $\widehat{M_i} :=$ AbsRefSELTL($CE \downharpoonleft \gamma_i, \mathcal{C}_i, \phi_i, \gamma_i$); //refine the abstraction and repeat
```

Let $\mathcal{P} = \langle \mathcal{C}_1, \dots, \mathcal{C}_n \rangle$. Then **SELTL-CEGAR** maintains a set of abstractions $\widehat{M_1}, \dots, \widehat{M_n}$ where $\widehat{M_i}$ is a predicate abstraction of $\mathcal{C}_i$ for $i \in \{1, \dots, n\}$. Note that by Theorem 7, $\widehat{M} = \widehat{M_1} \parallel \cdots \parallel \widehat{M_n}$ is an abstraction of $\mathcal{P}$. Initially each $\widehat{M_i}$ is set to the predicate abstraction of $\mathcal{C}_i$ corresponding to an empty set of seed branches. Also for each $\mathcal{C}_i$, **SELTL-CEGAR** maintains a boolean formula ϕ_i (initialized to TRUE) used for predicate minimization. Next **SELTL-CEGAR** iteratively performs the following steps:

1. **(Verify)** Invoke algorithm **ModelCheck** to check if $\widehat{M}$ satisfies ϕ. If

ModelCheck returns "$\widehat{M} \models \phi$" then output "$\mathcal{P} \models \phi$" and exit. Otherwise let CE be the Lasso returned by **ModelCheck**. Go to step 2.

2. **(Validate)** For $i \in \{1, \ldots, n\}$ invoke **WeakSimLasso**($CE \downharpoonleft \gamma_i, \mathcal{C}_i, \gamma_i$). If every invocation of **WeakSimLasso** returns TRUE then output "$\mathcal{P} \not\models \phi$" and exit. Otherwise let i be the minimal element of $\{1, \ldots, n\}$ such that **WeakSimLasso**($CE \downharpoonleft \gamma_i, \mathcal{C}_i, \gamma_i$) returns FALSE. Go to step 3.

3. **(Refine)** Invoke **AbsRefSELTL**($CE \downharpoonleft \gamma_i, \mathcal{C}_i, \phi_i, \gamma_i$). If **AbsRefSELTL** returns ERROR, output ERROR and stop. Otherwise set $\widehat{M_i}$ to the abstraction returned by **AbsRefSELTL**. Repeat from step 1.

Theorem 24 *Algorithm* **SELTL-CEGAR** *is correct.*

Proof. When **SELTL-CEGAR** returns "$\mathcal{P} \models \phi$" its correctness follows from Theorem 7, Theorem 21 and Theorem 1. When **SELTL-CEGAR** returns "$\mathcal{P} \not\models \phi$" its correctness follows from Theorem 12, Theorem 22 and Theorem 23.

□

8.7 Experimental Results

We experimented with two broad sets of benchmarks. All our experiments were performed on an AMD Athlon XP 1600+ machine with 900 MB RAM running RedHat Linux 7.1. The first set of our examples were based on OpenSSL. This is a popular protocol used for secure exchange of sensitive information over untrusted networks. The target of our verification process was the implementation of the initial handshake required for the establishment of a secure channel between a client and a server.

Name	St(B)	Tr(B)	St(Mdl)	T(BA)	T(Mdl)	T(Ver)	T(Total)	Mem
srvr-1-ss	4	5	5951	213	32195	1654	34090	-
srvr-1-se	3	4	4269	209	18116	1349	19674	-
srvr-2-ss	11	23	4941	292	31331	2479	34102	-
srvr-2-se	3	4	4269	196	17897	1317	19410	-
srvr-3-ss	37	149	5065	1147	26958	4031	32137	-
srvr-3-se	3	4	4269	462	17950	1908	20319	-
srvr-4-ss	16	41	5446	806	29809	7382	39341	28.6
srvr-4-se	7	14	4333	415	21453	3513	25906	24.1
srvr-5-ss	25	47	7951	690	48810	6842	56888	39.3
srvr-5-se	20	45	4331	497	18808	2925	22765	24.2
clnt-1-ss	16	41	4867	793	24488	1235	26953	25.8
clnt-1-se	7	14	3693	376	17250	583	18683	22.1
clnt-2-ss	25	47	7574	699	43592	1649	46444	38.1
clnt-2-se	18	40	3691	407	15304	1087	17269	21.2
ssl-1-ss	25	47	24799528	874	65585	*	*	850.5
ssl-1-se	20	45	13558984	655	33091	2172139	2206983	162.4
ssl-2-ss	25	47	32597042	836	66029	*	*	346.6
ssl-2-se	18	40	15911791	713	34641	4148550	4185068	320.7
UCOS-BUG	8	14	873	205	3409	261	3880	-
UCOS-1	8	14	873	194	3365	2797	6357	-
UCOS-2	5	8	873	123	3372	2630	6127	-

Table 8.2: Experimental results with OpenSSL and μC/OS-II. **St(B)** and **Tr(B)** = respectively the number of states and transitions in the Büchi automaton; **St(Mdl)** = number of states in the model; **T(Mdl)** = model construction time; **T(BA)** = Büchi construction time; **T(Ver)** = model checking time; **T(Total)** = total verification time. All reported times are in milliseconds. **Mem** is the total memory requirement in MB. A * indicates that the model checking did not terminate within 2 hours and was aborted. In such cases, other measurements were made at the point of forced termination. A - indicates that the corresponding measurement was not taken.

From the official SSL specification [109] we derived a set of nine properties that every correct SSL implementation should satisfy. The first five properties are relevant only to the server, the next two apply only to the client, and the last two properties refer to both a server and a client executing concurrently. For instance, the first property states that whenever the server asks the client to terminate the handshake, it eventually either gets a correct response from the client or exits with an error code. The second property expresses the fact that whenever the server receives a handshake request from a client, it eventually acknowledges the request or returns with an error code. The third property states that a server never exchanges encryption keys with a client once the cipher scheme has been changed.

Each of these properties were then expressed in SE-LTL, once using only states and again using both states and events. Table 8.2 summarizes the results of our experiments with these benchmarks. The SSL benchmarks have names of the form **x-y-z** where **x** denotes the type of the property and can be either **srvr**, **clnt** or **ssl**, depending on whether the property refers respectively to only the server, only the client, or both server and client. **y** denotes the property number while **z** denotes the specification style and can be either **ss** (only states) or **se** (both states and events). We note that in each case the numbers for state/event properties are considerably better than those for the corresponding pure-state properties.

The second set of our benchmarks were obtained from the source code of μC/OS-II version 2.70 (which we will refer to simply as μC/OS-II in the rest of this thesis). This is a popular, lightweight, real-time, multi-tasking operating system written in about 6000 lines of ANSI C. μC/OS-II uses a lock to ensure mutual exclusion for critical section code. Using SE-LTL we expressed two properties of μC/OS-II: (i) the lock is acquired and released alternately starting with an acquire and (ii) every time the lock is acquired it is eventually released. These properties were expressed using only events.

We found four **bugs** in μC/OS-II that causes it to violate the first property. One of these bugs was unknown to the developers while the other three had been found previously. The second property was found to be valid. In Table 8.2 these experiments are named **UCOS-BUG** and **UCOS-2** respectively. Next we fixed the bug and verified that the first property holds for the corrected μC/OS-II. This experiment is called **UCOS-1** in Table 8.2.

Chapter 9

Two-Level Abstraction Refinement

In this chapter, we attempt to address the *state-space explosion* problem in the context of verifying simulation conformance between a concurrent (message-passing) C program and an LKS specification. More specifically, we present a fully automated compositional framework which combines two orthogonal abstraction techniques (operating respectively on data and events) within a counterexample guided abstraction refinement (CEGAR) scheme. In this way, our algorithm incrementally increases the granularity of the abstractions until the specification is either established or refuted. Our explicit use of compositionality delays the onset of state space explosion for as long as possible. To our knowledge, this is the first compositional use of CEGAR in the context of model checking concurrent C programs. We describe our approach in detail, and report on some very encouraging preliminary experimental results obtained with our tool MAGIC.

9.1 Introduction

As mentioned before, there has been a tremendous amount of research and advancement over the years devoted to the abstract modeling and validation of concurrent systems and their specifications. However, the majority of these advances target specific—and often orthogonal—aspects of the problem, but fail to solve it as a whole. The work we present here attempts to provide a more complete approach to efficiently verify global specifications on concurrent C programs in a fully automated way. More specifically, we focus on reactive systems, implemented using concurrent C programs that communicate with each other through synchronous (blocking) message-passing. Examples of such systems include client-server protocols, web services [15], schedulers, telecommunication applications, etc. As in previous chapters, we consider specifications expressed as LKSs.

We propose a fully automated *compositional two-level counterexample guided abstraction refinement* scheme to verify that a concurrent C program $\mathcal{P}$ conforms to an LKS specification Sp in a context Γ. Let us assume that our program $\mathcal{P}$ consists of a set of components $\langle \mathcal{C}_1, \ldots, \mathcal{C}_n \rangle$. Naturally, our program context Γ must also consist of a set of component contexts $\langle \gamma_1, \ldots, \gamma_n \rangle$, one for each component $\mathcal{C}_i$.

We first transform each $\mathcal{C}_i$ into a finite-state predicate abstraction $\widehat{\mathcal{C}}_i$. Since the parallel composition of these predicate abstractions may well still have an unmanageably large state space, we further reduce each $\widehat{\mathcal{C}}_i$ by conservatively aggregating states together, based on the actions they can perform, yielding a smaller *action-guided* abstraction A_i; only then do we explicitly build the global state space of the much coarser parallel composition $\mathcal{A} = \mathcal{A}_1 \parallel \ldots \parallel \mathcal{A}_n$.

Recall that $[\![\mathcal{P}]\!]_\Gamma$ denotes the concrete semantics of our program $\mathcal{P}$. We know that by construction, $[\![\mathcal{P}]\!]_\Gamma \preccurlyeq \mathcal{A}$, i.e., $\mathcal{A}$ exhibits all of $\mathcal{P}$'s behaviors, and usually many more. We check $\mathcal{A} \preccurlyeq Sp$. If successful, we conclude that $[\![\mathcal{P}]\!]_\Gamma \preccurlyeq Sp$. Otherwise, we must examine the Counterexample Witness obtained to determine whether it is valid or not. It is important to note that this validation can be carried out not only component-wise (as in previous chapters), but also level-wise. Furthermore, we are able to avoid constructing in full the large state space of $\mathcal{P}$.

A valid Counterexample Witness shows $\mathcal{P} \not\preccurlyeq Sp$ and thus terminates the procedure. Otherwise, a (component-specific) refinement of the appropriate abstraction is carried out, eliminating the spurious Counterexample Witness, and the algorithm proceeds with a new iteration of the verification cycle. The crucial features of our approach therefore consist of the following:

- We leverage two very different kinds of abstraction to reduce a concurrent C program to a very coarse parallel composition of finite-state processes. The first (predicate) abstraction partitions the (potentially infinite) state space according to the possible values of variables, whereas the second (action-guided) abstraction groups these resulting states together according to the actions that they can perform.

- A counterexample guided abstraction refinement scheme incrementally refines these abstractions until the right granularity is achieved to decide whether the specification holds or not. We note that while termination of the entire algorithm obviously cannot be guaranteed[1], all of our experimental examples could be handled without requiring human input.

- Our use of compositional reasoning, grounded in standard process algebraic

[1]This of course follows from the fact that the halting problem is undecidable.

techniques, enables us to perform most of our analysis component by component, without ever having to construct global state spaces except at the most abstract level.

The verification procedure is fully automated, and requires no user input beyond supplying the C programs and the specification to be verified. We have implemented the algorithm within our tool MAGIC [24, 80] and have carried out a number of case studies, which we report here. To our knowledge, our algorithm is the first to invoke CEGAR over more than a single abstraction refinement scheme (and in particular over *action-based* abstractions), and also the first to combine CEGAR with fully automatic compositional reasoning for concurrent systems.

The experiments we have carried out range over a variety of sequential and concurrent examples, and yield promising results. The two-level approach constructs models that are often almost two orders of magnitude smaller than those generated by predicate abstraction alone. This also translates to over an order of magnitude reduction in actual memory requirement. Additionally, the two-level approach is faster, especially in the concurrent benchmarks where it often reduces verification time by a factor of over three. Full details are presented in Section 9.6.

9.2 Related Work

Predicate abstraction was introduced in [63] as a means to transform conservatively infinite-state systems into finite-state ones, so as to enable the use of finitary techniques such as model checking [32, 39]. It has since been widely used—see, for instance [5, 42, 44, 48, 50, 89].

The formalization of the more general notion of abstraction first appeared in [47].

We distinguish between *exact* abstractions, which preserve all properties of interest of the system, and *conservative* abstractions—used in this chapter—which are only guaranteed to preserve safety properties of the system (e.g., [36, 75]). The advantage of the latter is that they usually lead to much greater reductions in the state space than their exact counterparts. However, conservative abstractions in general require an iterated abstraction refinement mechanism (such as CEGAR [37]) in order to establish specification satisfaction.

The abstractions we use on finite-state processes essentially group together states that can perform the same set of actions, and gradually refine these partitions according to reachable successor states. Our refinement procedure can be seen as an atomic step of the Paige-Tarjan algorithm [96], and therefore yields successive abstractions which converge in a finite number of steps to the bisimulation quotient of the original process.

CEGAR has been used, among others, in non-automated [90], and automated [6, 29, 40, 66, 77, 98] forms. Compositionality, which features crucially in our work, is broadly concerned with the preservation of properties under substitution of components in concurrent systems. It has been most extensively studied in process algebra (e.g., [69, 85, 100]), particularly in conjunction with abstraction. In [10], a compositional framework for (non-automated) CEGAR over data-based abstractions is presented. This approach differs from ours in that communication takes place through shared variables (rather than blocking message-passing), and abstractions are refined by eliminating spurious transitions, rather than by splitting abstract states.

A technique closely related to compositionality is that of assume-guarantee reasoning [64, 67, 84]. It was originally developed to circumvent the difficulties associated with generating exact abstractions, and has recently been implemented as part of a fully automated and incremental verification framework [41].

Among the works most closely resembling ours we note the following. The Bandera project [44] offers tool support for the automated verification of Java programs based on abstract interpretation; there is no automated CEGAR and no explicit compositional support for concurrency. Păsăreanu et. al. [98] import Bandera-derived abstractions into an extension of Java PathFinder which incorporates CEGAR. However, once again no use is made of compositionality, and only a single level of abstraction is considered. Stoller [111, 112] describes another tool implemented in Java PathFinder which explicitly supports concurrency; it uses data-type abstraction on the first level, and partial order reduction with aggregation of invisible transitions on the second level. Since all abstractions are exact it does not require the use of CEGAR. The SLAM project [5, 6, 107] has been very successful in analyzing interfaces written in C. It is built around a single-level predicate abstraction and automated CEGAR treatment, and offers no explicit compositional support for concurrency. Lastly, the BLAST project [13, 65, 66] proposes a single-level (i.e., only predicate abstraction) lazy (on-the-fly) CEGAR scheme and thread-modular assume-guarantee reasoning. The BLAST framework is based on shared variables rather than message-passing as the communication mechanism.

The next section presents a series of standard definitions that are used throughout the rest of this chapter. Section 9.5 then describes the two-level CEGAR algorithm. Finally, Section 9.6 summarizes the results of our experiments.

9.3 Abstraction

Recall, from Definition 1, than an LKS is a 6-tuple $(S, Init, AP, L, \Sigma, T)$. In this section we present our notion of abstraction. Our framework employs *quotient LKSs* as abstractions of concrete LKSs. Given a concrete LKS M, one can obtain a quotient LKS as follows. The states of the quotient LKS are obtained by grouping together states of M such that all states in a particular group agree on their propositional labeling. Alternatively, one can view these groups as equivalence classes of some equivalence relation on S_M. Transitions of the quotient LKS are defined *existentially.* We now present a formal definition of these concepts.

Definition 27 (Propositional Compatibility) *Let $M = (S, Init, AP, L, \Sigma, T)$ be any LKS. An equivalence relation $R \subseteq S \times S$ is said to be propositionally compatible iff the following condition holds:*

$$\forall s_1 \in S \centerdot \forall s_2 \in S \centerdot (s_1, s_2) \in R \implies L(s_1) = L(s_2)$$

Definition 28 (Quotient LKS) *Let $M = (S, Init, AP, L, \Sigma, T)$ be an LKS and $R \subseteq S \times S$ be a propositionally compatible equivalence relation. For an arbitrary $s \in S$ we let $[s]^R$ denote the equivalence class of s. M and R then induce a quotient LKS $M^R = \left(S^R, Init^R, AP^R, L^R, \Sigma^R, T^R\right)$ where: (i) $S^R = \{[s]^R \mid s \in S\}$, (ii) $Init^R = \{[s]^R \mid s \in Init\}$, (iii) $AP^R = AP$, (iv) $\forall [s]^R \in S^R \centerdot L^R([s]^R) = L(s)$ (note that this is well-defined because R is propositionally compatible), (v) $\Sigma^R = \Sigma$, and (vi) $T^R = \{([s]^R, \alpha, [s']^R) \mid (s, \alpha, s') \in T\}$.*

In the rest of this chapter we will only restrict ourselves to propositionally compatible equivalence relations. We write $[s]$ to mean $[s]^R$ when R is clear from the context. M^R is often called an *existential abstraction* of M. The states of M are referred to as *concrete states* while those of M^R are called *abstract states*. Quotient LKSs have been studied in the verification literature. In particular, the following result is well-known [39].

Theorem 25 *Let $M = (S, Init, AP, L, \Sigma, T)$ be an LKS, R an equivalence relation on S, and M^R the quotient LKS induced by M and R. Then $M \preccurlyeq M^R$.*

9.4 Counterexample Validation and Refinement

Recall that our program $\mathcal{P}$ consists of a set of components $\langle \mathcal{C}_1, \ldots, \mathcal{C}_n \rangle$ and our program context Γ consists of a set of component contexts $\langle \gamma_1, \ldots, \gamma_n \rangle$, one for each $\mathcal{C}_i$. Our goal is to verify whether $\mathcal{P} \preccurlyeq Sp$.

For $i \in \{1, \ldots, n\}$, let us denote by M_i the LKS obtained by predicate abstraction of $\mathcal{C}_i$, and let R_i be an equivalence relation over S_{M_i}. Suppose CW is a Counterexample Witness for $(M_1^{R_1} \parallel \cdots \parallel M_n^{R_n}) \preccurlyeq Sp$. We would now like to verify in a component-wise manner whether CW is a valid Counterexample Witness. Recall that this involves checking whether $CW \downharpoonleft \gamma_i \precsim \mathcal{C}_i$ for $i \in \{1, \ldots, n\}$.

However we want to perform our validation check also in a level-wise manner. In other words we first check, for $i \in \{1, \ldots, n\}$, whether $CW \downharpoonleft \gamma_i \precsim M_i$. If this is not the case for some $i \in \{1, \ldots, n\}$, we refine the abstraction $M_i^{R_i}$ and repeat the simulation check. Otherwise we proceed with checking $CW \downharpoonleft \gamma_i \precsim \mathcal{C}_i$ and subsequent refinement (if required) as described in earlier chapters.

In our framework, refinement of action-guided abstractions involves computing proper refinements of equivalence relations based on abstract successors. We now present this refinement scheme, beginning with a few preliminary definitions.

Definition 29 (Equivalence Refinement) *Let R_1 and R_2 be two equivalence relations over some set S. Then R_1 is said to be a refinement of R_2 iff the following condition holds:*

$$\forall s \in S \centerdot [s]^{R_1} \subseteq [s]^{R_2}$$

R_1 is said to be a proper refinement of R_2 iff the following condition holds:

$$\big(\forall s \in S \centerdot [s]^{R_1} \subseteq [s]^{R_2}\big) \wedge \big(\exists s \in S \centerdot [s]^{R_1} \subset [s]^{R_2}\big)$$

Definition 30 (Abstract Successor) *Let $M = (S, Init, AP, L, \Sigma, T)$ be an LKS, and $R \subseteq S \times S$ be an equivalence relation. Let $M^R = \big(S^R, Init^R, AP^R, L^R, \Sigma^R, T^R\big)$, $s \in S$ and $\alpha \in \Sigma$. Then the function $\widehat{Succ} : S \times \Sigma \to 2^{S^R}$ is defined as follows:*

$$\widehat{Succ}(s, \alpha) = \{[s']^R \in S^R \mid s' \in Succ(s, \alpha)\}$$

In other words, $[s']^R \in S^R$ is an abstract successor of s under action α iff M has an α-labeled transition from s to some element of $[s']^R$.

9.4.1 Splitting Equivalence Classes

Given M, R, $[s]^R \in S^R$ and $A \subseteq \Sigma$, we denote by $Split(M, R, [s]^R, A)$ the equivalence relation obtained from R by sub-partitioning the equivalence class $[s]^R$ according to the following scheme: $\forall s_1, s_2 \in [s]^R$, s_1 and s_2 belong to the same sub-partition of $[s]^R$ iff $\forall \alpha \in A \centerdot \widehat{Succ}(s_1, \alpha) = \widehat{Succ}(s_2, \alpha)$.

Note that the equivalence classes (abstract states) other than $[s]^R$ are left unchanged. Recall Definition 29 of refinement between equivalence relations. It is easy to see that $Split(M, R, [s]^R, A)$ is a refinement of R. In addition, $Split(M, R, [s]^R, A)$ is a *proper* refinement of R iff $[s]^R$ is split into more than one piece, i.e., if the following condition holds:

$$\exists \alpha \in A \,\centerdot\, \exists s_1 \in [s]^R \,\centerdot\, \exists s_2 \in [s]^R \,\centerdot\, \exists [s']^R \in S^R \centerdot$$
$$[s']^R \in \widehat{Succ}(s_1, \alpha) \wedge [s']^R \notin \widehat{Succ}(s_2, \alpha) \tag{9.1}$$

The correctness of the above claim is easy to see since if Condition 9.1 holds, then according to our definition, s_1 and s_2 must belong to different sub-partitions of $[s]^R$ after the application of *Split*.

9.4.2 Checking Validity of a Counterexample Witness

Let M be an LKS obtained by predicate abstraction from a component and R be an equivalence relation on the states of M. Let CW be a Counterexample Witness projection such that $CW \precsim M^R$ where M^R is the quotient LKS induced by M and R. Recall that we are interested to check if $CW \precsim M$. This is achieved by algorithm **WeakSimulAG** shown in Figure 9.1. **WeakSimulAG** first invokes algorithm **CanSimulAG** (shown in Figure 9.2) to compute the set S of states of M which can weakly simulate the initial state $Init$ of CW. It then returns TRUE if some initial state of M belongs to S and FALSE otherwise.

Procedure 9.1 WeakSimulAG returns TRUE if $CW \precsim M$ and FALSE otherwise.

```
Algorithm WeakSimulAG($CW, M$)
- $CW$ : is a Counterexample Witness projection, $M$ : is an LKS
let $Init$ = initial state of $CW$;
$S$ := CanSimulAG($CW, Init, M$);
let $Init'$ = set of initial states of $M$;
return ($S \cap Init' \neq \emptyset$);
```

Procedure 9.2 CanSimulAG returns the set of states of M which can weakly simulate s.

```
Algorithm CanSimulAG($CW, s, M$)
- $CW$ : is a Counterexample Witness projection, $s$ : is a state of $CW$
- $M$ : is an LKS
let $CW = (S_1, Init_1, AP_1, L_1, \Sigma_1, T_1)$;
let $M = (S_2, Init_2, AP_2, L_2, \Sigma_2, T_2)$;
$S := Restrict(S_2, L_1(s))$;    //$S$ = subset of $S_2$ with same propositional labeling as $s$
for each $s \xrightarrow{\alpha} s' \in T_1$    //$s'$ is a successor state of $s$
    $S' :=$ CanSimulAG($CW, s', M$);    //compute result for successor
    if $(\alpha \neq \tau)$ then $S' := \{s' \in S_2 \mid Succ(s', \alpha) \cap S' \neq \emptyset\}$;    //take non-$\tau$ pre-image
    $S := S \cap S'$;    //update result
return $S$;
```

9.4.3 Refining an Action-Guided Abstraction

In the previous section we described an algorithm to check if a Counterexample Witness projection CW is weakly simulated by an LKS M. If this is the case then we know that CW is a valid Counterexample Witness projection. However, if $CW \not\precsim M$ then we need to properly refine our equivalence relation R so as to obtain a more precise quotient LKS for the next iteration of the CEGAR loop. In this section we present an algorithm to refine R given that $CW \not\precsim M$.

We begin with the notion of a *simulation map*. Intuitively, given two LKSs M_1 and M_2, a simulation map is a function that maps each state of M_1 to a state of M_2 which weakly simulates it.

Definition 31 (Simulation Map) *Let $M_1 = (S_1, Init_1, AP_1, L_1, \Sigma_1, T_1)$ and $M_2 = (S_2, Init_2, AP_2, L_2, \Sigma_2, T_2)$ be two LKSs such that:* **(SM1)** *M_1 has a tree structure,* **(SM2)** *$\tau \in \Sigma_1$, and* **(SM3)** *$\tau \notin \Sigma_2$. Then a function $\theta : S_1 \to S_2$ is said to be a simulation map between M_1 and M_2 iff it obeys the following conditions:*

$$\forall s \in S_1 \centerdot L_1(s) = L_2(\theta(s))$$

$$\forall s \in \mathit{Init}_1 \centerdot \theta(s) \in \mathit{Init}_2$$

$$\forall \alpha \in \Sigma_1 \setminus \{\tau\} \centerdot \forall s \in S_1 \centerdot \forall s' \in S_1 \centerdot s \xrightarrow{\alpha} s' \implies \theta(s) \xrightarrow{\alpha} \theta(s')$$

$$\forall s \in S_1 \centerdot \forall s' \in S_1 \centerdot s \xrightarrow{\tau} s' \implies \theta(s) = \theta(s')$$

Clearly, if conditions **SM1–SM4** above are satisfied, then there exists a simulation map θ between M_1 and M_2 iff $M_1 \precsim M_2$. Figure 9.1 shows such a simulation map θ on the left. Moreover, suppose we have another LKS $M_3 = (S_3, \mathit{Init}_3, \mathit{AP}_3, L_3, \Sigma_3, T_3)$ and suppose that there is a function $\nu : S_2 \to S_3$ such that the following conditions hold:

$$(\mathbf{NU1}) \qquad \forall s \in S_2 \centerdot L_2(s) = L_3(\nu(s))$$

$$(\mathbf{NU2}) \qquad \forall s \in \mathit{Init}_2 \centerdot \nu(s) \in \mathit{Init}_3$$

$$(\mathbf{NU3}) \qquad \forall \alpha \in \Sigma_2 \centerdot \forall s \in S_2 \centerdot \forall s' \in S_2 \centerdot s \xrightarrow{\alpha} s' \implies \nu(s) \xrightarrow{\alpha} \nu(s')$$

Then it is obvious that the composition of θ and ν, i.e., $\theta \circ \nu$, is a simulation map between M_1 and M_3. Figure 9.1 shows such a composition simulation map on the right.

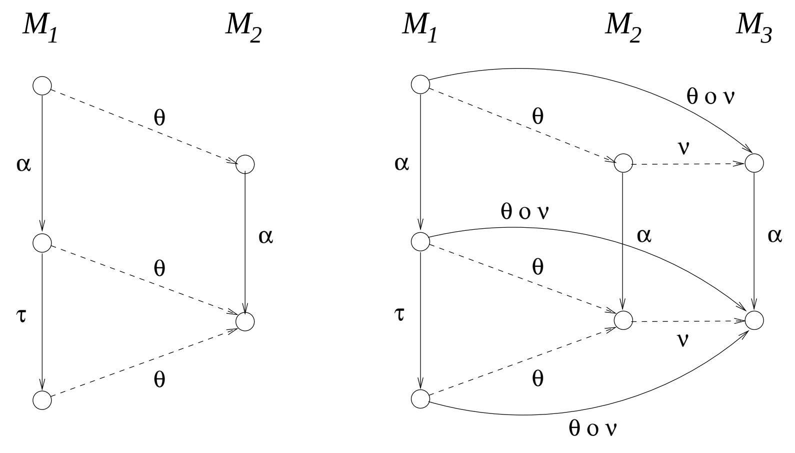


Figure 9.1: On the left is a simulation map θ between M_1 and M_2. On the right is a simulation map $\theta \circ \nu$ between M_1 and M_3.

Let M be an LKS obtained by predicate abstraction from a component and R be an equivalence relation over the states of M. Let CW be a Counterexample Witness projection such that $CW \precsim M^R$. Clearly there exists a simulation map θ between CW and M^R, where M^R is the quotient LKS induced by M and R.

Now suppose that $CW \not\precsim M$. Then we claim that there exists a state $[s] \in Range(\theta)$ and an outgoing action α from $[s]$ such that splitting the equivalence class $[s]$ on the basis of α (cf. Section 9.4.1) will yield a proper refinement of R.

To understand why this claim is true, consider the converse. In other words suppose that for every $[s] \in Range(\theta)$ and for every outgoing action α from $[s]$, splitting $[s]$ on the basis of α does not yield a proper refinement. This means that every element of $[s]$ must have the same set of abstract successors (cf. Definition 30) on α. But then, it follows that we can define a mapping ν from $Range(\theta)$ to the states of M which satisfies conditions **NU1–NU3** above. This would mean of course that $\theta \circ \nu$ would be a simulation map from CW to M which would further imply that $CW \precsim M$. This is clearly a contradiction.

The summary of the above paragraph is that if $CW \not\precsim M$, then there exists a state $[s] \in Range(\theta)$ and an outgoing action α from $[s]$ such that splitting the equivalence class $[s]$ on the basis of α will yield a proper refinement of R. Our algorithm **AbsRefineAG** to refine R is therefore very simple. For each equivalence class $[s] \in Range(\theta)$, and for each outgoing action α from $[s]$, **AbsRefineAG** attempts to split $[s]$ on the basis of α. **AbsRefineAG** stops as soon as a proper refinement of R is obtained.

9.4.4 Overall Action-Guided Abstraction Refinement

Our algorithm to check the validity of CW at the action-guided abstraction level is called **ValidateAndRefineAG** and is presented in Figure 9.3. **ValidateAndRefineAG** takes as input a composition of quotient LKSs $\widehat{M} = \widehat{M_1}^{R_1} \parallel \cdots \parallel \widehat{M_n}^{R_n}$ and a Counterexample Witness CW. For each $i \in \{1, \ldots, n\}$, it attempts to either verify that $CW \downharpoonleft \gamma_i \precsim \widehat{M_i}$ or refine the abstraction $\widehat{M_i}^{R_i}$.

To do this it first invokes **WeakSimulAG** to check if $CW \downharpoonleft \gamma_i \precsim \widehat{M_i}$. Is **WeakSimulAG** returns TRUE, it proceeds with the next index i. Otherwise it invokes **AbsRefineAG** to construct a proper refinement of the equivalence relation R_i. **ValidateAndRefineAG** returns TRUE if the some abstraction $\widehat{M_i}^{R_i}$ was refined and FALSE otherwise.

Procedure 9.3 ValidateAndRefineAG checks the validity of CW at the action-guided abstraction level. It returns FALSE if CW is found to be valid. Otherwise it properly refines some equivalence relation R_i and returns TRUE.

```
Algorithm ValidateAndRefineAG($\widehat{M}$, $CW$)
- $\widehat{M}$ : is a composition of quotient LKSs
- $CW$ : is a Counterexample Witness
let $\widehat{M} = \widehat{M_1}^{R_1} \parallel \cdots \parallel \widehat{M_n}^{R_n}$;    //components of $\widehat{M}$
for $i = 1$ to $n$ //try to refine one of the $R_i$'s using $CW$
    if ($\neg$WeakSimulAG($CW \downharpoonleft \gamma_i, \widehat{M_i}$))
        AbsRefineAG($\widehat{M_i}^{R_i}$);
        return TRUE;
return FALSE;    //none of the quotient LKSs was refined
```

9.5 Two-Level CEGAR

Algorithm **TwoLevelCEGAR**, presented in Procedure 9.4, captures the complete two-level CEGAR algorithm for simulation conformance. It is very similar to **SimulCEGAR** other than the invocation of **ValidateAndRefineAG** to perform Counterexample Witness validation and refinement at the level of action-guided abstractions. We now give a line-by-line explanation of **TwoLevelCEGAR**.

Procedure 9.4 TwoLevelCEGAR checks simulation conformance between a program $\mathcal{P}$ and a specification Sp in a context Γ.

```
Algorithm TwoLevelCEGAR($\mathcal{P}, Sp, \Gamma$)
- $\mathcal{P}$ : is a program, $\Gamma$ : is a context for $\mathcal{P}$
- $Sp$ : is a specification LKS
1: let $\mathcal{P} = \langle \mathcal{C}_1, \ldots, \mathcal{C}_n \rangle$ and $\Gamma = \langle \gamma_1, \ldots, \gamma_n \rangle$;
2: for each $i \in \{1, \ldots, n\}$
3:    $\widehat{M_i}$ := predicate abstraction of $\mathcal{C}_i$ with empty set of predicates;
4:    $R_i$ := largest propositionally compatible equivalence on $\widehat{M_i}$'s states;
5: Loop:
6:    let $\widehat{M} = \widehat{M_1}^{R_1} \parallel \cdots \parallel \widehat{M_n}^{R_n}$;
7:    if (SimulWitness($\widehat{M}, Sp$) = "$\widehat{M} \preccurlyeq Sp$") return "$\mathcal{P} \preccurlyeq Sp$";
8:    let $CW$ = Counterexample Witness returned by SimulWitness;
9:    if (ValidateAndRefineAG($\widehat{M}, CW$)) goto Loop;
10:   find $i \in \{1, \ldots, n\}$ such that $\neg$WeakSimul($CW \downharpoonleft \gamma_i, \mathcal{C}_i, \gamma_i$);
11:   if (no such $i$ found) return "$\mathcal{P} \not\preccurlyeq Sp$";
12:   else $CW_i := CW_i \cup \{CW \downharpoonleft \gamma_i\}$;
13:   if (AbsRefine($CW_i, \mathcal{C}_i, \gamma_i$) = ERROR) return ERROR;
14:   $\widehat{M_i}$ := AbsRefine($CW_i, \mathcal{C}_i, \gamma_i$);
15:   $R_i$ := largest propositionally compatible equivalence on $\widehat{M_i}$'s states;
16:   goto Loop;
```

Let the input program $\mathcal{P}$ consist of the sequence of n components $\langle \mathcal{C}_1, \ldots, \mathcal{C}_n \rangle$ (line 1). For each $i \in \{1, \ldots, n\}$, (line 2) **TwoLevelCEGAR** constructs (line 3) a predicate abstraction $\widehat{M_i}$ of $\mathcal{C}_i$ with an empty set of predicates. It also maintains a sequence of equivalence relations $\langle R_1, \ldots, R_n \rangle$ such that R_i is a relation over the states of $\widehat{M_i}$. It initializes (line 4) each R_i to be the largest propositionally compatible equivalence relation over the states of $\widehat{M_i}$.

Now **TwoLevelCEGAR** begins a loop (line 5) where it first constructs the complete abstract model $\widehat{M}$ (line 6) by composing the quotient LKSs $\widehat{M_1}^{R_1}, \ldots, \widehat{M_n}^{R_n}$. It then checks (line 7) whether $\widehat{M}$ is simulated by the specification Sp. If so, we know that the original programs $\mathcal{P}$ is also simulated by Sp and **TwoLevelCEGAR** terminates successfully. Otherwise (line 8) let CW be a Counterexample Witness to $\widehat{M} \preccurlyeq Sp$.

Now **TwoLevelCEGAR** validates CW (line 9) at the action-guided abstraction level by invoking algorithm **ValidateAndRefineAG**. If **ValidateAndRefineAG** returns TRUE at line 9 means that some R_i was refined. In this case **TwoLevelCEGAR** repeats the loop from line 5. Otherwise it attempts to refine one of the predicate abstractions. This is done in lines 10–16 and exactly as in Procedure **SimulCEGAR**. The final result is either (i) the production of a real counterexample (line 11), or (ii) an ERROR report (line 13), or (iii) a refinement of some predicate abstraction M_i (line 14–15) and the repetition of the loop (line 16). The correctness of **TwoLevelCEGAR** follows from that of **SimulCEGAR** and of **ValidateAndRefineAG**.

9.6 Experimental Results

Our experiments were carried out with two broad goals in mind. The first goal was to compare the overall effectiveness of the proposed two-level CEGAR approach, particularly insofar as memory usage is concerned. The second goal was to verify the effectiveness of our LKS abstraction scheme by itself. To this end, we carried out experiments on 39 benchmarks, of which 26 were sequential programs and 13 were concurrent programs. Each example was verified twice, once with only predicate abstraction, and once with the full two-level algorithm. Tests that used only the low-level predicate abstraction refinement scheme are marked by *PredOnly* in our tables. In contrast, tests that also incorporated our action-guided abstraction refinement procedure are marked by *BothAbst*. Both schemes started out with the same initial sets of predicates.

For each experiment we measured several quantities: (i) the size of the final state-space on which the property was proved/disproved; note that, since our abstraction-refinement scheme produces increasingly refined models, and since we reuse memory from one iteration to the next, the size of the final state-space is a good indicator of the *maximum* memory used, (ii) the number of predicate refinement iterations required, (iii) the number of action-guided refinement iterations required, (iv) the total number of refinement iterations required, and (v) the total time required. In the tables summarizing our results, these measurements are reported in columns named respectively *St, PIt, LIt, It* and *T*. For the concurrent benchmarks, we also measured actual memory requirement and report these in the columns named *Mem*. Note that predicate minimization (cf. Chapter 7) was turned on during all the experiments described in this section.

9.6.1 Unix Kernel Benchmarks

The first set of examples was designed to examine how our approach works on a wide spectrum of implementations. The summary of our results on these examples is presented in Table 9.1. We chose ten code fragments from the Linux Kernel 2.4.0.

LOC	**Description**	**PredOnly**			**BothAbst**		
		St	*It*	*T*	*St*	*It*	*T*
27	*pthread_mutex_lock* (pthread)	26	1	52	16	3	54
24	*pthread_mutex_unlock* (pthread)	27	1	51	13	2	56
60	*socket* (socket)	187	3	1752	44	25	2009
24	*sock_alloc* (socket)	50	2	141	14	4	154
4	*sys_send* (socket)	7	1	92	6	1	93
11	*sock_sendmsg* (socket)	23	1	108	14	3	113
27	modified *pthread_mutex_lock*	23	1	59	14	2	61
24	modified *pthread_mutex_unlock*	27	1	61	12	2	66
24	modified *sock_alloc*	47	1	103	9	1	106
11	modified *sock_sendmsg*	21	1	96	10	1	97

Table 9.1: Summary of results for Linux Kernel code. **LOC** and **Description** denote the number of lines of code and a brief description of the benchmark source code. The measurements for *PIter* and *LIter* have been omitted because they are insignificant. All times are in milliseconds.

Corresponding to each code fragment we constructed a specification from the Linux manual pages. For example, the specification in the third benchmark[2] states that the socket system call either properly allocates internal data structures for a new socket and returns 1, or fails to do so and returns an appropriate negative error value.

9.6.2 OpenSSL Benchmarks

The next set of examples was aimed at verifying larger pieces of code. Once again we used OpenSSL handshake implementation to design a set of 29 benchmarks. However, unlike the previous OpenSSL benchmarks, some of these benchmarks were concurrent and comprised of both a client and a server component executing in parallel. The specifications were derived from the official SSL design documents. For example, the specification for the first concurrent benchmark states that the handshake is always initiated by the client.

The first 16 examples are sequential implementations, examining different

[2]This benchmark was also used as **socket-y** in the predicate minimization experiments described in the previous section.

PredOnly			**BothAbst**					**Gain**
St(S1)	*It*	*T*	*St(S2)*	*PIt*	*LIt*	*It*	*T*	*S1/S2*
597	4	141	114	7	193	200	180	**5.24**
1038	10	191	114	11	275	286	210	**9.11**
849	14	229	135	13	431	444	243	**6.29**
525	1	18	3	1	0	1	19	**175**
55363	48	762	2597	32	4432	4464	1813	**21.3**
3672	14	256	930	14	1009	1023	390	**3.95**
60570	120	3388	636	8	508	516	274	**95.24**
3600	14	251	750	11	662	673	322	**4.80**
1242	19	222	186	16	463	479	226	**6.68**
1029	18	246	252	18	978	996	303	**4.08**
705	12	196	213	12	644	656	226	**3.31**
1038	16	206	213	14	509	523	216	**4.87**
2422	16	230	483	8	366	374	190	**5.01**
2338	15	218	658	16	726	742	273	**3.55**
2366	19	250	665	15	716	731	269	**3.56**
2422	20	257	609	15	710	725	274	**3.98**

Table 9.2: Summary of results for sequential OpenSSL examples. The first eight are server benchmarks while the last eight are client benchmarks. Note that for the **PredOnly** case, LIt is always zero and $PIt = It$. All times are in seconds. The improvement in state-space size is shown in bold.

properties of **SrvrCode** and **ClntCode** separately. Each of these examples contains about 350 comment-free LOC. The results for these are summarized in Table 9.2. The remaining 13 examples test various properties of **SrvrCode** and **ClntCode** when executed together. These examples are concurrent and consist of about 700 LOC. The results for them are summarized in Table 9.3. All OpenSSL benchmarks other than the seventh server benchmark passed the property.

In terms of state-space size, the two-level refinement scheme outperforms the one-level scheme by factors of up to 175. The fourth server benchmark shows particular improvement with the two-level approach. In this benchmark, the property holds on the very initial abstraction, thereby requiring no refinement and letting us achieve maximum reduction in state-space. The two-level approach is also an improvement in terms of actual memory usage, particularly for the concurrent benchmarks. In most instances it reduces the memory requirement by over an order of magnitude.

Finally, the two-level approach is also faster on most of the concurrent benchmarks.

PredOnly				**BothAbst**						**Gain**
St	*It*	*T*	*Mem(M1)*	*St(S2)*	*PIt*	*LIt*	*It*	*T*	*Mem(M2)*	*M1/M2*
157266	12	**886**	1023	15840	13	742	755	1081	122	**8.39**
201940	18	1645	1070	6072	10	547	557	**500**	64	**16.72**
203728	12	**1069**	1003	20172	13	908	921	1805	130	**7.72**
201940	17	1184	640	7808	11	439	450	**482**	69	**9.28**
184060	16	1355	780	6240	8	384	392	**407**	64	**12.19**
158898	11	695	426	2310	5	195	200	**219**	56	**7.61**
103566	10	**447**	250	7743	11	513	524	472	74	**3.38**
161580	14	1071	945	4617	11	464	475	**387**	64	**14.77**
214989	13	1515	1475	13800	8	471	479	**716**	106	**13.92**
118353	10	628	663	3024	12	550	562	**402**	60	**11.05**
204708	8	794	1131	8820	5	306	311	**446**	79	**14.32**
121170	5	303	373	2079	5	152	157	**204**	56	**6.66**
152796	12	579	361	3780	10	404	414	**349**	60	**6.02**

Table 9.3: Summary of results for concurrent OpenSSL examples. Note that for the **PredOnly** case, LIt is always zero and $PIt = It$. All times are in seconds and memory is in MB. Best times and the improvement in memory requirement is shown in bold.

In many instances it achieves a speedup by a factor of over three when compared to the one-level scheme. The savings in time and space for the concurrent examples are significantly higher than for the sequential ones. We expect the two-level approach to demonstrate increasingly improved performance with the number of concurrent components in the implementation.

Chapter 10

Deadlock

In this chapter, we present an algorithm to detect deadlocks in concurrent message-passing programs. Even though deadlock is inherently non-compositional and its absence is not preserved by standard abstractions, our framework employs both *abstraction* and *compositional reasoning* to alleviate the state space explosion problem. We iteratively construct increasingly more precise abstractions on the basis of spurious counterexamples to either detect a deadlock or prove that no deadlock exists. Our approach is inspired by the counterexample guided abstraction refinement paradigm. However, our notion of abstraction as well as our schemes for verification and abstraction refinement differ in key respects from existing abstraction refinement frameworks. Our algorithm is also compositional in that abstraction, counterexample validation, and refinement are all carried out component-wise and do not require the construction of the complete state space of the concrete system under consideration. Finally, our approach is completely *automated* and provides diagnostic feedback in case a deadlock is detected. We have implemented our technique in the MAGIC verification tool and present encouraging results (up to 20 times speed-up in time and 4 times less memory consumption) with concurrent message-passing C programs. We also report a bug in the real-time operating system μC/OS-II.

10.1 Introduction

Ensuring that standard software components are assembled in a way that guarantees the delivery of reliable services is an important task for system designers. Certifying the absence of deadlock in a composite system is an example of a stringent requirement that has to be satisfied before the system can be deployed in real life. This is especially true for safety-critical systems, such as embedded systems or controllers, that are expected to always service requests within a fixed time limit or be responsive to external stimuli. In addition, many formal analysis techniques, such as temporal logic model checking [32, 39], assume that the systems being analyzed are deadlock-free. In order for the results of such analysis to be valid, one usually needs to establish deadlock freedom separately. Last but not least, in case a deadlock is detected, it is highly desirable to be able to provide system designers and implementers with appropriate diagnostic feedback.

However, despite significant efforts, validating the absence of deadlock in systems of realistic complexity remains a major challenge. The problem is especially acute in the context of concurrent programs that communicate via mechanisms with blocking semantics, e.g., synchronous message-passing and semaphores. The primary obstacle is the well-known *state space explosion* problem whereby the size of the state space of a concurrent system increases exponentially with the number of components. Two paradigms are usually recognized as being the most effective against the state space explosion problem: *abstraction* and *compositional reasoning*. Even though these two approaches have been widely studied in the context of formal verification [36, 64, 67, 84], they find much less use in deadlock detection. This is possibly a consequence of the fact that deadlock is inherently non-compositional and its absence is not preserved by standard abstractions (see Example 20). Therefore, a compositional and abstraction-based deadlock detection scheme, such as the one we present in this chapter, is especially significant.

Counterexample guided abstraction refinement [76] (CEGAR for short) is a methodology that uses abstraction in an automated manner and has been successful in verifying real-life hardware [37] and software [6] systems. A CEGAR-based scheme iteratively computes more and more precise abstractions (starting with a very coarse one) of a target system on the basis of spurious counterexamples until a real counterexample is obtained or the system is found to be correct. The approach presented in this chapter combines both abstraction and compositional reasoning within a CEGAR-based framework for verifying the absence of deadlocks in concurrent message-passing systems. More precisely, suppose we have a system M composed of components $M_1, \ldots, M_n$ executing concurrently. Then our technique checks for deadlock in M using the following three-step iterative process:

1. **Abstract.** Create an abstraction $\widehat{M}$ such that if M has a deadlock, then so does $\widehat{M}$. This is done component-wise without having to construct the full state space of M.

2. **Verify.** Check if $\widehat{M}$ has a deadlock. If not, report absence of deadlock in M and exit. Otherwise let π be a counterexample that leads to a deadlock in $\widehat{M}$.

3. **Refine.** Check if π corresponds to a deadlock in M. Once again this is achieved component-wise. If π corresponds to a real deadlock, report presence of deadlock

in M along with appropriate diagnostic feedback and exit. Otherwise refine $\widehat{M}$ on the basis of π to obtain a more precise abstraction and repeat from step 1.

In our approach, systems as well as their components are represented as LKSs. Note that only the verification stage (step 2) of our technique requires explicit composition of systems. All other stages can be performed one component at a time. Since verification is performed only on abstractions (which are usually much smaller than the corresponding concrete systems), this technique is able to significantly reduce the state space explosion problem. Finally, when a deadlock is detected, our scheme provides useful diagnostic feedback in the form of counterexamples.

To the best of our knowledge, this is the first counterexample guided, compositional abstraction refinement scheme to perform deadlock detection on concurrent systems. We have implemented our approach in our C verification tool MAGIC [80] which extracts LKS models from C programs automatically via predicate abstraction [24, 63]. Our experiments with a variety of benchmarks have yielded encouraging results (up to 20 times speed-up in time and 4 times less memory consumption). We have also discovered a bug in the real-time operating system μC/OS-II.

The rest of this chapter is organized as follows. In Section 10.2 we summarize related work. This is followed by some preliminary definitions and results in Section 10.3. In Section 10.4 we present our abstraction scheme, followed by counterexample validation and abstraction refinement in Section 10.5 and Section 10.6 respectively. Our overall deadlock detection algorithm is described in Section 10.7. Finally, we present experimental results in Section 10.8.

10.2 Related Work

The formalization of a general notion of abstraction first appeared in [47]. The abstractions used in our approach are *conservative*. They are only guaranteed to preserve safety properties of the system (e.g., [36, 75]). Conservative abstractions usually lead to significant reductions in the state space but in general require an iterated abstraction refinement mechanism (such as CEGAR) in order to establish specification satisfaction. CEGAR has been used, among others, in non-automated [90], and automated [6, 29, 40, 66, 77, 98] forms.

CEGAR-based schemes have been used for the verification of both safety [6, 24, 37, 66] (i.e., reachability) and liveness [28] properties. Compositionality has been most extensively studied in process algebra (e.g., [69, 85, 100]), particularly in conjunction with abstraction. Abstraction and compositional reasoning have been combined [23] within a single two-level CEGAR scheme to verify safety properties of concurrent message-passing C programs. None of these techniques attempt to detect deadlock. In fact, the abstractions used in these schemes do not preserve deadlock freedom and hence cannot be used directly in our approach.

Deadlock detection has been widely studied in various contexts. One of the earliest deadlock-detection tools, for the process algebra CSP, was FDR [60]; see also [17, 82, 83, 100, 101]. Corbett has evaluated various deadlock-detection methods for concurrent systems [45] while Demartini et. al. have developed deadlock-detection tools for concurrent Java programs [53]. However, to the best of our knowledge, none of these approaches involve abstraction refinement or compositionality in automated form.

10.3 Background

Recall, from Definition 1, than an LKS is a 6-tuple $(S, Init, AP, L, \Sigma, T)$. In this section, we present some additional preliminary definitions and results (many of which originate from CSP [69, 100]) that are used in the rest of the chapter. In this chapter, we will only concern ourselves with *finite* paths and traces (which will be usually represented with the letters π and θ respectively). The deadlocking behavior of a system is dependent purely on the communication between its components. Since communication in our framework is based purely on actions, our notion of a trace will ignore atomic propositions. We now define paths and traces formally.

Definition 32 (Finite Path and Trace) *Let $M = (S, Init, AP, L, \Sigma, T)$ be an LKS. A finite path of M is a finite sequence $\langle s_0, a_0, s_1, a_1, \ldots, a_{n-1}, s_n\rangle$ such that: (i) $s_0 \in Init$ and (ii) $\forall 0 \leq i < n \centerdot s_i \xrightarrow{a_i} s_{i+1}$. In such a case, the finite sequence $\langle a_0, a_1, \ldots, a_{n-1}\rangle$ is called a finite trace of M.*

Let $M = (S, Init, AP, L, \Sigma, T)$ be any LKS. We denote the set of all paths of M by $Path(M)$. A state s of M is said to *refuse* an action α iff $Succ(s, \alpha) = \emptyset$. The *refusal* of a state is the set of all actions that it refuses. Suppose $\theta \in \Sigma^*$ is a finite sequence of actions and $F \subseteq \Sigma$ is a set of actions. Then (θ, F) is said to be a *failure* of M iff M can participate in the sequence of actions θ and then reach a state whose refusal is F. Finally, M has a *deadlock* iff it can reach a state which refuses the entire alphabet Σ. We now present these notions formally.

Definition 33 (Refusal) *Let $M = (S, Init, AP, L, \Sigma, T)$ be an LKS. Then the function $Ref : S \to 2^{\Sigma}$ is defined as follows:*

$$Ref(s) = \{\alpha \in \Sigma \mid Succ(s, \alpha) = \emptyset\}$$

Definition 34 (Failure) *Let $M = (S, Init, AP, L, \Sigma, T)$ be an LKS. A pair $(\theta, F) \in \Sigma^* \times 2^{\Sigma}$ is a failure of M iff the following condition holds: if $\theta = \langle a_0, \dots, a_{n-1} \rangle$, then there exist states $s_0, s_1, \dots, s_n$ such that (i) $\langle s_0, a_0, s_1, a_1, \dots, a_{n-1}, s_n \rangle \in Path(M)$ and (ii) $F = Ref(s_n)$. We write $Fail(M)$ to denote the set of all failures of M.*

Definition 35 (Deadlock) *An LKS $M = (S, Init, AP, L, \Sigma, T)$ is said to have a deadlock iff $(\theta, \Sigma) \in Fail(M)$ for some $\theta \in \Sigma^*$.*

Example 19 *Figure 10.1(a) shows two LKSs $M_1 = (S_1, Init_1, AP_1, L_1, \Sigma_1, T_1)$ and $M_2 = (S_2, Init_2, AP_2, L_2, \Sigma_2, T_2)$. Let $\Sigma_1 = \{a, b, c\}$ and $\Sigma_2 = \{a, b', c\}$. Then M_1 has seven paths: $\langle P \rangle$, $\langle P, a, Q \rangle$, $\langle P, a, R \rangle$, $\langle P, a, Q, b, S \rangle$, $\langle P, a, R, b, S \rangle$, $\langle P, a, Q, b, S, c, T \rangle$, and $\langle P, a, R, b, S, c, T \rangle$. It has four traces: $\langle \rangle$, $\langle a \rangle$, $\langle a, b \rangle$, and $\langle a, b, c \rangle$, and four failures $(\langle \rangle, \{b, c\})$, $(\langle a \rangle, \{a, c\})$, $(\langle a, b \rangle, \{a, b\})$, and $(\langle a, b, c \rangle, \{a, b, c\})$. Hence M_1 has a deadlock. Also, M_2 has four paths, four traces, four failures and a deadlock. Finally, Figure 10.1(b) shows the LKS $M_1 \parallel M_2$ where M_1 and M_2 are the LKSs shown in Figure 10.1(a).*

Given a trace of a concurrent system $M_{\parallel}$, one can construct *projections* by restricting the trace to the alphabets of each of the components of $M_{\parallel}$. In the following, we will write $\theta_1 \bullet \theta_2$ to denote the concatenation of two sequences θ_1 and θ_2.

Definition 36 (Projection) *Consider LKSs $M_1, \dots, M_n$ with alphabets $\Sigma_1, \dots, \Sigma_n$ respectively. Let $M_{\parallel} = M_1 \parallel \cdots \parallel M_n$ and let us denote the alphabet of $M_{\parallel}$ by $\Sigma_{\parallel}$. Then for $1 \leq i \leq n$, the projection function $Proj_i : \Sigma_{\parallel}^* \to \Sigma_i^*$ is defined inductively as follows. We will write $\theta \downharpoonleft i$ to mean $Proj_i(\theta)$:*

1. *$\langle \rangle \downharpoonleft i = \langle \rangle$.*

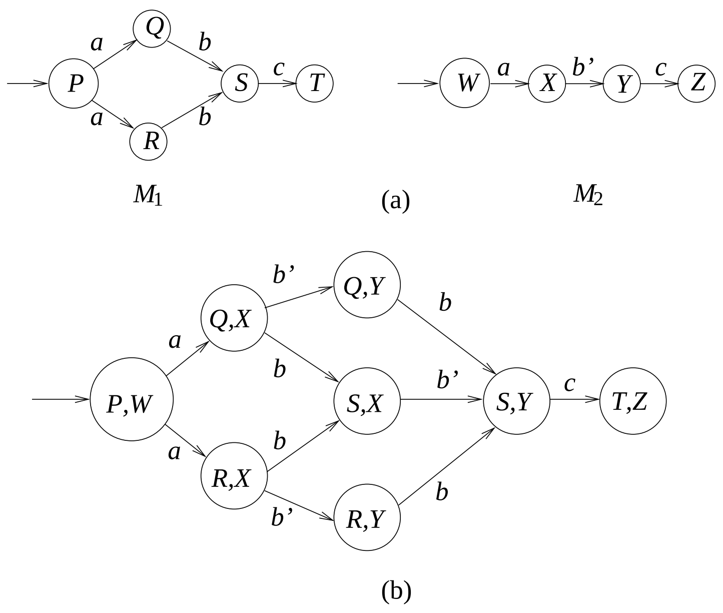


Figure 10.1: (a) Sample LKSs M_1 and M_2; (b) $M_1 \parallel M_2$.

2. *If* $a \in \Sigma_i$ *then* $(\langle a \rangle \bullet \theta) \downharpoonleft i = \langle a \rangle \bullet (\theta \downharpoonleft i)$.

3. *If* $a \notin \Sigma_i$ *then* $(\langle a \rangle \bullet \theta) \downharpoonleft i = \theta \downharpoonleft i$.

Definition 5 for the parallel composition of LKSs and Definition 36 immediately lead to the following theorem, which essentially highlights the compositional nature of failures. Its proof, as well as the proofs of related results, are well-known [100].

Theorem 26 *Let* $M_1, \ldots, M_n$ *be LKSs and let* $M_{\parallel} = M_1 \parallel \cdots \parallel M_n$. *Then* $(\theta, F) \in Fail(M_{\parallel})$ *iff there exist refusals* $F_1, \ldots, F_n$ *such that: (i)* $F = \bigcup_{i=1}^{n} F_i$, *and (ii) for* $1 \leq i \leq n$, $(\theta \downharpoonleft i, F_i) \in Fail(M_i)$.

10.4 Abstraction

In this section we present our notion of abstraction. Once again, we employ quotient LKSs as abstractions of concrete LKSs. Recall that given a concrete LKS M, one can obtain a quotient LKS as follows. The states of the quotient LKS are obtained by grouping together states of M; alternatively, one can view these groups as equivalence classes of some equivalence relation on the set of states of M. Transitions of the quotient LKS are defined *existentially.* We now present a formal definition of these concepts.

Definition 37 (Quotient LKS) *Let* $M = (S, Init, AP, L, \Sigma, T)$ *be an LKS and* $R \subseteq S \times S$ *be an equivalence relation. For an arbitrary* $s \in S$ *we let* $[s]^R$ *denote the equivalence class of* s*.* M *and* R *then induce a quotient LKS* $M^R = \big(S^R, Init^R, AP^R, L^R, \Sigma^R, T^R\big)$ *where: (i)* $S^R = \{[s]^R \mid s \in S\}$*, (ii)* $Init^R = \{[s]^R \mid s \in Init\}$*, (iii)* $AP^R = AP$*, (iv)* $\forall [s]^R \in S^R \centerdot L^R([s]^R) = \bigcup_{s' \in [s]^R} L(s')$*, (v)* $\Sigma^R = \Sigma$*, and (vi)* $T^R = \{([s]^R, a, [s']^R) \mid (s, a, s') \in T\}$*.*

Note that the crucial difference between Definition 28 and Definition 37 is that in the latter we do not require equivalence relations to be propositionally compatible. Instead we let the set of propositions labeling a state $[s]^R$ of M^R be simply the union of the propositions labeling the states of M belonging to the equivalence class $[s]^R$. This definition is somewhat arbitrary but suffices for our deadlock detection framework since propositions do not play any role in the deadlocking behavior of a system.

As usual, we write $[s]$ to mean $[s]^R$ when R is clear from the context. M^R is often called an *existential abstraction* of M. The states of M are referred to as *concrete states* while those of M^R are called *abstract states.* We will often use α to represent abstract states, and continue to denote concrete states with s. The following result concerning quotient LKSs is well-known [39].

Theorem 27 *Let $M = (S, Init, AP, L, \Sigma, T)$ be an LKS, R an equivalence relation on S, and M^R the quotient LKS induced by M and R. If $\langle s_0, a_0, s_1, a_1, \ldots, a_{n-1}, s_n\rangle \in Path(M)$, then $\langle [s_0], a_0, [s_1], a_1, \ldots, a_{n-1}, [s_n]\rangle \in Path(M^R)$.*

Example 20 *Note the following facts about the LKSs in Figure 10.2: (i) M_1 and M_2 both have deadlocks but $M_1 \parallel M_2$ does not; (ii) neither M_3 nor M_4 has a deadlock but $M_3 \parallel M_4$ does; (iii) M_1 has a deadlock and M_3 does not have a deadlock but $M_1 \parallel M_3$ has a deadlock; (iv) M_1 has a deadlock and M_4 does not have a deadlock but $M_1 \parallel M_4$ does not have a deadlock; (v) M_1 has a deadlock but the quotient LKS obtained by grouping all the states of M_1 into a single equivalence class does not have a deadlock.*

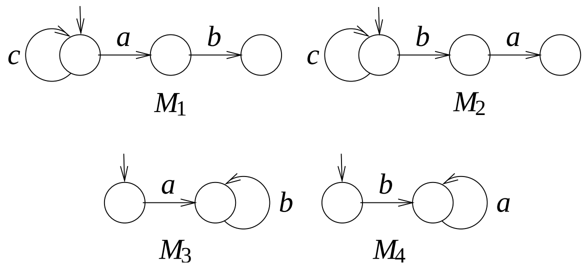


Figure 10.2: Four sample LKSs demonstrating the non-compositional nature of deadlock.

As Example 20 highlights, deadlock is non-compositional and its absence is not preserved by existential abstractions (nor in fact is it preserved by *universal* abstractions). So far we have presented well-known definitions and results to prepare the background. We now present what constitute the core technical contributions of this chapter.

We begin by taking a closer look at the non-preservation of deadlock by existential abstractions. Consider a quotient LKS M^R and a state $[s]$ of M^R. It can be proved that $Ref([s]) = \bigcap_{s' \in [s]} Ref(s')$. In other words, the refusal of an abstract state $[s]$ under-approximates the refusals of the corresponding concrete states. However, in order to preserve deadlock we require that refusals be *over-approximated*. We achieve this by taking the union of the refusals of the concrete states. This leads to the notion of an *abstract refusal*, which we now define formally.

Definition 38 (Abstract Refusal) *Let $M = (S, Init, AP, L, \Sigma, T)$ be an LKS, $R \subseteq S \times S$ be an equivalence relation, and M^R be the quotient LKS induced by M and R. Let S^R be the set of states of M^R. Then the abstract refusal function $\widehat{Ref} : S^R \rightarrow 2^\Sigma$ is defined as follows:*

$$\widehat{Ref}(\alpha) = \bigcup_{s \in \alpha} Ref(s)$$

For a parallel composition of quotient LKSs, we extend the notion of abstract refusal as follows. Let $M_1^{R_1}, \ldots, M_n^{R_n}$ be quotient LKSs. Let $\alpha = (\alpha_1, \ldots, \alpha_n)$ be a state of $M_1^{R_1} \parallel \cdots \parallel M_n^{R_n}$. Then $\widehat{Ref}(\alpha) = \bigcup_{i=1}^n \widehat{Ref}(\alpha_i)$.

Next, we introduce the notion of *abstract failures*, which are similar to failures, except that abstract refusals are used in place of refusals.

Definition 39 (Abstract Failure) *Let $\widehat{M} = (S, Init, AP, L, \Sigma, T)$ be an LKS for which abstract refusals are defined (i.e., $\widehat{M}$ is either a quotient LKS or a parallel composition of such). A pair $(\theta, F) \in \Sigma^* \times 2^\Sigma$ is said to be an abstract failure of $\widehat{M}$ iff the following condition holds: if $\theta = \langle a_0, \ldots, a_{n-1} \rangle$, then there exist $\alpha_0, \alpha_1, \ldots, \alpha_n$ such that (i) $\langle \alpha_0, a_0, \alpha_1, a_1, \ldots, a_{n-1}, \alpha_n \rangle \in Path(\widehat{M})$ and (ii) $F = \widehat{Ref}(\alpha_n)$. We write $AbsFail(\widehat{M})$ to denote the set of all abstract failures of $\widehat{M}$.*

The following theorem essentially states that the failures of an LKS M are always subsumed by the abstract failures of its quotient LKS M^R.

Theorem 28 *Let $M = (S, Init, AP, L, \Sigma, T)$ be an LKS, $R \subseteq S \times S$ be an equivalence relation, and M^R be the quotient LKS induced by M and R. Then for all $(\theta, F) \in Fail(M)$, there exists $F' \supseteq F$ such that $(\theta, F') \in AbsFail(M^R)$.*

Proof. Here is a proof sketch. Let $\theta = \langle a_0, \ldots, a_{n-1} \rangle$.

1. From $(\theta, F) \in Fail(M)$ and Definition 34: let $\langle s_0, a_0, s_1, a_1, \ldots, a_{n-1}, s_n \rangle \in Path(M)$ such that $F = Ref(s_n)$.
2. From 1 and Proposition 27: $\langle [s_0], a_0, [s_1], a_1, \ldots, a_{n-1}, [s_n] \rangle \in Path(M^R)$.
3. From 2 and Definition 39: $(\theta, \widehat{Ref}([s_n])) \in AbsFail(M^R)$.
4. From Definition 38: $\widehat{Ref}([s_n]) \supseteq Ref(s_n)$.
5. From 3, 4 and using $F' = \widehat{Ref}([s_n])$ we get our result.

□

As the following two theorems show, abstract failures are compositional, In other words, the abstract failures of a concurrent system $M_\|$ can be decomposed naturally into abstract failures of the components of $M_\|$. Proofs of Theorem 29 and Theorem 30 follow the same lines as Theorem 26.

Theorem 29 *Let $M_1^{R_1}, \ldots, M_n^{R_n}$ be quotient LKSs, and $\langle \alpha_0, a_0, \ldots, a_{k-1}, \alpha_k \rangle \in Path(M_1^{R_1} \parallel \cdots \parallel M_n^{R_n})$. Let the trace $\theta = \langle a_0, \ldots, a_{k-1} \rangle$ and the final state $\alpha_k = (\alpha_k^1, \ldots, \alpha_k^n)$. Then for $1 \leq i \leq n$, $(\theta \downharpoonleft i, \widehat{Ref}(\alpha_k^i)) \in AbsFail(M_i^{R_i})$.*

Theorem 30 *Let $M_1^{R_1}, \ldots, M_n^{R_n}$ be quotient LKSs. Then $(\theta, F) \in AbsFail(M_1^{R_1} \parallel \cdots \parallel M_n^{R_n})$ iff there exist abstract refusals $F_1, \ldots, F_n$ such that: (i) $F = \bigcup_{i=1}^n F_i$, and (ii) for $1 \leq i \leq n$, $(\theta \downharpoonleft i, F_i) \in AbsFail(M_i^{R_i})$.*

In the rest of this chapter we often make implicit use of the following facts. Consider LKSs $M_1, \ldots, M_n$ with alphabets $\Sigma_1, \ldots, \Sigma_n$ respectively. Let $M_1^{R_1}, \ldots, M_n^{R_n}$ be quotient LKSs. Let us denote the alphabet of $M_1 \parallel \cdots \parallel M_n$ by $\Sigma_\parallel$ and the alphabet of $M_1^{R_1} \parallel \cdots \parallel M_n^{R_n}$ by $\widehat{\Sigma_\parallel}$. Then $\widehat{\Sigma_\parallel} = \Sigma_\parallel$. This follows directly from the fact the alphabet of $M_i^{R_i}$ is Σ_i for $1 \leq i \leq n$. The notion of abstract failures leads naturally to the notion of *abstract deadlocks*.

Definition 40 (Abstract Deadlock) *Let $M_1^{R_1}, \ldots, M_n^{R_n}$ be quotient LKSs and $\widehat{M_\parallel} = M_1^{R_1} \parallel \cdots \parallel M_n^{R_n}$. Let Σ be the alphabet of $\widehat{M_\parallel}$. Then $\widehat{M_\parallel}$ is said to have an abstract deadlock iff $(\theta, \Sigma) \in AbsFail(\widehat{M_\parallel})$ for some $\theta \in \Sigma^*$.*

Let $M_1^{R_1}, \ldots, M_n^{R_n}$ be quotient LKSs and $\widehat{M_\parallel} = M_1^{R_1} \parallel \cdots \parallel M_n^{R_n}$ with alphabet Σ. Clearly, $\widehat{M_\parallel}$ has an abstract deadlock iff there exists a path $\langle \alpha_0, a_0, \alpha_1, a_1, \ldots, a_{n-1}, \alpha_n \rangle$ of $\widehat{M_\parallel}$ such that $\widehat{Ref}(\alpha_n) = \Sigma$. We call such a path a counterexample to abstract deadlock freedom, or simply an abstract counterexample. It is easy to devise an algorithm to check whether $\widehat{M_\parallel}$ has an abstract deadlock and also generate a counterexample in case an abstract deadlock is detected. We call this algorithm **AbsDeadlock**.

Suppose the alphabet of $\widehat{M_\parallel}$ if Σ. Then **AbsDeadlock** explores the reachable states of $\widehat{M_\parallel}$ in, say, breadth-first manner. For each state α of $\widehat{M_\parallel}$, it checks if $\widehat{Ref}(\alpha) = \Sigma$. If so, it generates a counterexample from an initial state of $\widehat{M_\parallel}$ to α, reports "*abstract deadlock*" and terminates. If no state α with $\widehat{Ref}(\alpha) = \Sigma$ can be found, it reports "*no abstract deadlock*" and terminates. Since $\widehat{M_\parallel}$ has a finite number of states and transitions, **AbsDeadlock** always terminates with the correct answer.

The following lemma shows that abstract deadlock freedom in the composition of quotient LKSs entails deadlock freedom in the composition of the corresponding concrete LKSs.

Lemma 1 *Let $M_1, \ldots, M_n$ be LKSs and $R_1, \ldots, R_n$ be equivalence relations on the state of $M_1, \ldots, M_n$ respectively. If $M_1^{R_1} \parallel \cdots \parallel M_n^{R_n}$ does not have an abstract deadlock then $M_1 \parallel \cdots \parallel M_n$ does not have a deadlock.*

Proof. It suffices to prove the contrapositive. Let us denote $M_1 \parallel \cdots \parallel M_n$ by $M_\parallel$ and $M_1^{R_1} \parallel \cdots \parallel M_n^{R_n}$ by $\widehat{M_\parallel}$. We know that $M_\parallel$ and $\widehat{M_\parallel}$ have the same alphabet. Let this alphabet be Σ. Now suppose $M_\parallel$ has a deadlock.

1. By Definition 35: $(\theta, \Sigma) \in \mathit{Fail}(M_\parallel)$ for some θ.

2. From 1 and Theorem 26: there exist $F_1, \ldots, F_n$ such that: (i) $\bigcup_{i=1}^n F_i = \Sigma$ and (ii) for $1 \leq i \leq n$, $(\theta \downharpoonleft i, F_i) \in \mathit{Fail}(M_i)$.

3. From 2(ii) and Theorem 28: for $1 \leq i \leq n$, $\exists F_i' \supseteq F_i$ such that $(\theta \downharpoonleft i, F_i') \in \mathit{AbsFail}(M_i^{R_i})$.

4. From 2(i) and 3: $\bigcup_{i=1}^n F_i' \supseteq \bigcup_{i=1}^n F_i = \Sigma$.

5. From 3, 4 and Theorem 30: $(\theta, \Sigma) \in \mathit{AbsFail}(\widehat{M_\parallel})$.

6. From 5 and Definition 40: $\widehat{M_\parallel}$ has an abstract deadlock.

□

Unfortunately, the converse of Lemma 1 does not hold (a counterexample is not difficult to find and we leave this task to the reader). Suppose therefore that **AbsDeadlock** reports an abstract deadlock for $M_1^{R_1} \parallel \cdots \parallel M_n^{R_n}$ along with an abstract counterexample π. We must then decide whether π also leads to a deadlock in $M_1 \parallel \cdots \parallel M_n$ or not. This process is called counterexample validation and is presented in the next section.

10.5 Counterexample Validation

In this section we present our approach to check the validity of an abstract counterexample returned by **AbsDeadlock**. We begin by defining the notion of *valid counterexamples.*

Definition 41 (Valid Counterexample) *Let* $M_1^{R_1}, \ldots, M_n^{R_n}$ *be quotient LKSs and let* $\pi = \langle \alpha_0, a_0, \ldots, a_{k-1}, \alpha_k \rangle$ *be an abstract counterexample returned by* **AbsDeadlock** *on* $M_1^{R_1} \parallel \cdots \parallel M_n^{R_n}$*. Let the trace* $\theta = \langle a_0, \ldots, a_{k-1} \rangle$ *and the final state* $\alpha_k = (\alpha_k^1, \ldots, \alpha_k^n)$*. We say that* π *is a valid counterexample iff for* $1 \leq i \leq n$*,* $(\theta \downharpoonright i, \widehat{Ref}(\alpha_k^i)) \in Fail(M_i)$*.*

A counterexample is said to be *spurious* iff it is not valid. Let M be an arbitrary LKS with alphabet Σ, $\theta \in \Sigma^*$ be a trace, and $F \subseteq \Sigma$ be a refusal. It is easy to design an algorithm that takes M, θ, and F as inputs and returns TRUE if $(\theta, F) \in Fail(M)$ and FALSE otherwise. We call this algorithm **IsFailure** and give its pseudo-code in Procedure 10.1. Starting with the initial state, **IsFailure** repeatedly computes successors for the sequence of actions in θ. If the set of successors obtained at some point during this process is empty, then $(\theta, F) \notin Fail(M)$ and **IsFailure** returns FALSE. Otherwise, if X is the set of states obtained after all actions in θ have been processed, then $(\theta, F) \in Fail(M)$ iff there exists $s \in X$ such that $Ref(s) = F$. The correctness of **IsFailure** should be clear from Definition 34.

Lemma 2 *Let* $M_1^{R_1}, \ldots, M_n^{R_n}$ *be quotient LKSs and let* π *be an abstract counterexample returned by* **AbsDeadlock** *on* $M_1^{R_1} \parallel \cdots \parallel M_n^{R_n}$*. If* π *is a valid counterexample then* $M_1 \parallel \cdots \parallel M_n$ *has a deadlock.*

Procedure 10.1 IsFailure returns TRUE if $(\theta, F) \in Fail(M)$ and FALSE otherwise.

```
Algorithm IsFailure($M, \theta, F$)
- $M$ : is an LKS, $\theta$ : is a trace of $M$, $F$ : is a set of actions of $M$
let $M = (S, Init, AP, L, \Sigma, T)$ and $\theta = \langle a_0, \ldots, a_{n-1} \rangle$;
$X := Init$;
//simulate $\theta$ on $M$
for $i := 0$ to $n - 1$ do $X := \bigcup_{s \in X} Succ(s, a_i)$;
//simulation complete, now check if one of the end states refuses $F$
return $\exists s \in X \centerdot Ref(s) = F$;
```

Proof. Let us denote $M_1 \parallel \cdots \parallel M_n$ by $M_\parallel$ and $M_1^{R_1} \parallel \cdots \parallel M_n^{R_n}$ by $\widehat{M_\parallel}$. Once again we know that $M_\parallel$ and $\widehat{M_\parallel}$ have the same alphabet. Let this alphabet be Σ. Also let $\pi = \langle \alpha_0, a_0, \ldots, a_{k-1}, \alpha_k \rangle$, $\theta = \langle a_0, \ldots, a_{k-1} \rangle$, and $\alpha_k = (\alpha_k^1, \ldots, \alpha_k^n)$.

1. Since π is an abstract counterexample: $\widehat{Ref}(\alpha_k) = \Sigma$.
2. From 1 and Definition 38: $\bigcup_{i=1}^n \widehat{Ref}(\alpha_k^i) = \widehat{Ref}(\alpha_k) = \Sigma$.
3. Counterexample is valid: for $1 \leq i \leq n$, $(\theta \downharpoonleft i, \widehat{Ref}(\alpha_k^i)) \in Fail(M_i)$.
4. From 3 and Theorem 26: $(\theta, \bigcup_{i=1}^n \widehat{Ref}(\alpha_k^i)) \in Fail(M_\parallel)$.
5. From 2, 4 and Definition 35: $M_\parallel$ has a deadlock.

□

10.6 Abstraction Refinement

In case the abstract counterexample π returned by **AbsDeadlock** is found to be spurious, we wish to refine our abstraction on the basis of π and re-attempt the deadlock check. Recall, from Chapter 9, Definition 29, Definition 30, the definition of

$\widehat{Succ}$ and Condition 9.1. For deadlock, abstraction refinement also involves computing proper refinements of equivalence relations based on abstract successors. This is achieved by the algorithm **AbsRefine** presented in Procedure 10.2.

Procedure 10.2 AbsRefine for doing abstraction refinement.

```
Algorithm AbsRefine($M, R, \theta, F$)
- $M$ : is an LKS, $\theta$ : is a trace of $M$, $F$ : is a set of actions of $M$
- $R$ : is an equivalence relation over the states of $M$
1: let $\theta = \langle a_0, \dots, a_{k-1} \rangle$;
2: find $\pi = \langle \alpha_0, a_0, \dots, a_{k-1}, \alpha_k \rangle \in Path(M^R)$ such that $F = \widehat{Ref}(\alpha_k)$;
        // $\pi$ exists because of condition AR1
3: $X := \alpha_0$;
4: for $i := 0$ to $k-1$
5:        $X := (\bigcup_{s \in X} Succ(s, a_i)) \cap \alpha_{i+1}$;
6:        if $X = \emptyset$ then return $Split(M, R, \alpha_i, \{a_i\})$;
7: return $Split(M, R, \alpha_k, \widehat{Ref}(\alpha_k))$;
```

More precisely, **AbsRefine** takes the following as inputs: (i) an LKS $M = (S, Init, AP, L, \Sigma, T)$, (ii) an equivalence relation $R \subseteq S \times S$, (iii) a trace $\theta \in \Sigma^*$, and (iv) a set of actions $F \subseteq \Sigma$. In addition, the inputs to **AbsRefine** must obey the following two conditions: **(AR1)** $(\theta, F) \in AbsFail(M^R)$ and **(AR2)** $(\theta, F) \notin Fail(M)$. **AbsRefine** then computes and returns a proper refinement of R.We now establish the correctness of **AbsRefine**. We consider two possible scenarios.

1. Suppose **AbsRefine** returns from line 6 when the value of i is l. Since $\alpha_l \xrightarrow{a_l} \alpha_{l+1}$ we know that there exists $s \in \alpha_l$ such that $\alpha_{l+1} \in \widehat{Succ}(s, a_l)$. Let X' denote the value of X at the end of the previous iteration. For all $s' \in X'$, $\alpha_{l+1} \notin \widehat{Succ}(s', a_l)$. Note that $X' \neq \emptyset$ as otherwise **AbsRefine** would have terminated with $i = l - 1$. Therefore, there exists $s' \in X'$ such that

$\alpha_{l+1} \notin \widehat{Succ}(s', a_l)$. Hence the call to *Split* at line 6 satisfies Condition 9.1 and **AbsRefine** returns a proper refinement of R.

2. Suppose **AbsRefine** returns from line 7. We know that at this point $X \neq \emptyset$. Pick an arbitrary $s \in X$. It is clear that there exist $s_0, \ldots, s_{k-1}$ such that $\langle s_0, a_0, \ldots, s_{k-1}, a_{k-1}, s \rangle \in Path(M)$. Hence by condition **AR2**, $Ref(s) \neq F$. Again $s \in \alpha_k$, and from the way π has been chosen at line 2, $F = \widehat{Ref}(\alpha_k)$. Hence by Definition 38, $Ref(s) \subseteq F$. Pick $a \in \Sigma_M$ such that $a \in F$ and $a \notin Ref(s)$. Then $\widehat{Succ}(s, a) \neq \emptyset$. Again since $a \in \widehat{Ref}(\alpha_k)$ there exists $s' \in \alpha_k$ such that $a \in Ref(s')$. Hence $\widehat{Succ}(s', a) = \emptyset$. Hence the call to *Split* at line 8 satisfies Condition 9.1 and once again **AbsRefine** returns a proper refinement of R.

10.7 Overall Algorithm

In this section we present our iterative deadlock detection algorithm and establish its correctness. Let $M_1, \ldots, M_n$ be arbitrary LKSs and $M_\| = M_1 \| \cdots \| M_n$. The algorithm **IterDeadlock** takes $M_1, \ldots, M_n$ as inputs and reports whether $M_\|$ has a deadlock or not. If there is a deadlock, it also reports a trace of each M_i that would lead to the deadlock state. Procedure 10.3 gives the pseudo-code for **IterDeadlock**. It is an iterative algorithm and uses equivalence relations $R_1, \ldots, R_n$ such that, for $1 \leq i \leq n$, $R_i \subseteq S_{M_i} \times S_{M_i}$. Note that initially each R_i is set to the trivial equivalence relation $S_{M_i} \times S_{M_i}$.

Theorem 31 *The algorithm* **IterDeadlock** *is correct and always terminates.*

Procedure 10.3 IterDeadlock for iterative deadlock detection.

```
Algorithm IterDeadlock($M_1, \ldots, M_n$)        // $M_1, \ldots, M_n$ : are LKSs
1: for $i := 1$ to $n$, $R_i := S_{M_i} \times S_{M_i}$;
2: forever do
     // abstract and verify
3:   $x :=$ AbsDeadlock($M_1^{R_1}, \ldots, M_n^{R_n}$);
4:   if ($x =$ "no abstract deadlock") then report "no deadlock" and exit;
5:   let $\pi = \langle \alpha_0, a_0, \ldots, a_{k-1}, \alpha_k \rangle$ be the counterexample reported by AbsDeadlock;
6:   let $\theta = \langle a_0, \ldots, a_{k-1} \rangle$ and $\alpha_k = (\alpha_k^1, \ldots, \alpha_k^n)$;
     // validate counterexample
7:   find $i \in \{1, 2, \ldots, n\}$ such that $\neg$IsFailure($M_i, \theta \downharpoonleft i, \widehat{Ref}(\alpha_k^i)$);
8:   if no such $i$ then report "deadlock" and the ($\theta \downharpoonleft i$)'s as counterexamples;
       //$\pi$ is a valid counterexample, hence deadlock exists in $M_1 \parallel \cdots \parallel M_n$
9:   let $R_i :=$ AbsRefine($M_i, R_i, \theta \downharpoonleft i, \widehat{Ref}(\alpha_k^i)$);
       // refine abstraction and repeat
```

Proof. First we argue that both **AR1** and **AR2** are satisfied every time **AbsRefine** is invoked on line 9. The case for **AR1** follows from Theorem 29 and the fact that $\langle \alpha_0, a_0, \ldots, a_{k-1}, \alpha_k \rangle \in Path(M_1^{R_1} \parallel \cdots \parallel M_n^{R_n})$. The case for **AR2** is trivial from line 7 and the definition of **IsFailure**. Next we show that if **IterDeadlock** terminates it does so with the correct answer. There are two possible cases:

1. Suppose **IterDeadlock** exits from line 4. Then we know that $M_1^{R_1} \parallel \cdots \parallel M_n^{R_n}$ does not have an abstract deadlock. Hence by Lemma 1, $M_1 \parallel \cdots \parallel M_n$ does not have a deadlock.

2. Otherwise, suppose **IterDeadlock** exits from line 8. Then we know that for $1 \leq i \leq n$, $(\theta \downharpoonleft i, \widehat{Ref}(\alpha_k^i)) \in Fail(M_i)$. Hence by Definition 41, π is a valid counterexample. Therefore, by Lemma 2, $M_1 \parallel \cdots \parallel M_n$ has a deadlock.

Finally, termination follows from the fact that the **AbsRefine** routine invoked on line 9 always produces a *proper* refinement of the equivalence relation R_i. Since each

M_i has only finitely many states, this process cannot proceed indefinitely. (In fact, the abstract LKSs converge to the bisimulation quotients of their concrete counterparts, since **AbsRefine** each time performs a unit step of the Paige-Tarjan algorithm [96]; however in practice deadlock freedom is often established or disproved well before the bisimulation quotient is achieved).

□

10.8 Experimental Results

We implemented our technique in the MAGIC tool. MAGIC extracts finite LKS models from C programs using predicate abstraction. These LKSs are then analyzed for deadlock using the approach presented in this chapter. Once a real counterexample π is found at the level of the LKSs MAGIC analyzes π and, if necessary, creates more refined models by inferring new predicates. Our actual implementation is therefore a two-level CEGAR scheme. We elide details of the outer predicate abstraction-refinement loop as it is similar to our previous work [23].

Table 10.1 summarizes our results. The *ABB* benchmark was provided to us by our industrial partner, ABB [1] Corporation. It implements part of an interprocess communication protocol (IPC-1.6) used to mediate communication in a multi-threaded robotics control automation system developed by ABB. The implementation is required to satisfy various safety-critical properties, in particular, deadlock freedom. The IPC protocol supports multiple modes of communication, including synchronous point-to-point, broadcast, publish/subscribe, and asynchronous communication. Each of these modes is implemented in terms of messages passed between queues owned by different threads. The protocol handles the creation and manipulation of

Name	*Plain*					**IterDeadlock**				
	S_M	S_R	I	T	M	S_M	S_R	I	T	M
ABB	2.1×10^9	*	*	*	162	4.1×10^5	1973	861	**1446**	**33.3**
SSL	49405	25731	1	44	43.5	16	16	16	31.9	40.8
UCOSD-2	1.1×10^5	5851	5	24	14.5	374	261	77	14.5	12.9
UCOSD-3	2.1×10^7	*	*	*	58.6	6144	4930	120	**221.8**	**15**
UCOSN-4	1.9×10^7	39262	1	18.1	14.1	8192	2125	30	8.1	10.5
UCOSN-5	9.4×10^8	4.2×10^5	1	253	52.2	65536	12500	37	80	**12.7**
UCOSN-6	4.7×10^{10}	*	*	*	219.3	5.2×10^5	71875	44	**813**	**30.8**
RW-4	1.3×10^9	8369	4	6.48	10.8	5120	67	54	4.40	10.0
RW-5	9.0×10^{10}	54369	4	35.1	15.9	24576	132	60	7.33	10.4
RW-6	5.8×10^{12}	3.5×10^5	4	257	45.2	1.1×10^5	261	66	**12.6**	**10.8**
RW-7	1.5×10^{14}	*	*	*	178	5.2×10^5	518	72	**25.3**	**11.8**
RW-8	*	*	*	*	*	2.4×10^6	1031	78	60.5	14.0
RW-9	*	*	*	*	*	1.7×10^7	2056	84	132	14.5
DPN-3	3.6×10^7	1401	2	.779	-	5832	182	27	.849	-
DPN-4	1.1×10^{10}	16277	2	11.8	10.9	1.0×10^5	1274	34	7.86	9.5
DPN-5	3.2×10^{12}	1.9×10^5	2	197	28.0	1.9×10^6	8918	41	84.6	11.4
DPN-6	9.7×10^{14}	*	*	*	203	3.4×10^7	62426	48	**831**	**26.1**
DPD-9	3.5×10^{22}	11278	1	22.5	12.0	5.2×10^9	13069	46	191	12.2
DPD-10	1.1×10^{25}	38268	1	87.6	17.3	6.2×10^{10}	44493	51	755	18.4

Table 10.1: Experimental results. S_M = maximum # of states; S_R = # of reachable states; I = # of iterations; T = time in seconds; M = memory in MB; time limit = 1500 sec; - indicates negligible value; * indicates out of time; notable figures are highlighted.

message queues, synchronizing access to shared data using various operating system primitives (e.g., semaphores), and cleaning up internal state when a communication fails or times out.

In particular, we analyzed the portion of the IPC protocol that implements the primitives for synchronous communication (approximately 1500 LOC) among multiple threads. With this type of communication, a sender sends a message to a receiver and blocks until an answer is received or it times out. A receiver asks for its next message and blocks until a message is available or it times out. Whenever the receiver gets a synchronous message, it is then expected to send a response to the sender. MAGIC successfully verified the absence of deadlock in this implementation.

The *SSL* benchmark represents a deadlock-free system (approx. 700 LOC) consisting of one OpenSSL server and one OpenSSL client. The *UCOSD-n* benchmarks are derived from μC/OS-II, and consist of n threads executing concurrently. Access to shared data is protected via locks. This implementation suffers from deadlock. In contrast, the *UCOSN-n* benchmarks are deadlock-free. The *RW-n* benchmarks implement a deadlock-free reader-writer system (194 LOC) with n readers, n writers, and a controller. The controller ensures that at most one writer has access to the critical section. Finally, the *DPN-n* benchmarks represent a deadlock-free implementation of n dining philosophers (251 LOC), while *DPD-n* implements n dining philosophers (163 LOC) that can deadlock. As Table 10.1 shows, even though the implementations are of moderate size, the total state space is often quite large due to exponential blowup.

All our experiments were carried out on an AMD Athlon XP 1600+ machine with 1 GB of RAM. Values under **IterDeadlock** refer to measurements for our approach while those under *Plain* correspond to a naive approach involving only predicate abstraction refinement. We note that **IterDeadlock** outperforms *Plain* in almost all cases. In many cases **IterDeadlock** is able to establish deadlock or deadlock freedom while *Plain* runs out of time. Even when both approaches succeed, **IterDeadlock** can yield over 20 times speed-up in time and require over 4 times less memory (*RW-6*). For the experiments involving dining philosophers with deadlock however, *Plain* performs better than **IterDeadlock**. This is because in these cases *Plain* terminates as soon as it discovers a deadlocking scenario, without having to explore the entire state-space. In contrast, **IterDeadlock** has to perform many iterations before finding an actual deadlock.

Chapter 11

Future Directions

The domain of software verification abounds with intriguing and imposing challenges. In this dissertation I have presented a selected few of these problems along with their possible solutions. It is now time to step back and look at the bigger picture. In this chapter, I will attempt to point some significant directions that I was unable to delve into while doing my thesis research. My aim is to answer the question: "What are the next two or three Ph.D. theses to be written in this area" rather than "What would I do if I had another six months to work on this".

Security. One of the most relevant directions is the application of formal techniques to detect security vulnerabilities in software. An important question to ask here is: what do we mean by software security? It appears to me that at some level all security problems are violations of either safety (e.g., buffer overflow) or liveness (e.g., denial of service) requirements. Yet the current mechanisms for specifying safety and liveness properties are inappropriate with respect to non-trivial security claims. Perhaps state/event-based reasoning is an important avenue to investigate.

Another problem is the scalability of formal techniques to large programs running into tens of thousands or even millions of lines of code. There is always a trade-off between scalability and precision. It is possible to model check extremely large programs using a very coarse abstraction such as the control flow graph. On the other hand, coarser abstractions usually lead to numerous spurious counterexamples. It is crucial to find the right balance between precision and scalability and a property-driven approach like CEGAR seems to be quite promising in this regard.

Certification. The notion of proof or certification is essential for the wider applicability of model checking to safety critical systems. A model checker is usually an enormous untrusted computing base and hence of limited credibility if its results is positive, i.e., if it says that a specification holds on a system. The notion of proof carrying code (PCC) [91] aims to increase our confidence in any system analysis. The idea is to generate a proof of correctness of the analysis results which can be checked by a simple trusted proof checker.

The problem with PCC is that the proofs can be quite large even for simple properties. This is similar to the state-space explosion problem in model checking. In addition the original formulation of PCC was restricted to the certification of properties such as memory safety. While considerable progress has been made on limiting proof sizes, it is vital that we extend PCC-like technology to a richer class of specifications.

Learning. Even with a sophisticated approach like compositional CEGAR, one is left at the mercy of state-space explosion during the verification step in the CEGAR loop. It is therefore imperative that our verification step be as efficient as possible. One of the most promising techniques for compositional verification is assume-guarantee (AG) style reasoning. However this approach usually involves the manual construction of appropriate *assumptions* and hence is inherently difficult to apply to large systems.

A very promising development [41] is the use of learning techniques to automatically construct appropriate assumptions in the context of AG-style verification. This approach has yielded encouraging results while verifying safety properties on relatively simple programs. However, its effectiveness on a wide range non-trivial benchmarks is yet to be evaluated. Moreover, the use of learning in the context of non-safety specifications such as liveness and simulation is an area yet to be explored.

I will stop with the above three important directions for future investigation. This list is clearly non-exhaustive and I believe that as far as software analysis is concerned we have simply scratched the surface. The gap between what we can do and what we would like to achieve is quite staggering, and bridging this gap will be one of the foremost problems to concern us in the foreseeable future.

Acknowledgments

We thank Rupak Majumdar and Ranjit Jhala for their help with BLAST.

Bibliography

[1] ABB website. `http://www.abb.com`.

[2] F. Aloul, A. Ramani, I. Markov, and K. Sakallah. PBS: A backtrack search pseudo Boolean solver. In *Proceedings of the 5th International Symposium on the Theory and Applications of Satisfiability Testing (SAT '02)*, pages 346–353, May 2002.

[3] T. S. Anantharaman, Edmund M. Clarke, M. J. Foster, and B. Mishra. Compiling path expressions into VLSI circuits. In *Proceedings of the 12th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '85)*, pages 191–204. ACM Press, January 1985.

[4] Giorgio Ausiello and Giuseppe F. Italiano. On-line algorithms for polynomially solvable satisfiability problems. *Journal of Logic Programming*, 10(1):69–90, January 1991.

[5] Thomas Ball, Rupak Majumdar, Todd D. Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI '01)*, volume 36(5) of *SIGPLAN Notices*, pages 203–213. ACM Press, June 2001.

[6] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In Matthew B. Dwyer, editor, *Proceedings of the 8th International SPIN Workshop on Model Checking of Software (SPIN '01)*, volume 2057 of *Lecture Notes in Computer Science*, pages 103–122. Springer-Verlag, May 2001.

[7] Thomas Ball and Sriram K. Rajamani. Generating abstract explanations of spurious counterexamples in C programs. Technical report MSR-TR-2002-09, Microsoft Research, Redmond, Washington, USA, January 2002.

[8] Bandera website. `http://www.cis.ksu.edu/santos/bandera`.

[9] BEHAVE! website. `http://research.microsoft.com/behave`.

[10] S. Bensalem, Y. Lakhnech, and S. Owre. Computing abstractions of infinite state systems compositionally and automatically. In Alan J. Hu and Moshe Y. Vardi, editors, *Proceedings of the 10th International Conference on Computer Aided Verification (CAV '98)*, volume 1427 of *Lecture Notes in Computer Science*, pages 319–331. Springer-Verlag, June 1998.

[11] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Y. Zue. *Bounded Model Checking*, volume 58 of *Advances in computers*. Academic Press, 2003. To appear.

[12] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Yhu. Symbolic model checking without BDDs. In Rance Cleaveland, editor, *Proceedings of the 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '99)*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer-Verlag, March 1999.

[13] BLAST website. `http://www-cad.eecs.berkeley.edu/~rupak/blast`.

[14] Christie Bolton, Jim Davies, and Jim Woodcock. On the refinement and simulation of data types and processes. In Keijiro Araki, Andy Galloway, and Kenji Taguchi, editors, *Proceedings of the 1st International Conference on Integrated Formal Methods (IFM '99)*, pages 273–292. Springer-Verlag, June 1999.

[15] Business Process Execution Language for Web Services. `http://www.ibm.com/developerworks/library/ws-bpel`.

[16] Julian Bradfield and Colin Stirling. *Modal Logics and Mu-Calculi : An Introduction*, pages 293–330. Handbook of Process Algebra. Elsevier Science Publishers B. V., 2001.

[17] S. D. Brookes and A. W. Roscoe. Deadlock analysis of networks of communicating processes. *Distributed Computing*, 4:209–230, 1991.

[18] M. C. Browne. *Automatic verification of finite state machines using temporal logic*. PhD thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, 1989. Technical report no. CMU-CS-89-117.

[19] T. Bultan. Action Language: A specification language for model checking reactive systems. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE '00)*, pages 335–344. IEEE Computer Society Press, June 2000.

[20] J. Burch. *Trace algebra for automatic verification of real-time concurrent systems*. PhD thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, 1992. Technical report no. CMU-CS-92-179.

[21] J.R. Burch, Edmund M. Clarke, David E. Long, K.L. MacMillan, and David 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.

[22] J.R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science (LICS '90)*, pages 1–33. IEEE Computer Society Press, 1990.

[23] S. Chaki, J. Ouaknine, K. Yorav, and Edmund M. Clarke. Automated compositional abstraction refinement for concurrent C programs: A two-level approach. In *Proceedings of the 2nd Workshop on Software Model Checking (SoftMC '03)*, volume 89(3) of *Electonic Notes in Theoretical Computer Science*, July 2003.

[24] Sagar Chaki, Edmund Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components in C. In *Proceedings of the 25th International Conference on Software Engineering (ICSE '03)*, pages 385–395. IEEE Computer Society Press, May 2003.

[25] Sagar Chaki, Edmund Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components in C. *IEEE Transactions on Software Engineering (TSE)*, 30(6):388–402, June 2004.

[26] Sagar Chaki, Edmund Clarke, Alex Groce, Joël Ouaknine, Ofer Strichman, and Karen Yorav. Efficient verification of sequential and concurrent C programs. *Formal Methods in System Design (FMSD)*, 25(2–3):129–166, September – November 2004.

[27] Sagar Chaki, Edmund Clarke, Joël Ouaknine, and Natasha Sharygina. Automated, compositional and iterative deadlock detection. In *Proceedings of the 2nd ACM-IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE '04)*, pages 201–210. OMNI Press, June 2004.

[28] Sagar Chaki, Edmund M. Clarke, Joël Ouaknine, Natasha Sharygina, and Nishant Sinha. State/event-based software model checking. In Eerke A. Boiten, John Derrick, and Graeme Smith, editors, *Proceedings of the 4th International Conference on Integrated Formal Methods (IFM '04)*, volume 2999 of *Lecture Notes in Computer Science*, pages 128–147. Springer-Verlag, April 2004.

[29] P. Chauhan, Edmund M. Clarke, J. H. Kukula, S. Sapra, Helmut Veith, and Dong Wang. Automated abstraction refinement for model checking large state spaces using SAT based conflict analysis. In Mark Aagaard and John W. O'Leary, editors, *Proceedings of the 4th International Conference on Formal*

Methods in Computer-Aided Design (FMCAD '02), volume 2517 of *Lecture Notes in Computer Science*, pages 33–51. Springer-Verlag, November 2002.

[30] CIL website. `http://manju.cs.berkeley.edu/cil`.

[31] E. M. Clarke. Programming language constructs for which it is impossible to obtain "good" hoare-like axiom systems. In *Proceedings of the 4th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '77)*, pages 10–20. ACM Press, January 1977.

[32] Edmund Clarke and Allen Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In Dexter Kozen, editor, *Proceedings of Workshop on Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer-Verlag, May 1981.

[33] Edmund Clarke, O. Grumberg, M. Talupur, and Dong Wang. Making predicate abstraction efficient: eliminating redundant predicates. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *Cav03*, volume 2725 of *Lecture Notes in Computer Science*, pages 126–140. Springer-Verlag, July 2003.

[34] Edmund Clarke, David Long, and Kenneth McMillan. Compositional model checking. In *Proceedings of the 4th Annual IEEE Symposium on Logic in Computer Science (LICS '89)*, pages 353–362. IEEE Computer Society Press, June 1989.

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

[36] Edmund M. Clarke, O. Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and System (TOPLAS)*, 16(5):1512–1542, September 1994.

[37] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. Allen Emerson and A. Prasad Sistla, editors, *Proceedings of the 12th International Conference on Computer Aided Verification (CAV '00)*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer-Verlag, July 2000.

[38] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM (JACM)*, 50(5):752–794, September 2003.

[39] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking.* MIT Press, Cambridge, MA, 2000.

[40] Edmund M. Clarke, A. Gupta, J. Kukula, and O. Strichman. SAT based abstraction - refinement using ILP and machine learning techniques. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Proceedings of the 14th International Conference on Computer Aided Verification (CAV '02)*, volume 2404 of *Lecture Notes in Computer Science*, pages 265–279. Springer-Verlag, July 2002.

[41] J. M. Cobleigh, Dimitra Giannakopoulou, and Corina S. Păsăreanu. Learning assumptions for compositional verification. In Hubert Garavel and John Hatcliff, editors, *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '03)*, volume 2619 of *Lecture Notes in Computer Science*, pages 331–346. Springer-Verlag, April 2003.

[42] M. Colón and T. E. Uribe. Generating finite-state abstractions of reactive systems using decision procedures. In Alan J. Hu and Moshe Y. Vardi, editors, *Proceedings of the 10th International Conference on Computer Aided Verification (CAV '98)*, volume 1427 of *Lecture Notes in Computer Science*, pages 293–304. Springer-Verlag, June 1998.

[43] S. A. Cook. Axiomatic and interpretative semantics for an Algol fragment. Technical Report 79, Department of Computer Science, University of Toronto, Toronto, Canada, 1975. to be published in SCICOMP.

[44] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng. Bandera: extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE '00)*, pages 439–448. IEEE Computer Society Press, June 2000.

[45] James C. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Transactions on Software Engineering (TSE)*, 22(3):161–180, March 1996.

[46] Costas Courcoubetis, Moshe Y. Vardi, Pierre Wolper, and Mihalis Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in System Design (FMSD)*, 1(2–3):275–288, October 1992.

[47] 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 ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '77)*, pages 238–252. ACM Press, January 1977.

[48] Dennis Dams and Kedar S. Namjoshi. Shape analysis through predicate abstraction and model checking. In Lenore D. Zuck, Paul C. Attie,

Agostino Cortesi, and Supratik Mukhopadhyay, editors, *Proceedings of the 4th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI '03)*, volume 2575 of *Lecture Notes in Computer Science*, pages 310–324. Springer-Verlag, January 2003.

[49] Satyaki Das and David L. Dill. Counter-example based predicate discovery in predicate abstraction. In Mark Aagaard and John W. O'Leary, editors, *Proceedings of the 4th International Conference on Formal Methods in Computer-Aided Design (FMCAD '02)*, volume 2517 of *Lecture Notes in Computer Science*, pages 19–32. Springer-Verlag, November 2002.

[50] Satyaki Das, David L. Dill, and Seungjoon Park. Experience with predicate abstraction. In Nicolas Halbwachs and Doron Peled, editors, *Proceedings of the 11th International Conference on Computer Aided Verification (CAV '99)*, volume 1633 of *Lecture Notes in Computer Science*, pages 160–171. Springer-Verlag, July 1999.

[51] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM (CACM)*, 5(7):394–397, July 1962.

[52] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM (JACM)*, 7(3):201–215, June 1960.

[53] C. Demartini, R. Iosif, and R. Sisto. A deadlock detection tool for concurrent java programs. *Software - Practice and Experience (SPE)*, 29(7):577–603, June 1999.

[54] Edsger Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[55] David L. Dill. *Trace theory for automatic hierarchical verification of speed-independent circuits*. PhD thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, 1988. Technical report no. CMU-CS-88-119.

[56] David L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. MIT Press, 1989.

[57] W. F. Dowling and J. H. Gallier. Linear time algorithms for testing the satisfiability of propositional horn formula. *Journal of Logic Programming*, 1(3):267–284, October 1984.

[58] Matthew B. Dwyer, John Hatcliff, Roby Joehanes, Shawn Laubach, Corina S. Pasareanu, Hongjun Zheng, and Willem Visser. Tool-supported program abstraction for finite-state verification. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE '01)*, pages 177–187. IEEE Computer Society Press, May 2001.

[59] E. Allen Emerson. *Temporal and modal logic*, volume B: formal models and semantics of *Handbook of Theoretical Computer Science*, pages 995–1072. MIT Press, 1990.

[60] Formal Systems (Europe) Ltd. website. `http://www.fsel.com`.

[61] R. Gerth, Doron Peled, Moshe Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In Piotr Dembinski and Marek Sredniawa, editors, *Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification XV*, volume 38 of *IFIP Conference Proceedings*, pages 3–18. Chapman & Hall, Ltd., June 1996.

[62] Dimitra Giannakopoulou and J. Magee. Fluent model checking for event-based systems. In *Proceedings of the 11th ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE '03)*, pages 257–266. ACM Press, September 2003.

[63] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In Orna Grumberg, editor, *Proceedings of the 9th International Conference on Computer Aided Verification (CAV '97)*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer-Verlag, June 1997.

[64] Orna Grumberg and David E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and System (TOPLAS)*, 16(3):843–871, May 1994.

[65] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Shaz Qadeer. Thread-modular abstraction refinement. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *Proceedings of the 15th International Conference on Computer Aided Verification (CAV '03)*, volume 2725 of *Lecture Notes in Computer Science*, pages 262–274. Springer-Verlag, July 2003.

[66] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '02)*, volume 37(1) of *SIGPLAN Notices*, pages 58–70. ACM Press, January 2002.

[67] Thomas A. Henzinger, Shaz Qadeer, and Sriram K. Rajamani. Decomposing refinement proofs using assume-guarantee reasoning. In *Proceedings of the 2000 International Conference on Computer-Aided Design (ICCAD '00)*, pages 245–252. IEEE Computer Society Press, November 2000.

[68] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM (CACM)*, 21(8):666–677, August 1978.

[69] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, London, 1985.

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

[71] G.J. Holzmann, Doron Peled, and M. Yannakakis. On nested depth first search. In *Proceedings of the 2nd International SPIN Workshop on Model Checking of Software (SPIN '96)*, pages 81–89, August 1996.

[72] M. Huth, R. Jagadeesan, and D. Schmidt. Modal transition systems: A foundation for three-valued program analysis. In David Sands, editor, *Proceedings of the 10th European Symposium On Programming (ESOP '01)*, volume 2028 of *Lecture Notes in Computer Science*, pages 155–169. Springer-Verlag, April 2001.

[73] Ekkart Kindler and Tobias Vesper. ESTL: A temporal logic for events and states. In J. Desel and M. Silva, editors, *Proceedings of the 19th International Conference on Application and Theory of Petri Nets (ICATPN '98)*, volume 1420 of *Lecture Notes in Computer Science*, pages 365–383. Springer-Verlag, June 1998.

[74] Dexter Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science (TCS)*, 27(3):333–354, December 1983.

[75] R. P. Kurshan. Analysis of discrete event coordination. In J. W. de Bakker, Willem P. de Roever, and Grzegorz Rozenberg, editors, *Proceedings of Stepwise Refinement of Distributed Systems, Models, Formalisms, Correctness, REX Workshop*, volume 430 of *Lecture Notes in Computer Science*, pages 414–453. Springer-Verlag, May–June 1989.

[76] Robert P. Kurshan. *Computer-aided verification of coordinating processes: the automata-theoretic approach.* Princeton University Press, 1994.

[77] Y. Lakhnech, S. Bensalem, S. Berezin, and S. Owre. Incremental verification by abstraction. In Tiziana Margaria and Wang Yi, editors, *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '01)*, volume 2031 of *Lecture Notes in Computer Science*, pages 98–112. Springer-Verlag, April 2001.

[78] O. Lichtenstein and A. Pnueli. Checking that finite-state concurrent programs satisfy their linear specification. In *Proceedings of the 12th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '85)*, pages 97–107. ACM Press, January 1985.

[79] J. Magee and J. Kramer. *Concurrency: State Models & Java Programs.* Wiley.

[80] MAGIC website. `http://www.cs.cmu.edu/~chaki/magic`.

[81] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag, New York, 1992.

[82] J. M. R. Martin and Y. Huddart. Parallel algorithms for deadlock and livelock analysis of concurrent systems. In *Proceedings of Communicating Process Architectures*, volume 58 of *Concurrent Systems Engineering.* IOS Press, September 2000.

[83] J. M. R. Martin and S. Jassim. A tool for proving deadlock freedom. In *Proceedings of the 20th World Occam and Transputer User Group Technical Meeting.* IOS Press, April 1997.

[84] Kenneth L. McMillan. A compositional rule for hardware design refinement. In Orna Grumberg, editor, *Proceedings of the 9th International Conference on Computer Aided Verification (CAV '97)*, volume 1254 of *Lecture Notes in Computer Science*, pages 24–35. Springer-Verlag, June 1997.

[85] Robin Milner. *Communication and Concurrency.* Prentice-Hall International, London, 1989.

[86] Robin Milner. *Communicating and Mobile Systems: the π-Calculus.* Cambridge University Press, 1999.

[87] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th ACM IEEE Design Automation Conference (DAC '01)*, pages 530–535. ACM Press, June 2001.

[88] Glenford J. Myers. *Art of Software Testing.* John Wiley & Sons, Inc., 1979.

[89] Kedar S. Namjoshi and Robert P. Kurshan. Syntactic program transformations for automatic abstraction. In E. Allen Emerson and A. Prasad Sistla, editors, *Proceedings of the 12th International Conference on Computer Aided Verification (CAV '00)*, volume 1855 of *Lecture Notes in Computer Science*, pages 435–449. Springer-Verlag, July 2000.

[90] G. Naumovich, L. A. Clarke, L. J. Osterweil, and Matthew B. Dwyer. Verification of concurrent software with FLAVERS. In *Proceedings of the 19th International Conference on Software Engineering (ICSE '97)*, pages 594–595. ACM Press, May 1997.

[91] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97)*, pages 106–119. ACM Press, January 1997.

[92] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In R. Nigel Horspool, editor, *Proceedings of the 11th International Conference on Compiler Construction (CC '02)*, volume 2304 of *Lecture Notes in Computer Science*, pages 213–228. Springer-Verlag, April 2002.

[93] R. De Nicola, A. Fantechi, S. Gnesi, and G. Ristori. An action-based framework for verifying logical and behavioural properties of concurrent systems. *Computer Networks and ISDN Systems*, 25(7):761–778, February 1993.

[94] R. De Nicola and F. Vaandrager. Three logics for branching bisimulation. *Journal of the ACM (JACM)*, 42(2):458–487, March 1995.

[95] OpenSSL website. `http://www.openssl.org`.

[96] R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SIAM Journal of Computing (SIAMJC)*, 16(6):973–989, December 1987.

[97] Amir Pnueli. Application of temporal logic to the specification and verification of reactive systems: A survey of current trends. In J.W. de Bakker, W. P. de Roever, and G. Rozenburg, editors, *Current Trends in Concurrency*, volume 224 of *Lecture Notes in Computer Science*, pages 510–584. Springer-Verlag, 1986.

[98] Corina S. Păsăreanu, Matthew B. Dwyer, and Willem Visser. Finding feasible counter-examples when model checking abstracted Java programs. In Tiziana Margaria and Wang Yi, editors, *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '01)*, volume 2031 of *Lecture Notes in Computer Science*, pages 284–298. Springer-Verlag, April 2001.

[99] J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In M. Dezani and U. Montanari, editors, *Proceedings of the 5th Colloquium of the International Symposium on Programming (ISP '82)*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer-Verlag, April 1982.

[100] A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice-Hall International, London, 1997.

[101] A. W. Roscoe and N. Dathi. The pursuit of deadlock freedom. *Information and Computation*, 75(3):289–327, December 1987.

[102] Vlad Rusu and Eli Singerman. On proving safety properties by integrating static analysis, theorem proving and abstraction. In Rance Cleaveland, editor, *Proceedings of the 5th International Conference on Tools and Algorithms for the*

Construction and Analysis of Systems (TACAS '99), volume 1579 of *Lecture Notes in Computer Science*, pages 178–192. Springer-Verlag, March 1999.

[103] S. K. Shukla. *Uniform Approaches to the Verification of Finite State Systems.* PhD thesis, State University of New York, Albany, New York, USA, 1997.

[104] Joao P. Marques Silva and Karem A. Sakallah. GRASP - a new search algorithm for satisfiability. In *Proceedings of the 1996 International Conference on Computer-Aided Design (ICCAD '96)*, pages 220–227. IEEE Computer Society Press, November 1996.

[105] Joao P. Marques Silva and Karem A. Sakallah. GRASP - a new search algorithm for satisfiability. Technical report CSE-TR-292-96, University of Michigan, Ann Arbor, Michigan, USA, April 1996.

[106] Joao P. Marques Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, May 1999.

[107] SLAM website. `http://research.microsoft.com/slam`.

[108] Fabio Somenzi and R. Bloem. Efficient Büchi automata from LTL formulae. In E. Allen Emerson and A. Prasad Sistla, editors, *Proceedings of the 12th International Conference on Computer Aided Verification (CAV '00)*, volume 1855 of *Lecture Notes in Computer Science*, pages 248–263. Springer-Verlag, July 2000.

[109] SSL 3.0 Specification. `http://wp.netscape.com/eng/ssl3`.

[110] Colin Stirling. The Joys of Bisimulation. In Lubos Brim, Jozef Gruska, and Jir Zlatuska, editors, *Proceedings of the 23rd International Symposium on Mathematical Foundations of Computer Science (MFCS '98)*, volume 1450 of *Lecture Notes in Computer Science*, pages 142–151. Springer-Verlag, August 1998.

[111] Scott D. Stoller. Model-checking multi-threaded distributed Java programs. In Klaus Havelund, John Penix, and Willem Visser, editors, *Proceedings of the 7th International SPIN Workshop on Model Checking of Software (SPIN '00)*, volume 1885 of *Lecture Notes in Computer Science*, pages 224–244. Springer-Verlag, August–September 2000.

[112] Scott D. Stoller. Model-checking multi-threaded distributed Java programs. *International Journal on Software Tools for Technology Transfer (STTT)*, 4(1):71–91, October 2002.

[113] Wring website. `http://vlsi.colorado.edu/~rbloem/wring.html`.

[114] H. Zhang. SATO: An efficient propositional prover. In William McCune, editor, *Proceedings of the 14th International Conference on Automated Deduction (CADE '97)*, volume 1249 of *Lecture Notes in Computer Science*, pages 272–275. Springer-Verlag, July 1997.

Appendix A

OpenSSL Example

In this chapter we describe the OpenSSL benchmark used in our experimental evaluation of the CEGAR framework for simulation conformance (cf. Section 6.5). Recall that the benchmark consists of two components - one for the server and the other for the client. We will first present the server and then the client. We begin with the server source code. Essentially, the source consists of a core C procedure `ssl3_accept`. The procedure simulates a state machine via a top-level **while** loop and a variable `s->state` to keep track of the current state of the machine. The complete source code for `ssl3_accept` is presented next.

A.1 Server Source

```
int ssl3_accept(SSL *s )
{
  BUF_MEM *buf ;
  unsigned long l ;
  unsigned long Time ;
  unsigned long tmp ;
  void (*cb)() ;
```

```
long num1 ;
int ret ;
int new_state ;
int state ;
int skip ;
int got_new_session ;
int *tmp___0 ;
int tmp___1 ;
int tmp___2 ;
int tmp___3 ;
int tmp___4 ;
int tmp___5 ;
int tmp___6 ;
int tmp___7 ;
long tmp___8 ;
int tmp___9 ;
int tmp___10 ;

tmp = (unsigned long )time((time_t *)((void *)0));
Time = tmp;
cb = (void (*)())((void *)0);
ret = -1;
skip = 0;
got_new_session = 0;
RAND_add((void const *)(& Time), (int )sizeof(Time), (double )0);
ERR_clear_error();
tmp___0 = __errno_location();
(*tmp___0) = 0;
if ((unsigned long )s->info_callback != (unsigned long )((void *)0)) {
  cb = s->info_callback;
} else {
  if ((unsigned long )(s->ctx)->info_callback != (unsigned long )((void *)0)) {
    cb = (s->ctx)->info_callback;
  }
}
s->in_handshake ++;
tmp___1 = SSL_state(s);
if (tmp___1 & 12288) {
  tmp___2 = SSL_state(s);
  if (tmp___2 & 16384) SSL_clear(s);
} else SSL_clear(s);
if ((unsigned long )s->cert == (unsigned long )((void *)0)) {
    ERR_put_error(20, 128, 179, (char const *)"s3_srvr.c", 187);
    return (-1);
  }
  while (1) {
    state = s->state;
    switch (s->state) {
    case 12292:
    s->new_session = 1;
    case 16384: ;
    case 8192: ;
    case 24576: ;
    case 8195:
    s->server = 1;
    if ((unsigned long )cb != (unsigned long )((void *)0)) {
      ((*cb))(s, 16, 1);
    }
    if (s->version >> 8 != 3) {
      ERR_put_error(20, 128, 157, (char const *)"s3_srvr.c", 211);
      return (-1);
    }
    s->type = 8192;
    if ((unsigned long )s->init_buf == (unsigned long )((void *)0)) {
      buf = BUF_MEM_new();
      if ((unsigned long )buf == (unsigned long )((void *)0)) {
        ret = -1;
        goto end;
      }
      tmp___3 = BUF_MEM_grow(buf, 16384);
      if (! tmp___3) {
        ret = -1;
        goto end;
      }
      s->init_buf = buf;
    }
    tmp___4 = ssl3_setup_buffers(s);
    if (! tmp___4) {
      ret = -1;
      goto end;
    }
    s->init_num = 0;
    if (s->state != 12292) {
      tmp___5 = ssl_init_wbio_buffer(s, 1);
  if (! tmp___5) {
    ret = -1;
    goto end;
  }
  ssl3_init_finished_mac(s);
  s->state = 8464;
  (s->ctx)->stats.sess_accept ++;
} else {
  (s->ctx)->stats.sess_accept_renegotiate ++;
  s->state = 8480;
}
break;
case 8480: ;
case 8481:
s->shutdown = 0;
ret = ssl3_send_hello_request(s);
if (ret <= 0) {
  goto end;
}
(s->s3)->tmp.next_state = 8482;
s->state = 8448;
s->init_num = 0;
ssl3_init_finished_mac(s);
break;
case 8482:
s->state = 3;
break;
case 8464: ;
case 8465: ;
case 8466:
s->shutdown = 0;
ret = ssl3_get_client_hello(s);
if (ret <= 0) {
  goto end;
}
got_new_session = 1;
s->state = 8496;
s->init_num = 0;
break;
case 8496: ;
case 8497:
ret = ssl3_send_server_hello(s);
```

```
if (ret <= 0) {
  goto end;
}
if (s->hit) {
  s->state = 8656;
} else {
  s->state = 8512;
}
s->init_num = 0;
break;
case 8512: ;
case 8513: ;
if (((s->s3)->tmp.new_cipher)->algorithms & 256UL) {
  skip = 1;
} else {
  ret = ssl3_send_server_certificate(s);
  if (ret <= 0) {
    goto end;
  }
}
s->state = 8528;
s->init_num = 0;
break;
case 8528: ;
case 8529:
l = ((s->s3)->tmp.new_cipher)->algorithms;
if (s->options & 2097152UL) {
  (s->s3)->tmp.use_rsa_tmp = 1;
} else {
  (s->s3)->tmp.use_rsa_tmp = 0;
}
if ((s->s3)->tmp.use_rsa_tmp) {
  goto _L___0;
} else {
  if (l & 30UL) {
    goto _L___0;
  } else {
    if (l & 1UL) {
      if ((unsigned long )(s->cert)->pkeys[0].privatekey ==
          (unsigned long )((void *)0)) {
        goto _L___0;
      } else {
        if (((s->s3)->tmp.new_cipher)->algo_strength & 2UL) {
          tmp___6 = EVP_PKEY_size((s->cert)->pkeys[0].privatekey);
          if (((s->s3)->tmp.new_cipher)->algo_strength & 4UL) {
            tmp___7 = 512;
          } else {
            tmp___7 = 1024;
          }
          if (tmp___6 * 8 > tmp___7) {
            _L___0:
            _L:
            ret = ssl3_send_server_key_exchange(s);
            if (ret <= 0) {
              goto end;
            }
          } else {
            skip = 1;
          }
        } else {
          skip = 1;
        }
      }
    } else {
      skip = 1;
    }
  }
}
s->state = 8544;
s->init_num = 0;
break;
case 8544: ;
case 8545: ;
if (s->verify_mode & 1) {
  if ((unsigned long )(s->session)->peer != (unsigned long )((void *)0)) {
    if (s->verify_mode & 4) {
      skip = 1;
      (s->s3)->tmp.cert_request = 0;
      s->state = 8560;
    } else {
      goto _L___2;
    }
  } else {
    _L___2:
```

```
    if (((s->s3)->tmp.new_cipher)->algorithms & 256UL) {
      if (s->verify_mode & 2) {
        goto _L___1;
      } else {
        skip = 1;
        (s->s3)->tmp.cert_request = 0;
        s->state = 8560;
      }
    } else {
      _L___1:
      (s->s3)->tmp.cert_request = 1;
      ret = ssl3_send_certificate_request(s);
      if (ret <= 0) {
        goto end;
      }
      s->state = 8448;
      (s->s3)->tmp.next_state = 8576;
      s->init_num = 0;
    }
  }
} else {
  skip = 1;
  (s->s3)->tmp.cert_request = 0;
  s->state = 8560;
}
break;
case 8560: ;
case 8561:
ret = ssl3_send_server_done(s);
if (ret <= 0) {
  goto end;
}
(s->s3)->tmp.next_state = 8576;
s->state = 8448;
s->init_num = 0;
break;
case 8448:
num1 = BIO_ctrl(s->wbio, 3, 0L, (void *)0);
if (num1 > 0L) {
  s->rwstate = 2;
  tmp___8 = BIO_ctrl(s->wbio, 11, 0L, (void *)0);
  num1 = (long )((int )tmp___8);
  if (num1 <= 0L) {
    ret = -1;
    goto end;
  }
  s->rwstate = 1;
}
s->state = (s->s3)->tmp.next_state;
break;
case 8576: ;
case 8577:
ret = ssl3_check_client_hello(s);
if (ret <= 0) {
  goto end;
}
if (ret == 2) {
  s->state = 8466;
} else {
  ret = ssl3_get_client_certificate(s);
  if (ret <= 0) {
    goto end;
  }
  s->init_num = 0;
  s->state = 8592;
}
break;
case 8592: ;
case 8593:
ret = ssl3_get_client_key_exchange(s);
if (ret <= 0) {
  goto end;
}
s->state = 8608;
s->init_num = 0;
((*(((s->method)->ssl3_enc)->cert_verify_mac)))
    (s, & (s->s3)->finish_dgst1, & (s->s3)->tmp.cert_verify_md[0]);
((*(((s->method)->ssl3_enc)->cert_verify_mac)))
    (s, & (s->s3)->finish_dgst2, & (s->s3)->tmp.cert_verify_md[16]);
break;
case 8608: ;
case 8609:
ret = ssl3_get_cert_verify(s);
if (ret <= 0) {
  goto end;
}
s->state = 8640;
s->init_num = 0;
break;
case 8640: ;
case 8641:
ret = ssl3_get_finished(s, 8640, 8641);
if (ret <= 0) {
  goto end;
}
if (s->hit) {
  s->state = 3;
} else {
  s->state = 8656;
}
s->init_num = 0;
break;
case 8656: ;
case 8657:
(s->session)->cipher = (s->s3)->tmp.new_cipher;
tmp___9 = ((*(((s->method)->ssl3_enc)->setup_key_block)))(s);
if (! tmp___9) {
  ret = -1;
  goto end;
}
ret = ssl3_send_change_cipher_spec(s, 8656, 8657);
if (ret <= 0) {
  goto end;
}
s->state = 8672;
s->init_num = 0;
tmp___10 = ((*(((s->method)->ssl3_enc)->change_cipher_state)))(s, 34);
if (! tmp___10) {
  ret = -1;
  goto end;
}
break;
case 8672: ;
case 8673:
ret = ssl3_send_finished(s, 8672, 8673,
            ((s->method)->ssl3_enc)->server_finished_label,
            ((s->method)->ssl3_enc)->server_finished_label_len);
if (ret <= 0) {
  goto end;
}
s->state = 8448;
if (s->hit) {
  (s->s3)->tmp.next_state = 8640;
} else {
  (s->s3)->tmp.next_state = 3;
}
s->init_num = 0;
break;
case 3:
ssl3_cleanup_key_block(s);
BUF_MEM_free(s->init_buf);
s->init_buf = (BUF_MEM *)((void *)0);
ssl_free_wbio_buffer(s);
s->init_num = 0;
if (got_new_session) {
  s->new_session = 0;
  ssl_update_cache(s, 2);
  (s->ctx)->stats.sess_accept_good ++;
  s->handshake_func = (int (*)())(& ssl3_accept);
  if ((unsigned long )cb != (unsigned long )((void *)0)) {
    ((*cb))(s, 32, 1);
  }
} ret = 1;
goto end;
default:
ERR_put_error(20, 128, 255, (char const *)"s3_srvr.c", 536);
ret = -1;
goto end;
}
if (! (s->s3)->tmp.reuse_message) {
  if (! skip) {
    if (s->debug) {
      ret = (int )BIO_ctrl(s->wbio, 11, 0L, (void *)0);
      if (ret <= 0) {
        goto end;
      }
    }
    if ((unsigned long )cb != (unsigned long )((void *)0)) {
          if (s->state != state) {
            new_state = s->state;
            s->state = state;
            ((*cb))(s, 8193, 1);
            s->state = new_state;
          }
        }
      }
    }
    skip = 0;
  }
  end:
  s->in_handshake --;
  if ((unsigned long )cb != (unsigned long )((void *)0)) {
    ((*cb))(s, 8194, ret);
  }
  return (ret);
}
```

A.2 Server Library Specifications

As discussed earlier, the essential idea behind MAGIC is to model each library routine call by an EFSM. For example, the call to `ssl3_send_hello_request` is modeled by an EFSM `SendHelloRequest` which either does the action `send_hello_request` and then returns the value 1 or simply returns -1. In MAGIC one can specify this information via the following syntax.

```
cproc ssl3_send_hello_request {
    abstract {ssl3_send_hello_request_abs,1,SendHelloRequest};
}

SendHelloRequest =
(
  send_hello_request -> return {$0 == 1} -> STOP |
  return {$0 == -1} -> STOP
).
```

The complete input to MAGIC for all the library routines invoked by `ssl3_accept` and their corresponding EFSMs is as follows:

```
cproc ssl3_send_hello_request {
    abstract {ssl3_send_hello_request_abs,1,SendHelloRequest};
}

cproc ssl3_get_client_hello {
    abstract {ssl3_get_client_hello_abs,1,GetClientHello};
}

cproc ssl3_send_server_hello {
    abstract {ssl3_send_server_hello_abs,1,SendServerHello};
}

cproc ssl3_send_server_certificate {
    abstract {ssl3_send_server_certificate_abs,1,SendServerCertificate};
}

cproc ssl3_send_server_key_exchange {
    abstract {ssl3_send_server_key_exchange_abs,1,SendServerKeyExchange};
}

cproc ssl3_send_certificate_request {
    abstract {ssl3_send_certificate_request_abs,1,SendCertificateRequest};
}

cproc ssl3_send_server_done {
    abstract {ssl3_send_server_done_abs,1,SendServerDone};
}

cproc ssl3_check_client_hello {
    abstract {ssl3_check_client_hello_abs,1,CheckClientHello};
}

cproc ssl3_get_client_certificate {
    abstract {ssl3_get_client_certificate_abs,1,GetClientCertificate};
}

cproc ssl3_get_client_key_exchange {
    abstract {ssl3_get_client_key_exchange_abs,1,GetClientKeyExchange};
}
```

```
cproc ssl3_get_cert_verify {
    abstract {ssl3_get_cert_verify_abs,1,GetCertVerify};
}

cproc ssl3_get_finished {
    abstract {ssl3_get_finished_abs,1,GetFinished};
}

cproc ssl3_send_change_cipher_spec {
    abstract {ssl3_send_change_cipher_spec_abs,1,SendChangeCipherSpec};
}

cproc ssl3_send_finished {
    abstract {ssl3_send_finished_abs,1,SendFinished};
}

SendHelloRequest =
(
  send_hello_request -> return {$0 == 1} -> STOP |
  return {$0 == -1} -> STOP
).

GetClientHello =
(
  exch_client_hello -> return {$0 == 1} -> STOP |
  return {$0 == -1} -> STOP
).

SendServerHello =
(
  exch_server_hello -> return {$0 == 1} -> STOP |
  return {$0 == -1} -> STOP
).

SendServerCertificate =
(
  exch_server_certificate -> return {$0 == 1} -> STOP |
  return {$0 == -1} -> STOP
).

SendChangeCipherSpec =
(
  send_change_cipher_spec -> return {$0 == 1} -> STOP |
  return {$0 == -1} -> STOP
).

GetClientKeyExchange =
(
  key_exchange_clnt_to_srvr -> return {$0 == 1} -> STOP |
  return {$0 == -1} -> STOP
).

SendServerKeyExchange =
(
  key_exchange_srvr_to_clnt -> return {$0 == 1} -> STOP |
  return {$0 == -1} -> STOP
).

SendFinished =
(
  finished_srvr_to_clnt -> return {$0 == 1} -> STOP |
  return {$0 == -1} -> STOP
).

SendCertificateRequest =
(
  certificate_request_srvr_to_clnt -> return {$0 == 1} -> STOP |
  return {$0 == -1} -> STOP
).

CheckClientHello =
(
  check_client_hello -> return {$0 == 1} -> STOP |
  return {$0 == -1} -> STOP
).

GetClientCertificate =
(
  exch_client_certificate -> return {$0 == 1} -> STOP |
  return {$0 == -1} -> STOP
).

GetFinished =
(
  finished_clnt_to_srvr -> return {$0 == 1} -> STOP |
  return {$0 == -1} -> STOP
).

GetCertVerify =
(
  cert_verify_clnt_to_srvr -> return {$0 == 1} -> STOP |
  return {$0 == -1} -> STOP
).

SendServerDone =
(
  send_server_done -> return {$0 == 1} -> STOP |
  return {$0 == -1} -> STOP
).
```

A.3 Client Source

We have presented the input to MAGIC as far as the server is concerned. We now present the client component beginning with the source code. The client consists of a core procedure called `ssl3_connect` whose structure is very similar to `ssl3_accept`. In particular, it also simulates a state machine via a top-level **while** loop and a variable `s->state` to keep track of the current state of the machine. The complete source code for `ssl3_connect` is presented next.

```
int ssl3_connect(SSL *s )
{
  BUF_MEM *buf ;
  unsigned long Time ;
  unsigned long tmp ;
  unsigned long l ;
  long num1 ;
  void (*cb)() ;
  int ret ;
  int new_state ;
int state ;
int skip ;
int *tmp___0 ;
int tmp___1 ;
int tmp___2 ;
int tmp___3 ;
int tmp___4 ;
int tmp___5 ;
int tmp___6 ;
int tmp___7 ;
int tmp___8 ;
long tmp___9 ;

tmp = (unsigned long )time((time_t *)((void *)0));
Time = tmp;
cb = (void (*)())((void *)0);
ret = -1;
skip = 0;
RAND_add((void const *)(& Time), (int )sizeof(Time), (double )0);
ERR_clear_error();
tmp___0 = __errno_location();
(*tmp___0) = 0;
if ((unsigned long )s->info_callback != (unsigned long )((void *)0)) {
  cb = s->info_callback;
} else {
  if ((unsigned long )(s->ctx)->info_callback !=
      (unsigned long )((void *)0)) {
    cb = (s->ctx)->info_callback;
  }
}
s->in_handshake ++;
tmp___1 = SSL_state(s);
if (tmp___1 & 12288) {
  tmp___2 = SSL_state(s);
  if (tmp___2 & 16384) {
    SSL_clear(s);
  }
} else {
  SSL_clear(s);
}
while (1) {
  state = s->state;
```

```
switch (s->state) {
case 12292:
s->new_session = 1;
s->state = 4096;
(s->ctx)->stats.sess_connect_renegotiate ++;
case 16384: ;
case 4096: ;
case 20480: ;
case 4099:
s->server = 0;
if ((unsigned long )cb != (unsigned long )((void *)0)) {
  ((*cb))(s, 16, 1);
}
if ((s->version & 65280) != 768) {
  ERR_put_error(20, 132, 157, (char const *)"s3_clnt.c", 146);
  ret = -1;
  goto end;
}
s->type = 4096;
if ((unsigned long )s->init_buf == (unsigned long )((void *)0)) {
  buf = BUF_MEM_new();
  if ((unsigned long )buf == (unsigned long )((void *)0)) {
    ret = -1;
    goto end;
  }
  tmp___3 = BUF_MEM_grow(buf, 16384);
  if (! tmp___3) {
    ret = -1;
    goto end;
  }
  s->init_buf = buf;
}
tmp___4 = ssl3_setup_buffers(s);
if (! tmp___4) {
  ret = -1;
  goto end;
}
tmp___5 = ssl_init_wbio_buffer(s, 0);
if (! tmp___5) {
  ret = -1;
  goto end;
}
```

```
ssl3_init_finished_mac(s);
s->state = 4368;
(s->ctx)->stats.sess_connect ++;
s->init_num = 0;
break;
case 4368: ;
case 4369:
s->shutdown = 0;
ret = ssl3_client_hello(s);
if (ret <= 0) {
  goto end;
}
s->state = 4384;
s->init_num = 0;
if ((unsigned long )s->bbio != (unsigned long )s->wbio) {
  s->wbio = BIO_push(s->bbio, s->wbio);
}
break;
case 4384: ;
case 4385:
ret = ssl3_get_server_hello(s);
if (ret <= 0) {
  goto end;
}
if (s->hit) {
  s->state = 4560;
} else {
  s->state = 4400;
}
s->init_num = 0;
break;
case 4400: ;
case 4401: ;
if (((s->s3)->tmp.new_cipher)->algorithms & 256UL) {
  skip = 1;
} else {
  ret = ssl3_get_server_certificate(s);
  if (ret <= 0) {
    goto end;
  }
}
s->state = 4416;
```

```
s->init_num = 0;
break;
case 4416: ;
case 4417:
ret = ssl3_get_key_exchange(s);
if (ret <= 0) {
  goto end;
}
s->state = 4432;
s->init_num = 0;
tmp___6 = ssl3_check_cert_and_algorithm(s);
if (! tmp___6) {
  ret = -1;
  goto end;
}
break;
case 4432: ;
case 4433:
ret = ssl3_get_certificate_request(s);
if (ret <= 0) {
  goto end;
}
s->state = 4448;
s->init_num = 0;
break;
case 4448: ;
case 4449:
ret = ssl3_get_server_done(s);
if (ret <= 0) {
  goto end;
}
if ((s->s3)->tmp.cert_req) {
  s->state = 4464;
} else {
  s->state = 4480;
}
s->init_num = 0;
break;
case 4464: ;
case 4465: ;
case 4466: ;
case 4467:
```

```
ret = ssl3_send_client_certificate(s);
if (ret <= 0) {
  goto end;
}
s->state = 4480;
s->init_num = 0;
break;
case 4480: ;
case 4481:
ret = ssl3_send_client_key_exchange(s);
if (ret <= 0) {
  goto end;
}
l = ((s->s3)->tmp.new_cipher)->algorithms;
if ((s->s3)->tmp.cert_req == 1) {
  s->state = 4496;
} else {
  s->state = 4512;
  (s->s3)->change_cipher_spec = 0;
}
s->init_num = 0;
break;
case 4496: ;
case 4497:
ret = ssl3_send_client_verify(s);
if (ret <= 0) {
  goto end;
}
s->state = 4512;
s->init_num = 0;
(s->s3)->change_cipher_spec = 0;
break;
case 4512: ;
case 4513:
ret = ssl3_send_change_cipher_spec(s, 4512, 4513);
if (ret <= 0) {
  goto end;
}
s->state = 4528;
s->init_num = 0;
(s->session)->cipher = (s->s3)->tmp.new_cipher;
if ((unsigned long )(s->s3)->tmp.new_compression ==
    (unsigned long )((void *)0)) {
  (s->session)->compress_meth = 0;
} else {
  (s->session)->compress_meth = ((s->s3)->tmp.new_compression)->id;
}
tmp___7 = ((*(((s->method)->ssl3_enc)->setup_key_block)))(s);
if (! tmp___7) {
  ret = -1;
  goto end;
}
tmp___8 = ((*(((s->method)->ssl3_enc)->change_cipher_state)))(s, 18);
if (! tmp___8) {
  ret = -1;
  goto end;
}
break;
case 4528: ;
case 4529:
ret = ssl3_send_finished(s, 4528, 4529,
    ((s->method)->ssl3_enc)->client_finished_label,
    ((s->method)->ssl3_enc)->client_finished_label_len);
if (ret <= 0) {
  goto end;
}
s->state = 4352;
(s->s3)->flags &= -5L;
if (s->hit) {
  (s->s3)->tmp.next_state = 3;
  if ((s->s3)->flags & 2L) {
    s->state = 3;
    (s->s3)->flags |= 4L;
    (s->s3)->delay_buf_pop_ret = 0;
  }
} else {
  (s->s3)->tmp.next_state = 4560;
}
s->init_num = 0;
break;
case 4560: ;
case 4561:
ret = ssl3_get_finished(s, 4560, 4561);
if (ret <= 0) {
  goto end;
}
if (s->hit) {
  s->state = 4512;
} else {
  s->state = 3;
}
s->init_num = 0;
break;
case 4352:
num1 = BIO_ctrl(s->wbio, 3, 0L, (void *)0);
if (num1 > 0L) {
  s->rwstate = 2;
  tmp___9 = BIO_ctrl(s->wbio, 11, 0L, (void *)0);
  num1 = (long )((int )tmp___9);
  if (num1 <= 0L) {
    ret = -1;
    goto end;
  }
  s->rwstate = 1;
}
s->state = (s->s3)->tmp.next_state;
break;
case 3:
ssl3_cleanup_key_block(s);
if ((unsigned long )s->init_buf != (unsigned long )((void *)0)) {
  BUF_MEM_free(s->init_buf);
  s->init_buf = (BUF_MEM *)((void *)0);
}
if (! ((s->s3)->flags & 4L)) {
  ssl_free_wbio_buffer(s);
}
s->init_num = 0;
s->new_session = 0;
ssl_update_cache(s, 1);
if (s->hit) {
  (s->ctx)->stats.sess_hit ++;
}
ret = 1;
s->handshake_func = (int (*)())(& ssl3_connect);
(s->ctx)->stats.sess_connect_good ++;
if ((unsigned long )cb != (unsigned long )((void *)0)) {
      ((*cb))(s, 32, 1);
    }
    goto end;
    default: ERR_put_error(20, 132, 255, (char const *)"s3_clnt.c", 418);
    ret = -1;
    goto end;
    }
    if (! (s->s3)->tmp.reuse_message) {
      if (! skip) {
        if (s->debug) {
          ret = (int )BIO_ctrl(s->wbio, 11, 0L, (void *)0);
          if (ret <= 0) {
            goto end;
          }
        }
        if ((unsigned long )cb != (unsigned long )((void *)0)) {
          if (s->state != state) {
            new_state = s->state;
            s->state = state;
            ((*cb))(s, 4097, 1);
            s->state = new_state;
          }
        }
      }
    }
    skip = 0;
  }
  end:
  s->in_handshake --;
  if ((unsigned long )cb != (unsigned long )((void *)0)) {
    ((*cb))(s, 4098, ret);
  }
  return (ret);
}
```

A.4 Client Library Specifications

The complete input to MAGIC for all the library routines invoked by `ssl3_connect` and their corresponding EFSMs is as follows:

```
cproc ssl3_client_hello {
    abstract {ssl3_client_hello_abs,1,Ssl3ClientHello};
}

cproc ssl3_get_server_hello {
    abstract {ssl3_get_server_hello_abs,1,Ssl3GetServerHello};
}

cproc ssl3_get_finished {
    abstract {ssl3_get_finished_abs,1,Ssl3GetFinished};
}

cproc ssl3_get_server_certificate {
    abstract {ssl3_get_server_certificate_abs,1,Ssl3GetServerCertificate};
}

cproc ssl3_send_change_cipher_spec {
    abstract {ssl3_send_change_cipher_spec_abs,1,Ssl3SendChangeCipherSpec};
}

cproc ssl3_get_key_exchange {
    abstract {ssl3_get_key_exchange_abs,1,Ssl3GetKeyExchange};
}

cproc ssl3_send_finished {
    abstract {ssl3_send_finished_abs,1,Ssl3SendFinished};
}

cproc ssl3_get_certificate_request {
    abstract {ssl3_get_certificate_request_abs,1,Ssl3GetCertificateRequest};
}

cproc ssl3_get_server_done {
    abstract {ssl3_get_server_done_abs,1,Ssl3GetServerDone};
}

cproc ssl3_send_client_certificate {
    abstract {ssl3_send_client_certificate_abs,1,Ssl3SendClientCertificate};
}

cproc ssl3_send_client_key_exchange {
    abstract {ssl3_send_client_key_exchange_abs,1,Ssl3SendClientKeyExchange};
}

cproc ssl3_send_client_verify {
    abstract {ssl3_send_client_verify_abs,1,Ssl3SendClientVerify};
}

Ssl3ClientHello =
(
  exch_client_hello -> return {$0 == 1} -> STOP |
  return {$0 == -1} -> STOP
).

Ssl3GetServerHello =
(
  exch_server_hello -> return {$0 == 1} -> STOP |
  return {$0 == -1} -> STOP
).

Ssl3GetFinished =
(
  finished_srvr_to_clnt -> return {$0 == 1} -> STOP |
  return {$0 == -1} -> STOP
).

Ssl3GetServerCertificate =
(
  exch_server_certificate -> return {$0 == 1} -> STOP |
  return {$0 == -1} -> STOP
).

Ssl3SendChangeCipherSpec =
(
  send_change_cipher_spec -> return {$0 == 1} -> STOP |
  return {$0 == -1} -> STOP
).

Ssl3GetKeyExchange =
(
  key_exchange_srvr_to_clnt -> return {$0 == 1} -> STOP |
  return {$0 == -1} -> STOP
).
```

```
Ssl3SendFinished =
(
  finished_clnt_to_srvr -> return {$0 == 1} -> STOP |
  return {$0 == -1} -> STOP
).

Ssl3GetCertificateRequest =
(
  certificate_request_srvr_to_clnt -> return {$0 == 1} -> STOP |
  return {$0 == -1} -> STOP
).

Ssl3GetServerDone =
(
  ssl3_get_server_done -> return {$0 == 1} -> STOP |
  return {$0 == -1} -> STOP
).

Ssl3SendClientCertificate =
(
  exch_client_certificate -> return {$0 == 1} -> STOP |
  return {$0 == -1} -> STOP
).

Ssl3SendClientKeyExchange =
(
  key_exchange_clnt_to_srvr -> return {$0 == 1} -> STOP |
  return {$0 == -1} -> STOP
).

Ssl3SendClientVerify =
(
  cert_verify_clnt_to_srvr -> return {$0 == 1} -> STOP |
  return {$0 == -1} -> STOP
).
```

A.5 Complete OpenSSL Specification

We have presented the input to MAGIC for both the OpenSSL client and server components. We now present the input for providing a specification against which MAGIC will check simulation conformance. Note that the actions appearing in the specification must be the same as those in the alphabet of the EFSMs presented earlier. Also note the specification is non-deterministic. This causes it to blowup during determinization which contributes to the improved performance of simulation when compared to trace containment (cf. Section 6.5).

```
cprog ssl3 = ssl3_accept,ssl3_connect {
    abstract ssl3,
        {$1->state == (0x110|0x2000),
         $1->state == (0x04|(0x1000|0x2000))},Ssl3;
}

Ssl3 =
(
  epsilon -> SrClntHelloA |
  epsilon -> SwHelloReqA
),

SrClntHelloA =
(
  exch_client_hello -> SwSrvrHelloA |
  return {$0 == -1} -> STOP
),

SwHelloReqA =
(
  send_hello_request -> SwFlushSwHelloReqC |
  return {$0 == -1} -> STOP
),

SwFlushSwHelloReqC =
(
  epsilon -> SwHelloReqC |
  return {$0 == -1} -> STOP
),

SwHelloReqC =
(
  epsilon -> Ok |
  return {$0 == -1} -> STOP
),

Ok = ( return {$0 == 1} -> STOP ),

SwSrvrHelloA =
(
  exch_server_hello -> SwSrvrHelloA1 |
  return {$0 == -1} -> STOP
),

SwSrvrHelloA1 =
(
  epsilon -> SwChangeA |
  epsilon -> SwCertA |
  return {$0 == -1} -> STOP
),

SwChangeA =
(
  send_change_cipher_spec -> SwChangeA1 |
  return {$0 == -1} -> STOP
),

SwChangeA1 =
(
  epsilon -> SwFinishedA |
  return {$0 == -1} -> STOP
),

SwCertA =
(
  exch_server_certificate -> SwCertA1 |
  epsilon -> SwCertA1 |
  return {$0 == -1} -> STOP
),
SwCertA1 =
(
  epsilon -> SwKeyExchA |
  return {$0 == -1} -> STOP
),

SwKeyExchA =
(
  key_exchange_srvr_to_clnt -> SwKeyExchA1 |
  epsilon -> SwKeyExchA1 |
  return {$0 == -1} -> STOP
),

SwKeyExchA1 =
(
  epsilon -> SwCertReqA |
  return {$0 == -1} -> STOP
),

SwFinishedA =
(
  finished_srvr_to_clnt -> SwFinishedA1 |
  return {$0 == -1} -> STOP
),

SwFinishedA1 =
(
  epsilon -> SwFlushSrFinishedA |
  epsilon -> SwFlushOk |
  return {$0 == -1} -> STOP
),

SwFlushSrFinishedA =
(
  epsilon -> SrFinishedA |
  return {$0 == -1} -> STOP
),

SwFlushOk =
(
  epsilon -> Ok |
  return {$0 == -1} -> STOP
),

SwCertReqA =
(
  epsilon -> SwSrvrDoneA |
  certificate_request_srvr_to_clnt -> SwFlushSrCertA |
  return {$0 == -1} -> STOP
),

SwFlushSrCertA =
(
  epsilon -> SrCertA |
  return {$0 == -1} -> STOP
),

SrCertA =
(
  check_client_hello -> SrCertA1 |
  return {$0 == -1} -> STOP
),

SrCertA1 =
(
  epsilon -> SrClntHelloC |
  exch_client_certificate -> SrKeyExchA |
  exch_client_certificate -> return {$0 == -1} -> STOP |
  return {$0 == -1} -> STOP
),

SrFinishedA =
(
  finished_clnt_to_srvr -> SrFinishedA1 |
  return {$0 == -1} -> STOP
),

SrFinishedA1 =
(
  epsilon -> Ok |
  epsilon -> SwChangeA |
  return {$0 == -1} -> STOP
),
SrClntHelloC = ( epsilon -> SrClntHelloA ),

SrKeyExchA =
(
  key_exchange_clnt_to_srvr -> SrCertVrfyA |
  key_exchange_clnt_to_srvr -> return {$0 == -1} -> STOP |
  return {$0 == -1} -> STOP
),

SrCertVrfyA =
(
  cert_verify_clnt_to_srvr -> SrFinishedA |
  cert_verify_clnt_to_srvr -> return {$0 == -1} -> STOP |
  return {$0 == -1} -> STOP
),

SwSrvrDoneA =
(
  send_server_done -> SwFlushSrCertA |
  return {$0 == -1} -> STOP
).
```