An Iterative Framework for Simulation Conformance

Sagar Chaki Edmund Clarke
Carnegie Mellon University
{chaki|emc}@cs.cmu.edu

Somesh Jha
Univ of Wisconsin
jha@cs.wisc.edu

Helmut Veith
TU München
veith@cs.tum.edu

Abstract

MAGIC is a software verification project for C source code which verifies conformance of software components against state-machine specifications. To this aim, MAGIC extracts abstract software models using predicate abstraction, and resolves the inherent trade-off between model accuracy and scalability by an iterative abstraction refinement methodology. This paper presents the core principles implemented in the MAGIC verification engine, i.e., specification conformance using simulation and abstraction refinement. Viewing counterexamples as winning strategies in a simulation game between the implementation and the specification, we describe an algorithm where abstractions are refined on the basis of multiple winning strategies simultaneously. The refinement process is iterated until either a conformance with the specification is established, or a strategy to violate the specification is found to be realizable. In addition to the increase in expressiveness achieved by using simulation instead of trace containment, experimental results using OpenSSL indicate that our approach can lead to orders of magnitude improvement in verification time.

1 Introduction

The sustained success of computer technology has rendered our critical infrastructures, such as medicine, power, telecommunications, transportation and finance, highly dependent on software. As the disruption or malfunction of critical services can have catastrophic effects with life-endangering consequences, systematic approaches for asserting software correctness are more important than ever. Software verification has been the subject of ambitious projects for several decades, and this research tradition has provided us with important fundamental notions of program semantics and structure. Software verification tools, however, have until recently not attained the level of scalability and practical applicability required by industry. Owing to both technical and economic reasons, the hardware industry, in contrast, is significantly closer to automated industrial engineering processes than the software industry; in particular, formal verification methods such as model checking [18, 19] have been successfully applied by the industry, and are part of the development process.

Motivated by the urgent industrial need, the success and maturity of formal methods in hardware verification, and by the arrival of new paradigms such as predicate abstraction [30], several research groups [11, 28, 34, 43, 26, 27] have started to develop a new generation of software verification tools. A common feature of these tools is an extended model checking algorithm which interacts with theorem provers and decision procedures to reason about software abstractions, in particular about abstractions of data types.

MAGIC [35, 13] is a software verification tool from this family which focuses on modular verification of C code. In MAGIC, software components are described by labeled transition systems, a form of state machines. The goal of MAGIC is to verify whether the implementation *Comp* of the software component conforms to its specification S, i.e., whether all possible behaviors of the implementation are subsumed by the specification. In contrast to the other mentioned tools, the conformance notion used in MAGIC is *simulation* [37] denoted by $\preceq$. Further on, we shall use "conformance" and "simulation" synonymously.

Note that simulation is very different from trace containment, as reflected by a large body of work in both process algebra and model checking. Simulation is in fact a more general concept, analogous to the distinction between branching time and linear time. Moreover, simulation is easier to compute than trace containment, and guarantees preservation of the universal μ-calculus and its fragments such as ACTL*.

Throughout this paper, we assume that the specification S is stuttering, i.e., can self-loop in every state. This enables us to capture the correspondence between specification and implementation, where a single step of the specification can correspond to multiple steps of the implementation.

Since a software component generally gives rise to an infinite state system, MAGIC will not directly check *Comp* against S, but extract an intermediate abstract model $A(\mathit{Comp})$ which simulates *Comp by construction*, and then verify whether $A(\mathit{Comp})$ is, in turn, simulated by S, i.e.,

$$\mathit{Comp} \preceq A(\mathit{Comp}) \preceq S.$$

The obvious problem in this approach is to find a good abstraction $A(\mathit{Comp})$. If $A(\mathit{Comp})$ is too close to the original system *Comp*, then the computational cost for checking $A(\mathit{Comp}) \preceq S$ may be prohibitively expensive. On the other hand, if $A(\mathit{Comp})$ is very coarse then it may well be that *relevant features of Comp are abstracted away*, i.e., that $A(\mathit{Comp}) \not\preceq S$, even though $\mathit{Comp} \preceq S$ holds. In this case, nevertheless, an inspection of $A(\mathit{Comp}) \not\preceq S$ provides an abstract counterexample *CE*. Owing to the approximate nature of the abstraction, this abstract counterexample may be *spurious*, i.e., does not correspond to any real behavior of the implementation. This motivates the use of a recent abstraction refinement paradigm known as *CEGAR (***C***ounter***E***xample* **G***uided* **A***bstraction* **R***efinement)* [10, 20, 26, 33]:

1. **Abstract.** Create an abstraction $A(\mathit{Comp})$ of the component which simulates *Comp* by construction.

2. **Verify.** Verify whether $A(Comp) \preceq S$. If this is the case, verification is successful. Otherwise, we obtain a potentially spurious counterexample *CE*. Determine whether *CE* is spurious. If *CE* is not spurious, report the counterexample and stop; otherwise go to the next step.

3. **Refine.** Use the spurious *CE* to refine the abstraction $A(Comp)$ as to eliminate *CE*, and goto the first step.

The main contribution of this paper is a CEGAR-style method for simulation-based software verification. More specifically, we shall describe the following results:

- **Strategy Guided Abstraction Refinement.** Viewing simulation as a two-player game between the implementation and the specification, *a counterexample amounts to a winning strategy for the implementation in the simulation game* (see Section 4). This winning strategy can be described by a finite tree whose nodes are labeled by game states, and whose transitions are induced by the moves in the game. In Section 5, we give a formal foundation for this observation, and describe a fixed-point algorithm for computing a general game structure from which all possible strategies can be derived. Section 7 describes in detail how an abstract winning strategy can be matched against the C source code to analyze the strategy for spurious behavior, and how to refine the abstraction so as to eliminate spurious winning strategies.
- **Comparing simulation and trace containment.** The strategy trees obtained as counterexamples are more expressive than trace counterexamples, and provide us with a systematic way to reduce the number of refinement steps. We have performed experiments to compare simulation against trace containment on real-life benchmarks. The results reported in Section 8 indicate that *simulation can lead to over 20 times reduction in time and over 50 times reduction in the number of iterations required for verification over trace containment.*
- **Efficient Refinement on Multiple Counterexample Trees.** We compute multiple counterexample strategies using a modified algorithm for solving HORNSAT [42]. Simultaneous refinement on several counterexamples can achieve even faster convergence in the abstraction-refinement loop. There is however a trade-off between the number of simultaneous counterexamples and the overall efficiency. Beyond a certain threshold, the

overhead of manipulating a large number of counterexamples will outweigh the benefits. Experimental evidence supporting this insight is presented in Section 8.

The experiments reported in this paper are based on source code for OpenSSL. Additional case studies with the MicroC/OS [36] real time operating system and the controller software of a metal casting plant are currently being performed in collaboration with other members of the CMU verification group. These case studies have lead to the detection of errors to be reported soon.

2 Related Work

The architecture and general approach used in MAGIC along with the model-extraction process are described in an earlier paper [13]. Our earlier paper [13] does not describe the refinement process which was under development at that time, and is the focus of the current paper.

Using the MAGIC infrastructure, several other research directions have also been pursued: The problem of optimizing the number of predicates in predicate abstraction was addressed in [14] using pseudo-Boolean constraints. This work is orthogonal to the current work, and applicable to various tools [10, 33] which use predicate abstraction. The experiments reported in [14] use the same C source code, but for different experiments with different specifications.

Another orthogonal question of current interest is state/event-based specifications. MAGIC has been used to develop prototypes for verifying concurrent C programs against both linear [17] and branching time [15] state/event specifications. Finally, an iterative and compositional deadlock detection scheme based on the trace-refusal theory of CSP has been developed, and implemented in MAGIC [16].

During the last years, several software verification systems including Bandera [26], Java PathFinder [34], ESC Java [28], SLAM [43, 9, 10], BLAST [11, 33], and MC [27, 31] have been developed. The first three systems focus on Java and, therefore, have to deal with issues such as inheritance and shapes of heap allocated data structures. Hence, research on these systems cannot be directly compared to our work. The last three systems handle C programs. MC uses a sophisticated pattern matching algorithm on C programs with surprisingly good results in certain domains. Since MC does not use the predicate abstraction framework and the CEGAR methodology, it is hard to fairly compare MC-related research to the current paper. SLAM and BLAST are closest in spirit to our work. However, neither tool uses game-semantics-based refinement. Our tool, like SLAM, uses symbolic algorithms. On the other hand, BLAST uses an on-the-fly reachability analysis algorithm. We believe that our game-semantics-based refinement can be used in other software verification tools with advantageous effects.

Game semantics has applications in several areas including supervisory control [41], hardware and program synthesis [12, 39], modular verification [5, 6, 25, 3] and schedulability analysis [4]. Game semantics has also been used to devise efficient algorithms for computing certain process pre-orders [40], and to provide full abstract semantics for several languages [1, 2, 29]. The most interesting application of game semantics in our context is the recent work by Henzinger et al. [32] who present a counterexample guided abstraction refinement framework for controller *synthesis*. As the relationship between controller and plant is described as a two-player game, their synthesis problem also naturally leads to a strategy guided refinement approach. Our work differs from Henzinger et al.'s in several important aspects. While they provide a very general theoretical framework, we describe a concrete procedure for strategy guided refinement in the context of verifying C programs. In addition, we have implemented our methodology as part of a software verification tool and gauged its effectiveness on real-life benchmarks. Although it may be possible to paraphrase our approach in their terminology (and in Cousot's abstract interpretation framework [23] as well for that matter), to our best knowledge our paper is the first one to use and evaluate strategy guided refinement in the context of software verification.

The refinement technique presented in [32] is preceded by a systematic study on tree-like counterexamples in a counterexample-guided refinement framework [21]. This paper is a fundamental study on counterexamples for ACTL-style logics, which have a different format from the ones encountered in the current paper. On an abstract level, the refinement principles for tree-like counterexamples identified in [21] are applicable to the current paper as well as to [32]. For better understanding, we emphasize this resemblance in our description in Section 7.

3 The MAGIC Tool

This section gives background information about the MAGIC framework and our abstraction-refinement-based software verification algorithm.

Labeled Transition Systems. A labeled transition system (LTS) M is a 4-tuple $(Q, \mathit{init}, \mathit{Act}, T)$, where (i) Q is a finite non-empty set of states, (ii) $\mathit{init} \in Q$ is the initial state, (iii) Act is the set of actions, and (iv) $T \subseteq Q \times \mathit{Act} \times Q$ is the transition relation. We assume that there is a distinguished state STOP $\in Q$ which has no outgoing transitions, i.e., $\forall q' \in Q, \forall a \in \mathit{Act}, (\mathrm{STOP}, a, q') \notin T$. If $(q, a, q') \in T$, then (q, q') will be referred to as an a-transition and will be denoted by $q \xrightarrow{a} q'$. For a state q and action a, we define $\mathrm{Succ}(q,a) = \{q' : q \xrightarrow{a} q'\}$. Action a is said to be *enabled* at state q iff $\mathrm{Succ}(q,a) \neq \emptyset$.

Actions. In accordance with existing practice, we use actions to denote externally visible behaviors of systems being analyzed, e.g. acquiring a lock. Actions are atomic, and are distinguished simply by their names. Since we are analyzing C programs, a procedural language, we model the termination of a procedure (i.e., a return from the procedure) by a special class of actions called *return actions*. Every return action r is associated with a unique return value $\mathit{RetVal}(r)$. Return values are either integers or `void`. We denote the set of all return actions whose values are integers by *IntRet* and the special return action whose value is `void` by *VoidRet*. All actions which are not return actions are called *basic actions*. A distinguished basic action τ denotes the occurrence of an unobservable internal event.

Modular Verification. The goal of MAGIC is to verify whether the implementation of a software component is simulated by its specification. Since MAGIC can handle a group of procedures with a DAG-like call graph as one procedure by inlining, we assume that the implementation of a software component consists of a single procedure *proc*. The specification is expressed as a *procedure abstraction* (PA); formally a PA for *proc* is a tuple $\langle d, l\rangle$ where

- d is a declaration for *proc* that indicates its name, the names of its formal parameters and the type of its return value.
- l is a finite list $\langle g_1, M_1\rangle, \ldots, \langle g_n, M_n\rangle$ where each g_i is a guard formula ranging over the parameters and globals of *proc*, and each M_i is an LTS.

The above PA expresses that *proc* conforms to one LTS chosen among the M_i's. More precisely, *proc* conforms to M_i if the corresponding guard g_i evaluates to true over the *actual arguments* and globals passed to *proc*[1]. PAs can be viewed as a variation of the classical precondition-postcondition based specification scheme. In a PA, the g_i's correspond to preconditions while the M_i's express the observable behavior of *proc* when it is executed from an initial state satisfying g_i. The goal of MAGIC is to prove that a user-defined PA for *proc* is valid. The role of PAs in this process is twofold:

- A *target PA* is used to describe the desired behavior of the procedure *proc*.
- To assist the verification process, we employ valid PAs (called the *assumption* PAs) for library routines used by *proc*.

[1]We require that the guard formulas g_i be mutually exclusive and complete (i.e. cover all possibilities) so that the choice of M_i is always uniquely defined.

Without loss of generality we will assume throughout this paper that the target PA contains only one guard G_{Spec} and one LTS M_{Spec}. Also, we will only consider procedures that terminate by returning. In particular, we do not handle constructs like `setjmp` and `longjmp`. Furthermore, since LTSs are used to model procedures, any LTS $(Q, init, Act, T)$ must obey the following condition: $\forall s \in Q, s \xrightarrow{a}$ STOP iff a is a return action.

Strategy Guided Abstraction Refinement. MAGIC implements the strategy-guided abstraction refinement procedure described in the previous section. Based on the terminology introduced in this section, we can paraphrase the refinement process as follows:

- **Step 1 : Model Construction.** Extract an LTS M_{Imp} from *proc* using the assumed PAs and the guard G_{Spec}. In MAGIC, the model is constructed from the control flow graph (CFG) of the program in combination with an abstraction method called *predicate abstraction* [20, 24, 30]. To decide properties such as equivalence of predicates, we use decision procedures and theorem provers. The details of this step are described in Section 6. Note that *proc* and M_{Imp} are the analogue of *Comp* and $A(Comp)$ respectively in the context of MAGIC.
- **Step 2 : Verification.** Check whether M_{Spec} simulates M_{Imp}. If this is the case, the verification successfully terminates; otherwise, extract diagnostic feedback and perform step 3. We reduce simulation to the satisfiability of a certain Boolean formula, thus deferring the solution to highly efficient SAT procedures [13]. The details of this step are described in Section 4.
- **Step 3 : Refinement.** Use the diagnostic feedback obtained in step 2 to determine the reason behind the failure of the simulation check. If the cause is a real bug in *proc* we are done. Otherwise the property fails because M_{Imp} is not a sufficiently precise model for *proc*. In this case we refine the abstraction (see Section 7) and return to step 1 to compute a more precise M_{Imp}.

4 Simulation Games

As mentioned before, in our approach, verification amounts to checking that the specification LTS simulates the implementation LTS. Therefore, we consider simulation in more detail.

Simulation. Let $M_1 = (Q_1, init_1, Act, T_1)$ and $M_2 = (Q_2, init_2, Act, T_2)$ be two LTSs having the same alphabet. A relation $R \subseteq Q_1 \times Q_2$ is called a *simulation* between M_1 and M_2 iff the following condition holds: $\forall q_1, q_1' \in Q_1, q_2 \in Q_2, a \in Act$, if $(q_1, q_2) \in R$ and $q_1 \xrightarrow{a} q_1'$, then there exists $q_2' \in Q_2$ such that $q_2 \xrightarrow{a} q_2'$ and $(q_1', q_2') \in R$. We say that LTS M_2 *simulates* M_1 (denoted by $M_1 \preceq M_2$) if there exists a simulation relation $R \subseteq Q_1 \times Q_2$ between M_1 and M_2 such that $(init_1, init_2) \in R$.

In this section and the next, I will refer to the implementation M_{Imp}, S to the specification M_{Spec} and Act to Act_S. Also, let $S = (Q_S, init_S, Act, T_S)$ and $I = (Q_I, init_I, Act, T_I)$.

Games for Simulation. Suppose we want to determine whether $I \preceq S$. It is well-known [44] that this can be verified using a two-player game between the implementation I and the specification S. 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 LTS which he can move along transitions of his LTS. The location of the pebbles at the start of each round is called a game state, or position, and is denoted by (q_I, q_S) where q_I and q_S are the locations of the implementation's and specification's pebbles respectively. From a given position (q_I, q_S), the game proceeds as follows:

- **Implementation Challenge.** The implementation picks an action α and a next state $q_I' \in \text{Succ}(q_I, \alpha)$ and moves its pebble to q_I'. We denote such a challenge as simply $(q_I, q_S) \xrightarrow{\alpha} (q_I', ?)$.
- **Specification Response.** The specification responds by moving its pebble to a state $q_S' \in \text{Succ}(q_S, \alpha)$ and the game continues into the next round from position (q_I', q_S'). Note that the response must involve the same action (in this case α) as the corresponding challenge. Thus, the specification completes the challenge $(q_I, q_S) \xrightarrow{\alpha} (q_I', ?)$ into a transition $(q_I, q_S) \xrightarrow{\alpha} (q_I', q_S')$.
- **Winning Condition.** If any one of the players cannot move, then the other player wins. Moreover, if the game continues forever, the specification wins.

A simulation game is completely defined by I, S and the initial position. Let us denote the simulation game with (q_I, q_S) as the initial position by $\mathbf{Game}_{(q_I, q_S)}$. A position (q_I, q_S) is called a *winning position* iff I has a well-defined strategy to win $\mathbf{Game}_{(q_I, q_S)}$.

Fact 1 *$I \preceq S$ iff the implementation I does not have a strategy to win $\mathbf{Game}_{(init_I, init_S)}$, i.e., if $(init_I, init_S)$ is not a winning position.*

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

We will now give a formal framework which describes the game in such a way that CETrees can be easily extracted. We begin by defining the functions $\mathsf{Response}$ which maps a challenge c to the set of all new game positions that can result after S has responded to c.

$$\mathsf{Response}((q_I, q_S) \xrightarrow{\alpha} (q'_I, ?)) = \{q'_I\} \times \mathrm{Succ}(q_S, \alpha)$$

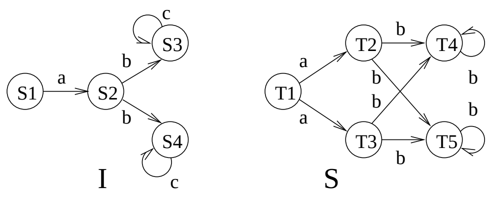

Figure 1: Two simple LTSs.

Example 1 *Let I and S be the LTSs from Figure 1. From position* $(\mathtt{S2}, \mathtt{T2})$*, I can pose the following two challenges due to two possible moves from* $\mathtt{S2}$ *on action b.*

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

For each of these challenges S can respond in two ways due to two possible moves from $\mathtt{T2}$ *on action b.*

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

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

Strategy Trees as Counterexamples. Formally, a CETree for $\mathbf{Game}_{(q_I, q_S)}$ is given by a labeled tree $(N, E, r, \mathrm{St}, \mathrm{Ch})$ 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 maps each tree node n to a game position $\mathrm{St}(n)$,
- Ch maps each tree node n to the challenge that I must pose from position $\mathrm{St}(n)$ in accordance with the strategy.

Note that, for a given node n, if $\mathrm{St}(n) = (q_I, q_S)$ then $\mathrm{Ch}(n) = (q_I, q_S) \xrightarrow{\alpha} (q'_I, ?)$ for some α and q'_I. Also, let $\mathrm{Child}(n)$ denote the set of children of n. We will write $n.\mathrm{St}$ and $n.\mathrm{Ch}$ to mean $\mathrm{St}(n)$ and $\mathrm{Ch}(n)$ respectively. Then the CETree $(N, E, r, \mathrm{St}, \mathrm{Ch})$ has to satisfy the following conditions:

C1 The root of the tree is mapped to the initial game state, i.e., $r.\mathrm{St} = (q_I, q_S)$.

C2 The children of a node n cover $\mathsf{Response}(n.\mathrm{Ch})$, i.e., the game positions to which the response of S can lead. In other words:

$$\mathsf{Response}(n.\mathrm{Ch}) = \{c.\mathrm{St} \mid c \in \mathrm{Child}(n)\}$$

C3 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: $\mathsf{Response}(l.\mathrm{Ch}) = \emptyset$.

Example 2 *Consider again I and S from Figure 1. Figure 2 shows a* CETree *for* $\mathbf{Game}_{(\mathtt{S1},\mathtt{T1})}$. *Inside each node n we show the challenge* $\mathrm{Ch}(n)$.

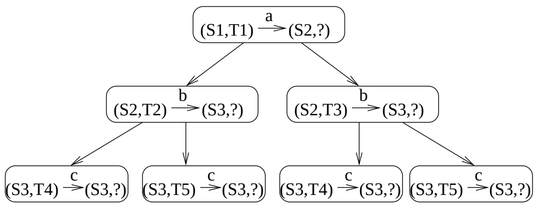

Figure 2: CETree for a simulation game.

5 Computing Strategies

In this section we describe an algorithm to compute CETrees. Recall that a CETree describes a winning strategy for the implementation I to win the the simulation game. We will first describe the algorithm **WinPos** which computes the set of winning positions along with their associated challenges; this data is then used to construct a CETree.

The Algorithm **WinPos** is described in Figure 3. It collects the winning positions of I in the set W. Starting with $W = \emptyset$, it adds new winning positions to W until no more winning positions can be found. Note that in the first iteration $W = \emptyset$, and therefore the condition $\mathsf{Response}(c) \subseteq W$ amounts to $\mathsf{Response}(c) = \emptyset$. The latter condition in turn expresses that c is a victorious challenge (see **C3** above).

```
**Algorithm WinPos**$(I, S)$
// $I = (Q_I, init_I, Act, T_I), S = (Q_S, init_S, Act, T_S)$
$W, \mathrm{Ch} := \emptyset$;
**do forever**
    **find** challenge $c := (q_I, q_S) \xrightarrow{\alpha} (q_I', ?)$
        such that $\mathsf{Response}(c) \subseteq W$
        // *all responses are winning positions*
    **if** not found **return** $\langle W, \mathrm{Ch} \rangle$;
    $W := W \cup \{(q_I, q_S)\}$;
    $\mathrm{Ch}(q_I, q_S) := c$;
**od**;
```

Figure 3: The algorithm **WinPos** computes the set W of winning positions for the implementation I; the challenges are stored in Ch.

In Figure 4 we present the verification algorithm **Verify** that works as follows: it first invokes **WinPos** to compute the set W of winning positions. If the initial position $(init_I, init_S)$ is not in W, then the implementation cannot win the simulation game $\mathbf{Game}_{(init_I, init_S)}$. In this case, **Verify** declares that "S simulates I" (cf. Fact 1) and terminates. Otherwise, it invokes algorithm **ComputeStrategy** (cf. Figure 5) to compute a CETree for $\mathbf{Game}_{(init_I, init_S)}$.

Algorithm **ComputeStrategy** takes the following as inputs: (i) a winning position (q_I, q_S), (ii) the set of all winning positions W, and (iii) additional challenge information $\mathsf{Ch0}$. It constructs a CETree for the simulation game $\mathbf{Game}_{(q_I, q_S)}$ and returns the root of this CETree. Note that at the top level, **ComputeStrategy** is invoked by **Verify** with the winning position $(init_I, init_S)$. This call therefore returns a CETree for the complete simulation game $\mathbf{Game}_{(init_I, init_S)}$.

```
**Algorithm Verify**$(I, S)$**;**
// $I = (Q_I, init_I, Act, T_I), S = (Q_S, init_S, Act, T_S)$
$\langle W, \mathrm{Ch} \rangle :=$ **WinPos**$(I, S)$;
**if** $(init_I, init_S) \notin W$
    **return** "S simulates I";
**else**
    **return ComputeStrategy**$(init_I, init_S, \langle W, \mathrm{Ch} \rangle)$;
```

Figure 4: The algorithm **Verify** checks for simulation, and invokes counterexample generation in case of violation.

When **ComputeStrategy** is invoked with position (q_I, q_S), it first creates a root node r and associates position (q_I, q_S) and challenge $\mathsf{Ch0}(q_I, q_S)$ with r. It then considers all the positions reachable by responding to $\mathsf{Ch0}(q_I, q_S)$, i.e., all the positions with which the next round of the game might begin. For each of these positions, **ComputeStrategy** constructs a CETree by invoking itself recursively. **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

$\mathsf{Response}(\mathsf{Ch0}(q_I, q_S)) = \emptyset$, i.e., if $\mathsf{Ch0}(q_I, q_S)$ is a victorious challenge, then r becomes a leaf node as expected (cf. **C3** above).

Efficient Computation of Multiple Counterexample Trees. For given I and S, the set of winning positions W computed by **WinPos** is uniquely defined, i.e., each position (q_I, q_S) is either the root of some winning strategy (i.e., $(q_I, q_S) \in W$) or not (i.e, $(q_I, q_S) \notin W$). There may, however, be multiple winning strategies from position (q_I, q_S), simply because there may be different challenges I can pose, which all will ultimately lead to I's victory.

In the algorithm **WinPos**, 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 $\mathrm{Ch}(q_I, q_S)$. The challenge information stored in Ch is crucially used in **ComputeStrategy**, the algorithm which constructs the winning strategy.

Thus, depending on **WinPos**'s choices for the challenges c, **ComputeStrategy** will output different winning strategies. While all these strategies are by construction winning strategies for I, they may differ in various aspects, for example, the tree size or the actions and states involved. Moreover, different winning strategies may exploit different implementation errors in I. In MAGIC, we use heuristics to guide **WinPos**'s choice of the challenge c, thereby obtaining different winning strategies. In Section 8, we will see that in our experiments, using a set of different winning strategies instead of one indeed helps to save time and memory.

As described in Figure 3, the algorithm **WinPos** is essentially a least fixed point algorithm[2] for computing W and additional challenge information Ch. In MAGIC we implement **WinPos** by reducing it to a satisfiability problem for weakly negated HORNSAT (N-HORNSAT) formulas. More precisely, given I and S, we construct an N-HORNSAT formula $\phi_{(I,S)}$ such that $I \preceq S$ iff $\phi_{(I,S)}$ is satisfiable [42]. We then input $\phi_{(I,S)}$ to a linear-time online algorithm. This algorithm is based on an existing N-HORNSAT satisfiability algorithm [8]. However we extended it to not only check if $\phi_{(I,S)}$ is satisfiable, but also to compute W and Ch as a side-effect.

6 Model Construction

Let $M_{Spec} = (Q_S, init_S, Act_S, T_S)$ and the assumption PAs be $\{PA_1, \ldots, PA_k\}$. In this section we show how to construct $M_{Imp} = (Q_I, init_I, Act_S, T_I)$ from *proc* using the assumption PAs, the guard G_{Spec} and the predicates. M_{Imp} models the execution of *proc* and its construction relies on two key principles:

[2]**WinPos** can be viewed as the dual of the greatest fixed point algorithm for computing the maximal simulation relation between I and S.

```
**Algorithm ComputeStrategy**$(q_I, q_S, \langle W, \mathsf{Ch0}\rangle)$;
// $(q_I, q_S)$ is a winning position in $W$
**create new tree node** $r$ with
    $r.\mathrm{St} := (q_I, q_S)$ and $r.\mathrm{Ch} := \mathsf{Ch0}(q_I, q_S)$;
**for all** $(c_I, c_S) \in \mathsf{Response}(\mathsf{Ch0}(q_I, q_S))$
    **create tree edge**
        $r \longrightarrow$ **ComputeStrategy**$(c_I, c_S, \langle W, \mathsf{Ch0}\rangle)$;
**return** $r$;
```

Figure 5: The algorithm **ComputeStrategy** recursively computes a winning strategy for showing that $(q_I, q_S) \in W$; it outputs the root of the strategy tree.

Principle 1. Every state q of M_{Imp} models a set of execution states $\hat{q}$ of *proc*; consequently every state of M_{Imp} is composed of a control component and a data component.

- The control components represent values of the program counter, and are obtained from the CFG of *proc*.
- The data components are *abstract representations* of the memory state of *proc*. These abstract representations are obtained using predicate abstraction.

Principle 2. The transition relation T_I obeys the following condition: let q_1 and q_2 be any two states of M_{Imp} and let $\hat{q_1}$ and $\hat{q_2}$ be the corresponding sets of execution states of *proc*. If there exists $e_1 \in \hat{q_1}$ and $e_2 \in \hat{q_2}$ such that e_1 makes a transition to e_2 during the execution of *proc*, then $(q_1, q_2) \in T_I$. This requirement on T_I ensures that *proc* $\preceq M_{Imp}$ [22]. The actual construction of T_I involves reasoning about C expressions, and will therefore require the use of a theorem prover.

Without loss of generality, we can assume that there are only five kinds of statements in *proc*: assignments, call-sites, `if-then-else` branches, `goto` and `return`. Note that call-sites correspond to library routines called by *proc* for which assumption PAs are available. We assume the absence of indirect function calls and pointer dereferences in the LHS's of assignments[3]. We denote by *Exp* the set of all *pure* (side-effect free) C expressions over the variables of *proc*.

As a running example of *proc*, we use the C program shown in Figure 6. It invokes two library routines `do_a` and `do_b`. Let the guard and LTS list in the assumption PA for `do_a` be $\langle$TRUE , `Do_A`$\rangle$. This means that under all invocation conditions, `do_a` is simulated by the LTS `Do_A`. Similarly the guard and LTS list in the assumption PA for `do_b` is $\langle$TRUE , `Do_B`$\rangle$. The LTSs `Do_A` and `Do_B` are described in Figure 7. Also we use $G_{Spec} =$ TRUE and $M_{Spec} =$ `Spec` (shown in Figure 7).

[3]MAGIC handles these constructs via conservative alias analysis [7].

```
S0:  int x,y=8;
S1:  if(x == 0) {
S2:    do_a();
S4:    if (y < 10) { S6:  return 0; }
        else { S7:  return 1; }
      } else {
S3:    do_b();
S5:    if(y > 5) { S8:  return 2; }
        else { S9:  return 3; }
      }
```

Figure 6: The example *proc*.

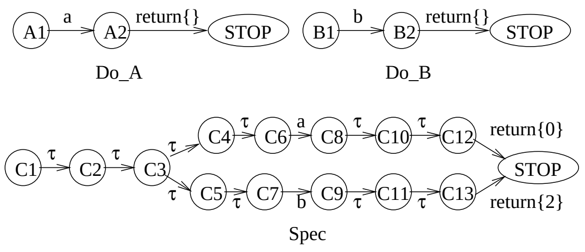

Figure 7: The LTSs in the assumption PAs for `do_a` and `do_b` and the specification LTS `Spec`. The return action with return value `void` is denoted by `return{}`.

Initial abstraction with control flow automata. The construction of M_{Imp} begins with the construction of the control flow automaton (CFA) of *proc*. The states of the CFA correspond to control points (or statements) of *proc* while the transitions model *proc*'s control flow. Formally the CFA is a 4-tuple $\langle CL, IL, CF, Stmt \rangle$ where:

- CL is a set of states.
- $IL \in CL$ is an initial state.
- $CF \subseteq CL \times CL$ is a set of transitions.
- $Stmt$ is a labeling function.

CL contains a distinguished FINAL state that models the termination point of *proc*. The labeling function $Stmt$ captures the correspondence between CL and *proc*'s statements; it maps each non-FINAL state of the CFA to a unique statement of *proc*. In particular, $Stmt(IL)$ is the initial statement of *proc*. Since CF models the flow of control, $(cl_1, cl_2) \in CF$ iff one of the following conditions hold:

- $Stmt(cl_1)$ is an assignment, call-site or `goto` with $Stmt(cl_2)$ as its unique successor in *proc*.
- $Stmt(cl_1)$ is a branch with $Stmt(cl_2)$ as its `then` or `else` successor in *proc*.
- $Stmt(cl_1)$ is a `return` statement and $cl_2 = $ FINAL.

Example 3 *The CFA of our example program is shown in Figure 8. Each non-final state is labeled by the corresponding statement label (the* FINAL *state is labeled by* `FINAL`*). Henceforth we will refer to each CFA state by its label. Therefore the states of the CFA in Figure 8 are* `S0 ...S9, FINAL` *with* `S0` *being the initial state.*

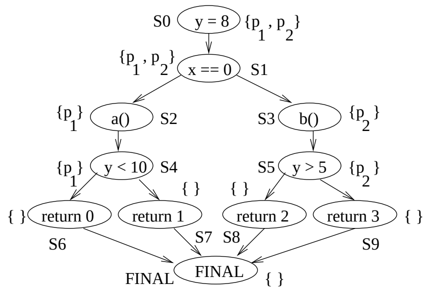

Figure 8: The CFA for our example program. Each non-FINAL state is labeled the same as its corresponding statement. The initial state is labeled `S0`. The states are also labeled with inferred predicates when $\mathcal{P} = \{p_1, p_2\}$ where $p_1 = (y < 10)$ and $p_2 = (y > 5)$. If $\mathcal{P} = \emptyset$, then the set of inferred predicates is also empty for every state of the CFA.

Predicate inference. Since the construction of M_{Imp} from *proc* involves predicate abstraction, it is parameterized by a set of predicates $\mathcal{P}$. Once $\mathcal{P}$ has been fixed, the next step in the construction of M_{Imp} is to associate with each state s of the CFA a finite subset of *Exp* derived from $\mathcal{P}$. This process is known as predicate inference and the set of expressions it associates with an arbitrary CFA state s is denoted by $\mathcal{P}_s$. The predicate inference algorithm has been presented elsewhere [14] and we shall not describe it here.

Example 4 *Consider the CFA described in Example 3. Suppose $\mathcal{P}$ contains the conditions for branches* `S4` *and* `S5`*. Figure 8 shows the CFA with each state s labeled by $\mathcal{P}_s$.*

Predicate valuation and concretization. For a CFA node s suppose $\mathcal{P}_s = \{p_1, \ldots, p_k\}$. Then a *valuation* of $\mathcal{P}_s$ is a Boolean vector $v_1, \ldots, v_k$. Therefore there are exactly 2^k possible valuations of $\mathcal{P}_s$. Let $\mathcal{V}_s$ be the set of these 2^k predicate valuations of $\mathcal{P}_s$. The *predicate concretization* function $\Gamma_s : \mathcal{V}_s \rightarrow Exp$ is defined as follows. Given a valuation $V = \{v_1, \ldots, v_k\} \in \mathcal{V}_s$, $\Gamma_s(V) = \bigwedge_{i=1}^{k} p_i^{v_i}$ where $p_i^{\text{TRUE}} = p_i$ and $p_i^{\text{FALSE}} = \neg p_i$. As a special case, if $\mathcal{P}_s = \emptyset$, then $\mathcal{V}_s = \{\perp\}$ and $\Gamma_s(\perp) = \text{TRUE}$.

Execution states of *proc*. An execution state of *proc* consists of a program counter (PC) value that corresponds to some control location l and a valuation v to various variables in scope at l. Let **CL** be a function that maps an execution state e to the CFA state corresponding to the PC value of e. Similarly let **PV** be a function mapping an execution state e and a set of predicates $\mathcal{P}$ to the valuation of $\mathcal{P}$ under e.

Example 5 *Consider the CFA described in Example 3 and the inferred predicates as explained in Example 4. Recall that* $\mathcal{P}_{\texttt{S1}} = \{(y < 10), (y > 5)\}$. *Suppose* $V_1 = \{0, 1\}$ *and* $V_2 = \{1, 0\}$. *Then* $\Gamma_{\texttt{S1}}(V_1) = (\neg(y < 10)) \wedge (y > 5)$ *and* $\Gamma_{\texttt{S1}}(V_2) = (y < 10) \wedge (\neg(y > 5))$. *Now consider an execution state* e *such that the PC value corresponds to the branch labeled* `S1` *and the current value of* y *is 11. Then* $\mathbf{CL}(e) = \texttt{S1}$ *and* $\mathbf{PV}(e, \mathcal{P}_{\texttt{S1}}) = V_1$.

States of M_{Imp} via predicate abstraction. Recall the first key principle behind predicate abstraction: each state q of M_{Imp} models a set of execution states $\hat{q}$ of *proc*. The correspondence between q and $\hat{q}$ is obtained using predicate abstraction and can be expressed as follows. First, every execution state in $\hat{q}$ corresponds to the same CFA state, i.e.

$$\forall e_1 \in \hat{q}, \forall e_2 \in \hat{q} \; . \; \mathbf{CL}(e_1) = \mathbf{CL}(e_2)$$

Let cl_q be the CFA state that every element of $\hat{q}$ corresponds to. Let $\mathcal{P}_q$ denote the set of predicates associated with cl_q. Second, every execution state in $\hat{q}$ assigns the same valuation to $\mathcal{P}_q$, i.e.

$$\forall e_1 \in \hat{q}, \forall e_2 \in \hat{q} \; . \; \mathbf{PV}(e_1, \mathcal{P}_q) = \mathbf{PV}(e_2, \mathcal{P}_q)$$

Let V_q be the unique predicate valuation mandated by the above definition. Thus, the elements of $\hat{q}$ are indistinguishable on the basis of cl_q and V_q. The state q can therefore be viewed as arising from a combination of a CFA state cl_q and a predicate valuation V_q.

Consider an arbitrary CFA state cl. In combination with $\mathcal{V}_{cl}$, cl gives rise to a set of states of M_{Imp}, denoted by $\mathcal{IS}_{cl}$. More precisely, the definition of $\mathcal{IS}_{cl}$ consists of the following sub-cases:

- The FINAL CFA state has no outgoing transitions and must yield the STOP state of M_{Imp}.

$$\mathcal{IS}_{\text{FINAL}} = \{\text{STOP}\}$$

- If *Stmt*(cl) is an assignment, branch, `goto` or `return` statement, then it combines with each possible predicate valuation to yield a state of M_{Imp}.

$$\mathcal{IS}_{cl} = \{cl\} \times \mathcal{V}_{cl}$$

- Suppose *Stmt*(cl) is a call-site for library routine `lib`. It can yield two types of states of M_{Imp}, i.e.

$$\mathcal{IS}_{cl} = \mathcal{IS}^1_{cl} \cup \mathcal{IS}^2_{cl}$$

where the first type arises from the combination with the corresponding predicate valuations. In other words:

$$\mathcal{IS}^1_{cl} = \{cl\} \times \mathcal{V}_{cl}$$

But $\mathcal{IS}^1_{cl}$ can only model execution states *before* `lib` is actually executed. In order to model the execution states *during* the execution of `lib`, we must use the assumption PA for `lib`. In particular, let $\langle g_1, P_1 \rangle, \ldots, \langle g_n, P_n \rangle$ be the guard and LTS list in the assumption PA for `lib`. For $1 \leq i \leq n$, let $\hat{Q}_i$ be the set of states of P_i. Then:

$$\mathcal{IS}^2_{cl} = \bigcup_{i=1}^{n} (\{cl\} \times \mathcal{V}_{cl} \times \hat{Q}_i)$$

In addition, M_{Imp} has an unique initial state INIT. In the rest of this article we shall refer to M_{Imp} states of the form (cl, V) as *normal* states. Also we shall call M_{Imp} states of the form (cl, V, a) as *inlined* states since these state arise due to inlining of assumption PAs at call-sites.

Fact 2 *A set of execution states E is said to be consistent with an expression p iff p evaluates to* TRUE *under every element of E. Let q be an arbitrary M_{Imp} state of the form (cl, V) or (cl, V, a). Then $\hat{q}$, i.e., the set of execution states modeled by q, is consistent with $\Gamma_{\mathcal{P}_{cl}}(V)$.*

Definition of M_{Imp}. Formally, M_{Imp} is an LTS $\langle Q_I, init_I, Act_S, T_I \rangle$ where:

- $Q_I = (\cup_{cl \in CL}\, \mathcal{IS}_{cl}) \cup \{\text{INIT}\}$
- $init_I = \text{INIT}$

Weakest precondition. Consider a C assignment statement *stmt* of the form $v = e$. Let φ be a pure C expression ($\varphi \in Exp$). Then the weakest precondition of φ with respect to *stmt*, denoted by $\mathcal{WP}[stmt]\{\varphi\}$ is obtained from φ by replacing every occurrence of v in φ with e[4].

Fact 3 *Let $q \equiv (cl, V)$ be an arbitrary M_{Imp} state and α be an arbitrary assignment statement. Let $\hat{q}'$ be the predecessors of $\hat{q}$ under α. Then $\hat{q}'$ is consistent with $\mathcal{WP}[\alpha]\{\Gamma_{cl}(V)\}$.*

Constructing the transition relation T_I. Recall the second key principle behind predicate abstraction. Let q_1 and q_2 be any two states of M_{Imp} and let $\hat{q_1}$ and $\hat{q_2}$ be the corresponding sets of execution states of *proc*. If there exists $e_1 \in \hat{q_1}$ and $e_2 \in \hat{q_2}$ such that e_1 makes a transition to e_2 during the execution of *proc*, then $(q_1, q_2) \in T_I$. The construction by MAGIC of a T_I that

[4]Note that we need not consider the case where a pointer appears in the LHS of *stmt* since we have disallowed such constructs from appearing in *proc*.

obeys the above condition has been presented in detail elsewhere [13]. Here we only describe some sub-cases of the construction to convey a general idea of the technique.

Branch statement. Suppose q_1 is of the form (cl_1, V_1) and $Stmt(cl_1)$ is an branch with condition *cond*. Let cl_2 be a successor of cl_1, $V_2 \in \mathcal{V}_{cl_2}$ and $q_2 \equiv (cl_2, V_2)$. By Fact 2, $\hat{q_1}$ is consistent with $\Gamma_{cl_1}(V_1)$ and $\hat{q_2}$ is consistent with $\Gamma_{cl_2}(V_2)$. Suppose $Stmt(cl_2)$ is the *then* successor of $Stmt(cl_1)$. Then the set of predecessors of $\hat{q_2}$ is consistent with $(\Gamma_{cl_2}(V_2) \wedge cond)$. Therefore, there *could be* a transition from some execution state $e_1 \in \hat{q_1}$ to some execution state $e_2 \in \hat{q_2}$ if the formulas $\Gamma_{cl_1}(V_1)$ and $(\Gamma_{cl_2}(V_2) \wedge cond)$ are not provably disjoint.

Let ψ_1 and ψ_2 be two first order formulas over the integers. We say ψ_1 and ψ_2 are *admissible* if we cannot prove their disjointness using a theorem prover, and denote this by $(\psi_1 \otimes \psi_2)$. Hence we add a transition $q_1 \xrightarrow{\tau} q_2$ iff $(\Gamma_{cl_1}(V_1) \otimes (\Gamma_{cl_2}(V_2) \wedge cond))$. Similarly, if $Stmt(cl_2)$ is the *else* successor of $Stmt(cl_1)$, we add a transition $q_1 \xrightarrow{\tau} q_2$ iff $(\Gamma_{cl_1}(V_1) \otimes (\Gamma_{cl_2}(V_2) \wedge \neg cond))$. In general checking admissibility is known to be undecidable. However for our purposes it is sufficient that the theorem prover be sound and always terminate. MAGIC has been integrated with several theorem provers with this characteristic, such as Simplify [38], SVC, CVC, CVC-Lite, ICS and CProver.

Assignment statement. Suppose q_1 is of the form (cl_1, V_1) and $Stmt(cl_1)$ is an assignment statement. Let cl_2 be the successor CFA state of cl_1, $V_2 \in \mathcal{V}_{cl_2}$ and $q_2 \equiv (cl_2, V_2)$. Again, by Fact 2, $\hat{q_1}$ is consistent with $\Gamma_{cl_1}(V_1)$ and $\hat{q_2}$ is consistent with $\Gamma_{cl_2}(V_2)$. Further, by Fact 3, the set of predecessors of $\hat{q_2}$ under the assignment $Stmt(cl_1)$ is consistent with $\mathcal{WP}[Stmt(cl_1)]\{\Gamma_{cl_2}(V_2)\}$. Therefore, we add a transition $q_1 \xrightarrow{\tau} q_2$ iff $(\Gamma_{cl_1}(V_1) \otimes \mathcal{WP}[Stmt(cl_1)]\{\Gamma_{cl_2}(V_2)\})$.

Call-site. Suppose q_1 is of the form (cl_1, V_1) and $Stmt(cl_1)$ is a call-site for library routine `lib`. Let $\langle g_1, P_1\rangle, \ldots, \langle g_n, P_n\rangle$ be the guard and LTS list in the assumption PA for `lib`. A guard g_i is applicable at q_1 iff $(\Gamma_{cl_1}(V_1) \otimes g_i)$. For each such applicable g_i we inline the corresponding LTS P_i at q_1. The details of this inlining procedure have been presented elsewhere [13]. The other sub-cases involved in the construction of T_I proceed in a similar manner.

Example 6 *Recall the CFA from Example 3 and the predicates corresponding to CFA nodes discussed in Example 4. The M_{Imp}'s obtained with $\mathcal{P} = \emptyset$ and $\mathcal{P} = \{(y < 10), (y > 5)\}$ are shown in Figure 9(a) and 9(b) respectively. Note that 9(a) is not simulated by* `Spec` *while 9(b) is simulated by* `Spec`.

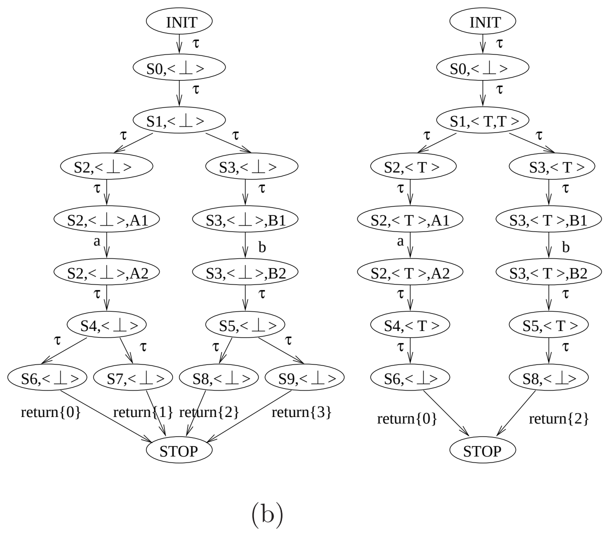

Figure 9: (a) example M_{Imp} with $\mathcal{P} = \emptyset$; (b) example M_{Imp} with $\mathcal{P} = \{(y < 10), (y > 5)\}$.

7 Refinement

In this section we describe the refinement phase in the context of MAGIC. Once a CETree $CE = (N, E, r, \mathrm{St}, \mathrm{Ac})$ has been constructed, we need to perform two steps: (i) check if *CE* is a valid CETree i.e. whether it represents an actual behavior of *proc*, and (ii) if *CE* is invalid then refine M_{Imp} so as to eliminate *CE* from future iterations. We now describe these two steps in more detail.

Checking validity of *CE*. The general strategy of the validity check resembles the method for tree-like counterexamples [21]. First, we traverse the strategy tree bottom-up, and for each node n we compute a verification condition (VC) for n. More precisely, the computed VC for n is an expression X_n over the variables of *proc* with the following property: an execution state e can play the sub-strategy rooted at n iff X_n evaluates to TRUE under e. The VC is computed by algorithm **VerCon** (cf. Figure 10). **VerCon** uses some notation that we explain next.

Consider an arbitrary node n of *CE*. Recall that $\mathrm{St}(n)$ is a game position (q_I, q_S) where q_I and q_S are states of M_{Imp} and M_{Spec} respectively. Furthermore, q_I itself can be a normal state of the form (cl, V) or an inlined state of the form (cl, V, a) where cl is a CFA state, V is a predicate valuation and a is a state of an inlined LTS. The terms $\mathsf{Stmt}(q_I)$ and $\mathsf{Stmt}(n)$ will denote the statement $\mathit{Stmt}(cl)$. Also if q_I is an inlined state, the terms $\mathsf{InSt}(q_I)$ and $\mathsf{InSt}(n)$ will denote a. If q_I is a normal state, $\mathsf{InSt}(q_I)$ and $\mathsf{InSt}(n)$ will evaluate to STOP.

Suppose $\mathsf{Stmt}(n)$ is a branch statement and the challenge $\mathrm{Ch}(n)$ is $(q_I, q_S) \xrightarrow{\alpha} (q'_I, ?)$. Then $\mathsf{Stmt}(q'_I)$ must be either the *then* or *else* successor of $\mathsf{Stmt}(n)$. In the former case we say that $\mathsf{Stmt}(n)$ is a *positive* branch, while in the latter case we say that it is a *negative* branch.

Suppose $\mathsf{Stmt}(n)$ is a call-site for `lib` and the challenge $\mathrm{Ch}(n)$ is $(q_I, q_S) \xrightarrow{\alpha} (q_I', ?)$. First, suppose that q_I is of the form (cl, V). This means that q_I models an execution state before the invocation of `lib`. Also q_I' must be of the form (cl, V, a) where a is the initial state of an LTS P such that the pair $\langle g, P \rangle$ appears in the assumption PA for `lib`. In such a case we call n and *entrance* state and use $\mathsf{IG}(n)$ to denote the guard g.

The other possibility is that q_I is of the form (cl, V, a). If q_I' is of the form (cl, V, a'), it models a state during the execution of `lib`. In such a case we call n an *intermediate* state. Otherwise q_I' models a state after the call to `lib` has terminated. If the value returned from `lib` is stored in some variable z, we call n an *exit to z* state and use $\mathsf{Ret}(n)$ to denote the set of return values associated with return actions enabled at a, i.e., $\mathsf{Ret}(n) = \{RetVal(\alpha) \mid \alpha \in IntRet \wedge a \xrightarrow{\alpha} \mathrm{STOP}\}$. Otherwise if the value returned from `lib` is ignored, we call n a *void exit* state.

```
Algorithm VerCon($N, E, r, \mathrm{St}, \mathrm{Ch}, n$);
$X$ := TRUE ;
if $\mathrm{Child}(n) \neq \emptyset$
    $X := \bigwedge_{c \in \mathrm{Child}(n)}$ VerCon($N, E, r, \mathrm{St}, \mathrm{Ch}, c$);
    // conjunction of conditions on subtrees;
case $\mathsf{Stmt}(n)$ of
    goto: return $X$;
    assignment: return $\mathcal{WP}[\mathsf{Stmt}(n)]\{X\}$;
    positive branch with condition $c$: return $X \wedge c$;
    negative branch with condition $c$: return $X \wedge \neg c$;
    call-site: case $n$ of
        entrance state: return $X \wedge \mathsf{IG}(n)$;
        intermediate state: return $X$;
        exit to $z$ state: return $\bigvee_{r \in \mathsf{Ret}(n)} \mathcal{WP}[z := r]\{X\}$;
        void exit state: return $X$;
    esac;
esac;
```

Figure 10: Algorithm **VerCon** computes the set of execution states from which it is possible to play the sub-strategy rooted at n.

Refining M_{Imp}. Refining M_{Imp} involves the construction of an improved set of predicates $\mathcal{P}_{new}$ that can eliminate the spurious CETree *CE*. This is followed by model construction using $\mathcal{P}_{new}$. Suppose the set of branches appearing in *CE* is $\{b_1, \ldots, b_k\}$ and let the associated branch conditions be $\sigma = \{c_1, \ldots, c_k\}$. We iterate through the subsets of σ in some non-decreasing order of size. For each such subset $\mathcal{P}'$, we check if $\mathcal{P}'$ can eliminate *CE*. The first $\mathcal{P}'$ that successfully eliminates *CE* is taken to be $\mathcal{P}_{new}$.

Given a candidate set of predicates $\mathcal{P}'$ we can check whether it eliminates *CE* as follows: we first construct a model M_{Imp}' from *proc* using $\mathcal{P}'$ and then check if M_{Imp}' can play a simulation game with the specification S according to the strategy described by *CE*. Then, $\mathcal{P}'$ eliminates *CE* iff M'_{Imp} cannot play according to *CE*.

Given an arbitrary state q of $M'_{Imp} = \langle Q'_I, init'_I, Act_S, T'_I \rangle$ and an arbitrary node n of *CE*, algorithm **CanPlay** (cf. Figure 11) returns TRUE iff M'_{Imp} can play the sub-strategy of *CE* rooted at n by starting from its own state q. Thus a top-level call with $q = init'_I$ and $n =$ root of *CE* will help us determine if $\mathcal{P}'$ can eliminate *CE*. Intuitively, q can play n if it can make the same challenge as n, and by doing so can go to some successor state q' that can play all the sub-strategies (i.e., strategies represented by the subtrees) of n. Additionally, q must correspond to the same control flow point and inlined LTS state as n.

The above procedure for refining M_{Imp} is straightforward and has been presented here for the sake of simplicity. In practice, MAGIC uses a more sophisticated technique based on predicate minimization [14]. This more involved approach also uses the algorithm **CanPlay** presented here to check if a set of predicates can eliminate a spurious counterexample.

```
Algorithm CanPlay($M_{Imp}, q, CE, n$);
// $q$ is a state of $M_{Imp}$ and $n$ is a node of $CE$
if (Stmt($q$) $\neq$ Stmt($n$)) $\vee$ (InSt($q$) $\neq$ InSt($n$))
    return FALSE
$(q_I, q_S) \xrightarrow{\alpha} (q'_I, ?) :=$ Ch($n$)
Find $q' \in$ Succ($q, \alpha$) such that:
    $\forall c \in$ Child($n$) . CanPlay($M_{Imp}, q', CE, c$)
If found $q'$, return TRUE
Else return FALSE
```

Figure 11: Algorithm **CanPlay** returns TRUE iff M_{Imp} can play the sub-strategy of *CE* rooted at n by starting from its own state q.

8 Experiments

We have implemented the game semantics based refinement approach within the MAGIC tool. As the source code for our experiments we used OpenSSL-0.9.6c, an open source implementation of the SSL protocol used for secure exchange of information over the Internet. In particular, we used the code that implements the server side of the initial *handshake* involved in SSL. The total size of the source code input to MAGIC was about 74,000 lines, out of which all but 350 lines were eliminated by MAGIC via an initial property dependent slicing. All our experiments were carried out on an AMD Athlon 1600 XP machine with 900 MB RAM running RedHat 7.1. Our experiments 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 CETrees per iteration leads to up to 25% improvement in performance. However, using more than four CETrees is counterproductive.

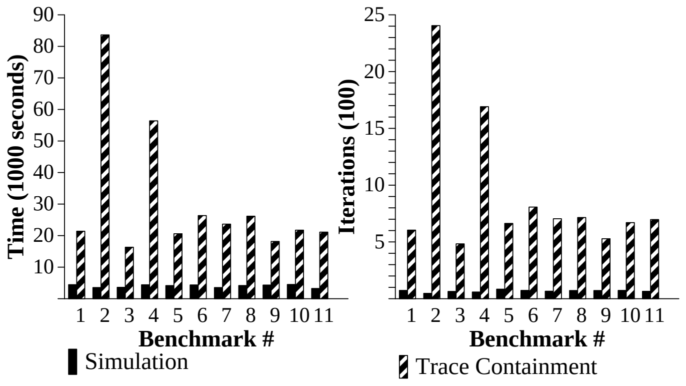

Figure 12: Comparison between simulation and trace containment in terms of time and number of iterations.

We manually designed a set of eleven specifications by reading the SSL documentation. Each of these specifications were 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 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 12. 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 CETree to create a more precise model of *proc*. It is possible that in each iteration of the CEGAR loop, we use not one but a set of CETrees **CE**. In other words, we first generate **CE** and then iterate algorithm *NewPred* over the elements of **CE**. Using our benchmarks, we investigated the effect of increasing the size of **CE**. In particular, the measurements for total time were obtained as follows. The size of **CE** was varied from 1 to 15 and for each value of |**CE**|, 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 13 summarize the results we obtained. The figures indicate that

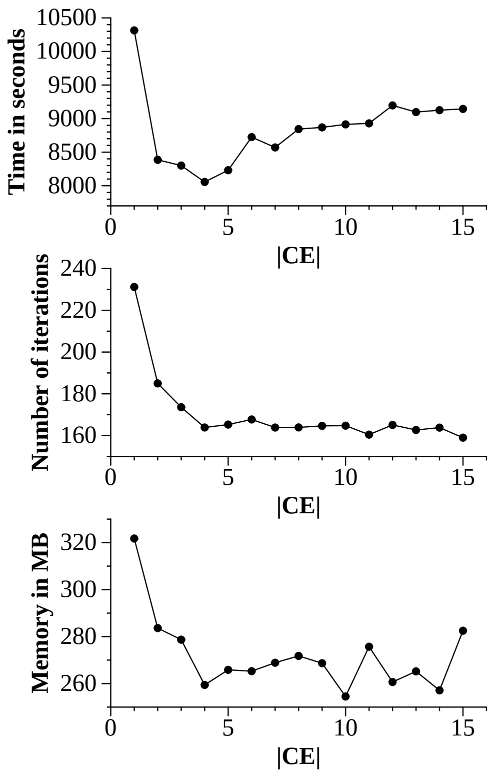

Figure 13: Time, iteration and memory requirements for different number of CETrees.

it makes sense to refine on multiple counterexamples. We note that there is consistent improvement in all three metrics up to $|\mathbf{CE}| = 4$. Increasing $|\mathbf{CE}|$ beyond four appears to be counterproductive.

9 Conclusion

In this paper, we presented a game-based refinement technique in the context of software verification. We implemented these techniques in the software verification tool MAGIC [13]. Experimental results clearly demonstrate the efficiency of the game-based refinement. There are several interesting directions to pursue. We want to explore whether game-based refinement can be used in other software verification systems, such as MC [31] and SLAM [10]. Since we handle calls by inlining, we are unable to handle recursion. We will explore techniques to extend our analysis to be fully inter-procedural. Finally, we want to extend our framework to handle multi-threaded and concurrent programs.

References

[1] S. Abramsky. Sematics of interaction: an introduction to game semantics. In *Semantics and Logics of Computation*, pages 1–32. Cambridge University Press, 1997.

[2] S. Abramsky, R. Jagadesan, and P. Malacaria. Full abstraction for PCF. *Information and Computation*, 163:409–470, 2000.

[3] Samson Abramsky, Dan R. Ghica, Andrzej S. Murawski, , and C.-H. Luke Ong. Applying game semantics to compositional software modeling and verification. In *Proc. of TACAS*, 2004.

[4] K. Altisen, G. Gossler, A. Pnueli, J. Sifakis, S. Tripakis, and S. Yovine. A framework for scheduler synthesis. In *RTSS: Real-Time Systems Symposium*, pages 154–163, 1999.

[5] R. Alur, L. de Alfaro, T.A. Henzinger, and F.Y.C. Mang. Automating modular verification. In *CONCUR: Concurrency Theory*, pages 82–97, 1999.

[6] R. Alur and T.A. Henzinger. Modularity for timed and hybrid systems. In *CONCUR: Concurrency Theory*, pages 74–88, 2001.

[7] L.O. Anderson. *Program analysis and specialization for the C programming language*. PhD thesis, Datalogisk Intitut, Univ. of Copenhagen, Copenhagen, Denmark, 1994.

[8] Giorgio Ausiello and Giuseppe F. Italiano. On-line algorithms for polynomially solvable satisfiability problems. *Journal of Logic Programming*, 10(1,2,3 & 4):69–90, January 1991.

[9] Thomas Ball, Rupak Majumdar, Todd D. Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 203–213, 2001.

[10] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. *Lecture Notes in Computer Science*, 2057, 2001.

[11] BLAST. *http://www-cad.eecs.berkeley.edu/∼rupak/blast.*

[12] J.R. Buchi and L.H. Landweber. Solving sequential conditions by finite-state strategies. *Transactions of the AMS*, 138:295–311, 1962.

[13] Sagar Chaki, Edmund Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components in C. In *Proc. of International Conference on Software Engineering (ICSE)*, pages 385–395, 2003.

[14] Sagar Chaki, Edmund Clarke, Alex Groce, and Ofer Strichman. Predicate abstraction with minimum predicates. In *Proc. of Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)*, 2003.

[15] Sagar Chaki, Edmund Clarke, Orna Grumberg, Joël Ouaknine, Natasha Sharygina, Tayssir Touili, and Helmut Veith. An expressive verification framework for state-event systems. Submitted.

[16] Sagar Chaki, Edmund Clarke, Joël Ouaknine, and Natasha Sharygina. Automated, compositional and iterative deadlock detection. In *Proc. of Second ACM-IEEE International Conference on Formal Methods and models for Codesign (MEMOCODE)*, 2004. To appear.

[17] Sagar Chaki, Edmund Clarke, Joël Ouaknine, Natasha Sharygina, and Nishant Sinha. State/event-based software model checking. In *Proc. of Fourth International Conference on Integrated Formal Methods (IFM)*, 2004.

[18] E. Clarke and E. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Proc. of Workshop on Logic of Programs*, pages 52–71, 1981.

[19] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, 2000.

[20] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794, 2003.

[21] Edmund Clarke, Somesh Jha, Yuan Lu, and Helmut Veith. Tree-like counterexamples in model checking. In *Proc. of Logic in Computer Science (LICS)*, 2002.

[22] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[23] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[24] Satyaki Das, David L. Dill, and Seungjoon Park. Experience with predicate abstraction. In *Computer Aided Verification*, pages 160–171, 1999.

[25] L. de Alfaro, T.A. Henzinger, and F.Y.C. Mang. Detecting errors before reaching them. In *CAV: Computer-Aided Verification*, pages 186–201, 2000.

[26] 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 *International Conference on Software engineering*, pages 177–187, 2001.

[27] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Symposium on Operating Systems Design and Implementation*, 2000.

[28] ESC-Java. *http://www.research.compaq.com/SRC/esc.*

[29] R. Ghica and G. McCusker. Reasoning about idealized Algol using regular languages. In *International Colloquium and Automata, Languages, and Programming (ICALP)*, pages 103–116, 2000.

[30] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Computer Aided Verification*, volume 1254, pages 72–83. Springer Verlag, 1997.

[31] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson Engler. A system and language for building system-specific, static analyses. In *SIGPLAN Conference on Programming Language Design and Implementation*, 2002.

[32] T. A. Henzinger, R. Jhala, and R. Majumdar. Counterexample-guided control. In *ICALP: 30th International Colloquium on Automata, Languages, and Programming*, 2003.

[33] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy abstraction. In *Symposium on Principles of Programming Languages*, pages 58–70, 2002.

[34] Java PathFinder. *ase.arc.nasa.gov/visser/jpf.*

[35] MAGIC. *http://www.cs.cmu.edu/∼chaki/magic.*

[36] μC/OS-II. *http://www.ucos-ii.com.*

[37] Robin Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[38] Greg Nelson. *Techniques for Program Verification.* PhD thesis, Stanford University, 1980.

[39] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 179–190. ACM Press, 1989.

[40] S.K. Rajamani. *New Directions in Refinement Checking.* PhD thesis, University of California, Berkeley, 1999.

[41] P.J. Ramadge and W.M. Wonham. Supervisory control of a class of discrete-event processes. *SIAM Journal of Control and Optimization*, 25:206–230, 1987.

[42] Sandeep K. Shukla. *Uniform Approaches to the Verification of Finite State Systems.* PhD thesis, SUNY, Albany, 1997.

[43] SLAM. *research.microsoft.com/slam*.

[44] Colin Stirling. The Joys of Bisimulation. *Proc. of Mathematical Foundations of Computer Science (MFCS)*, LNCS 1450:142–151, 1998.