Modular Verification of Software Components in C*

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

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

Helmut Veith
TU Vienna
veith@dbai.tuwien.ac.at

Abstract

We present a new methodology for automatic verification of C programs against finite state machine specifications. Our approach is compositional, naturally enabling us to decompose the verification of large software systems into subproblems of manageable complexity. The decomposition reflects the modularity in the software design. We use weak simulation as the notion of conformance between the program and its specification. Following the abstract-verify-refine paradigm, our tool MAGIC first extracts a finite model from C source code using predicate abstraction and theorem proving. Subsequently, simulation is checked via a reduction to Boolean satisfiability. MAGIC is able to interface with several publicly available theorem provers and SAT solvers. We report experimental results with procedures from the Linux kernel and the OpenSSL toolkit.

1 Introduction

State machines have been recognized repeatedly as a cardinal point in the software development process; in fact, variants of state machines have been proposed for virtually all software engineering methodologies, including, most notably, Statecharts [25] and the UML [9]. The sustained success of state machines in software engineering derives from the fact that state machines provide for both a concise mathematical theory, and an intuitive semantics of system behavior which naturally allows for visualization, hierarchy, and abstraction.

Traditionally, state machines are mainly used in the design phase of the software life-cycle; they are intended to guide and constrain the implementation and the test phase, and may later be reused for documentation purposes. In most cases, however, the assertion that a state machine safely abstracts the existing implementation is kept implicit and informal.

With the rise of Internet-based technologies, the significance of state machines has only increased. In particular, security protocols and communication protocols are naturally specified in terms of state machines. Similar applications of state machines can be found in other safety-critical domains including medicine and aerospace.

Moreover, the dramatic change of focus from relatively monolithic systems to highly distributed and heterogeneous systems whose development cycles are interdependent, calls for new specification methodologies; for example, in August 2002, IBM, Microsoft, and BEA announced the publication of three specifications (WS-Coordination, WS-Specification, BPEL4WS [3]) which "collectively describe how to reliably define, create and connect multiple business processes in a Web services environment". We foresee state machines being used for *contracts* describing software capabilities. In both cases – protocol specification and distributed computation – we observe that state machines are no longer just tools for internal use, but are increasingly introduced into the public domain.

In this paper, we describe our tool MAGIC (**M**odular **A**nalysis of pro**G**rams **I**n **C**) which is capable of verifying whether a state machine (or, more precisely, a labeled transition system) is a safe abstraction of a C procedure; the C procedure in turn may invoke other procedures *which are themselves specified in terms of state machines.*

Our approach has a number of tangible benefits:

- *Utility.* The capability of MAGIC to verify formally the correctness of state-machine specifications closes an evident gap in many software development methodologies, most notably, but not only, for security-related system features. In the future we envision that tools based on ideas from MAGIC will assist the contracting process with third party software providers.

- *Compositionality.* MAGIC verification can be used early on during the development cycle, as specifica-

*This research was supported by the NRL-ONR under Grant No. N00014-01-1-0796, by the NSF under Grant No. CCR-9803774, CCR-0121547 and CCR-0098072, by the Army-ARO under Grant No. DAAD 19-01-1-0485, the Austrian Science Fund Project NZ29-INF, the EU Research and Training Network GAMES and graduate student fellowships from Microsoft and NSF. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or the United States Government.

tions can be plugged in for missing system components. Compositionality evidently fosters concurrent development by independent groups of developers.

- *Complexity.* State-space explosion [18] remains the bottleneck of most automated verification tools. Due to compositionality, the size of the individual system parts to be verified by MAGIC remains manageable, as demonstrated by our experiments. Moreover, the verification process in MAGIC is reduced to computing a *simulation relation between finite state systems*, for which we can provide highly efficient algorithms.

- *Flexibility.* Internally, MAGIC uses several theorem provers and SAT solvers. The open design of MAGIC facilitates the easy integration of new and improved tools from this quickly developing area.

Consequently, we believe that MAGIC like tools have the potential to become indispensable in the software engineering process. In the rest of this section we describe the technical contributions of this paper.

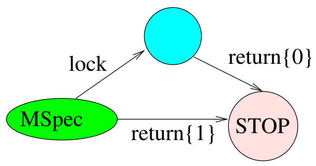
Labeled Transition Systems as Specification Mechanism. In the literature, several variants of state machines have been investigated; purely state-based formalisms such as Kripke structures [18] are often used to model and specify systems. For the MAGIC framework, however, we employ *labeled transition systems* (LTS), which are similar to Kripke structures but for the fact that state transitions are labeled by *actions*.

From a theoretical point of view the presence of actions does not increase the expressive power of LTS. In our experience, however, it is more natural for designers and software engineers to express the desired behavior of systems using a combination of states and actions. For example, the fact that a lock has been acquired or released can be expressed naturally by *lock* and *unlock* actions. In the absence of actions, the natural alternative is to introduce a new variable indicating the status of the lock, and update it accordingly. The LTS approach certainly is more intuitive, and allows both for a simpler theory and for an easier specification process. A simple example of an LTS is shown in the left part of Figure 1. A formal definition will be given in Section 2.

In the MAGIC framework, we use actions to denote externally visible behaviors of the system being analyzed, e.g. acquiring a lock. Actions are atomic, and are distinguished simply by their names. Often, the presence of an action indicates a certain behavior which is achieved by a subprocedure in the implementation. Since we shall analyze a procedural language, namely C, we will 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 a is associated with a unique return value $RetVal(a)$. Return values are either integers or *void*. All actions which are not return actions are called *basic* actions.

The use of LTSs is also motivated by work in concurrency. Process algebras like CCS [33], CSP [28] and the π-calculus [34] have been used widely to reason formally about message passing concurrent systems. In these formalisms, actions are crucial for modeling the sending and receiving of messages across channels. Process algebras lead very naturally to LTSs. Thus, even though we currently only analyze sequential programs, we believe that the use of LTSs will facilitate a smooth transition to concurrent message-passing programs in the future.

Procedure Abstractions. The goal of MAGIC is to verify whether the implementation of a system is safely abstracted by its specification. To this end, MAGIC verifies individual procedures against the respective LTS. In our implementation, it is possible to handle a group of procedures with a tree-like call graph as a single one by inlining; for simplicity, we speak only of single procedures in this paper. Figure 1 describes a simple case of a procedure *proc* and a corresponding LTS. We will use *proc* as a running example.




```
int proc()
{
  if(do_lock())
    return 0;
  else
    return 1;
}
```

Spec State Machine

Code to be analysed

Figure 1. The example M_{Spec} and $proc$.

In practice, it often happens that single procedures perform quite different tasks for certain settings of their parameters. In our approach, this phenomenon is accounted for by allowing multiple LTSs to represent a single procedure. The selection among these LTSs is achieved by *guards*, i.e., formulas, which describe the conditions on the procedure parameters under which a certain LTS is applicable.

This gives rise to the notion of *procedure abstraction* (PA); formally a PA for a procedure $proc$ is a tuple $\langle d, l\rangle$ where

- d is the declaration for $proc$, as it appears in a C header file.

- 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 of $proc$, and each M_i is an LTS with a single initial state.

The procedure abstraction 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* passed to *proc*. We require that the guard formulas g_i be mutually exclusive so that the choice of M_i is unambiguous. The goal of MAGIC then is to prove that a user-defined PA for *proc* is valid. The role of PAs in this process is twofold:

1. A *target PA* is used to describe the desired behavior of the procedure *proc*.

2. To assist the verification process, we employ valid PAs (called the *assumption* PAs) for library routines used by *proc*.

Thus, PAs can be seen both as conclusions and as assumptions of the verification process. Consequently, our methodology yields a scalable and compositional approach for verifying large software systems. Figure 2 illustrates this by depicting the call graph of an implementation and the verification steps; note that due to compositionality no particular order of these steps is required.

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}. To achieve the result in full generality, the described algorithm can be iterated for each guard of M_{Spec}.

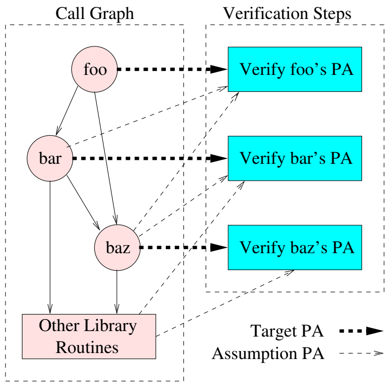


Figure 2. Compositional verification.

Algorithms and Tool Description. The MAGIC tool follows the well-known *abstract - verify - refine* paradigm [13, 16, 21, 27]:

- **Step 1 : Model Creation.** Extract an LTS M_{Imp} from *proc* using the assumed PAs and the guard G_{Spec}.

  In MAGIC, the model is computed from the control flow graph (CFG) of the program in combination with an abstraction method called *predicate abstraction* [16, 19, 37]. To decide properties such as equivalence of predicates, we use theorem provers. The details of this step are described in Section 3.

- **Step 2 : Verification.** Check whether M_{Spec} safely abstracts M_{Imp}. If this is the case, the verification successfully terminates; otherwise, extract diagnostic feedback and perform step 3.

  In MAGIC, the verification step amounts to checking whether a *simulation relation* holds between M_{Spec} and M_{Imp}, cf. Section 2. We reduce simulation to the satisfiability of a certain Boolean formula, thus deferring the solution to highly efficient SAT procedures. The details of this step are described in Section 2.

- **Step 3 : Refinement.** Use the diagnostic feedback to determine the reason behind the failure of the safe abstraction property. If the cause is a bug in *proc* we are done. Otherwise the property fails because M_{Imp} is not a sufficiently precise model for *proc*. In this case return to step 1 to compute an improved M_{Imp}.

At its current stage of development, MAGIC performs the first two of these steps automatically, while the third step is manually guided. The input to MAGIC consists of (i) a set of preprocessed ANSI-C files representing *proc* and (ii) a set of specification files containing textual descriptions of M_{Spec}, G_{Spec} and *predicates* for predicate abstraction. The textual descriptions of LTSs are given using an extended version of the FSP notation by Magee and Kramer [30]. For example, the LTS shown in Figure 1 is described textually as follows:

```
MyLock = ( lock -> return {$0 == 0} -> STOP
         | return {$0 == 1} -> STOP ).
```

The schematic in Figure 3 explains the software architecture of MAGIC. **Model Creation** is handled by Stage I and II of the program. In Stage I the input files are parsed and the control flow graph (CFG) of the C program is constructed. Simplifications are made so that the resulting CFG only has simple statements and side-effect free expressions. Then relevant predicates at each control location are computed and the CFG is annotated with them. In Stage II, M_{Imp} is extracted from the annotated CFG using the assumed PAs, G_{Spec} and the predicates. As described later, this process requires the use of theorem provers. MAGIC can interact with several public domain theorem provers viz. Simplify [36], CVC [39], ICS [23] and CPROVER [29].

Verification is performed in Stage III. As mentioned above, simulation here is reduced to Boolean satisfiability. MAGIC can interface with several publicly available SAT solvers viz. Chaff [35], FGRASP [31] and SATO [40]. We also have our own efficient SAT solver implementation which leverages the specific nature of SAT formulas that arise in this stage to deliver better performance than the public domain solvers. MAGIC does not generate diagnostic feedback yet, nor does it support automatic model

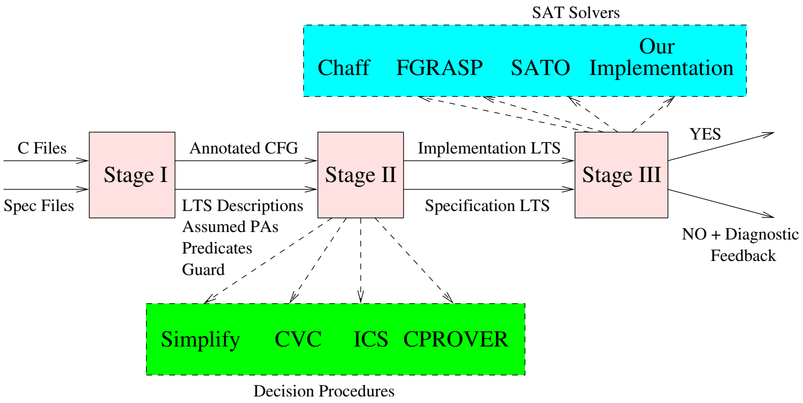


Figure 3. Overall architecture of MAGIC.

refinement. We consider this a significant area for future research.

Related Work. During the last years advances in verification methodology as well as in computing power have promoted renewed interest in software verification. The resulting systems – most notably Bandera [1] and Java PathFinder [5, 26], ESC Java [4], SLAM [7], BLAST [2] and MC [22, 24] – are increasingly able to handle industrial software. Among the six mentioned systems, the former three focus on Java, while the latter three all deal with C. Java verification is quite different from C, because object orientation, garbage collection and the logical memory model require specific analysis methods. Among the C verification tools, MC (which stands for meta-compilation) has a distinguished place because it amounts to a form of pattern matching on the source code, with surprisingly good results for scanning relatively simple errors in large amounts of code. SLAM and BLAST are closely related tools, whose technical flavor is most akin to ours. SLAM is primarily optimized to analyse device drivers, and is going to be included in the Windows development cycle. In contrast to SLAM which uses symbolic algorithms, BLAST is an on-the-fly reachability analysis tool. MAGIC is the only tool which uses LTS as specification formalism, and simulation as the notion of conformance. This choice reflects the area of security currently being our primary application domain.

Except for MC, the mentioned tools are based on variations of *model checking* [15, 18], and they all require abstraction methods to alleviate the state explosion problem, most notably data abstraction [17] and the more general predicate abstraction [37]. The abstraction method used in SLAM and BLAST is closest to ours. However, due to compositionality, we can afford to invest more computing power into computing abstractions, and are therefore able to improve on Cartesian abstraction [12]. Generally, we believe that the form of compositionality provided by MAGIC is unique among existing software verification systems.

Virtually all systems using abstraction interface with theorem provers for various purposes. The software architecture of MAGIC is designed as to facilitate the integration of various theorem provers. In addition, MAGIC is the only tool in this area which attempts to transfer the enormous success of SAT procedures in hardware verification [14] to software.

2 Transition Systems and Simulation

A labeled transition system (LTS) M is a 4-tuple (S, S_0, Act, T), where (i) S is a finite non-empty set of states, (ii) $S_0 \subseteq S$ is the set of initial states, (iii) Act is the set of actions, and (iv) $T \subseteq S \times Act \times S$ is the transition relation.

We assume that there is a distinguished state $STOP \in S$ which has no outgoing transitions, i.e., $\forall s' \in S, \forall a \in A, (STOP, a, s') \notin T$. In addition we assume the presence of a distinguished action in the set Act, which we denote by ϵ. If $(s, a, s') \in T$, then (s, s') will be referred to as a a-transition and will be denoted by $s \xrightarrow{a} s'$. If s is reachable from s' via zero or more ϵ-transitions, we will denote this by $s \xrightarrow{\epsilon^\star} s'$. The relation $\Rightarrow$ is defined as follows: $s \overset{a}{\Rightarrow} s'$ iff there exist s_1 and s_2 such that $s \xrightarrow{\epsilon^\star} s_1 \xrightarrow{a} s_2 \xrightarrow{\epsilon^\star} s'$.

Conformance via Simulation. In the context of LTS, *simulation* [33] is the natural notion of conformance between a specification LTS and an implementation LTS. We will therefore use simulation as our notion of conformance between the specification LTS and the program. Compared to conformance notions based on trace containment [13], simulation has the additional advantage that it is computationally less expensive to check. Among the many technical variants of simulation [33], we choose *weak simulation* because it allows for a limited form of asynchrony between the LTSs, i.e., one step of the specification LTS may simulate multiple steps of the implementation. This feature of weak simulation is crucial to our approach, because one step in M_{Spec} typically corresponds to multiple steps in M_{Imp}.

Weak Simulation. Let $M = (S, S_0, Act, T)$ and $M' = (S', S'_0, Act, T')$ be two LTSs. A relation $E \subseteq S \times S'$ is called a *weak simulation* between M and M' iff (i) for all $s \in S_0$ there exists $s' \in S'_0$ such that $(s, s') \in E$, and (ii) $(s, s') \in E$ implies that for all actions $a \in Act \setminus \{\epsilon\}$ if $s \overset{a}{\Rightarrow} s_1$, then there exists $s'_1 \in S'$ such that $s' \overset{a}{\Rightarrow} s'_1$ and $(s_1, s'_1) \in E$. We say that LTS M' *weakly simulates* M (denoted by $M \preceq M'$) if there exists a weak simulation relation $E \subseteq S \times S'$ between M and M'. In the rest of the paper, we use the convention that the terms *simulation* and *simulates* will always mean *weak simulation* and *weakly simulates* respectively.

Algorithm for Computing Weak Simulation. The existence of a simulation relation between M and M' can be checked efficiently by reducing the problem to an instance of Boolean satisfiability [38]. Interestingly the SAT instances produced by this method always belong to a restricted class of SAT formulas known as the *weakly negated* HORN formulas. In contrast to general SAT (which has no known polynomial time algorithm), satisfiability of weakly negated HORN formulas can be solved in linear time [20]. As part of MAGIC, we have implemented an online linear time HORNSAT algorithm based on [10]. MAGIC can also interface with public domain general SAT solvers like Chaff [35], FGRASP [31] and SATO [40].

3 Model Creation

Let $M_{Spec} = (S_{Spec}, S_{0,Spec}, Act_{Spec}, T_{Spec})$ and the assumption PAs be $\{PA_1, \ldots, PA_k\}$. In this section we show how to extract M_{Imp} from *proc* using the assumption PAs, the guard G_{Spec} and the predicates. The extraction of M_{Imp} relies on several principles:

- Every state of M_{Imp} models a state during the execution of *proc*; consequently every state is composed of a control component and a data component.

- The control components intuitively represent values of the program counter, and are formally 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.

- The transitions between states in M_{Imp} are derived from the transitions in the control flow graph, taking into account the assumption PAs and the predicate abstraction. This process involves reasoning about C expressions, and will therefore require the use of a theorem prover.

In the rest of this section, we will describe these steps in detail.

Control Flow Graph. The CFG of *proc* is a finite graph describing the flow of control in *proc*. The nodes of the CFG are called control locations, and intuitively correspond to the values of the program counter; the edges denote transfer of control. Ordinary C code however contains nested procedure calls, expressions with side-effects and other similar constructs that make it difficult to construct precise CFGs. In order to alleviate this problem, our tool first performs a set of natural simplifications on *proc* before constructing its CFG. The simplified procedure body contain only *normal assignments* (e.g. `x = y + 5;` or `*x = *y + 10;`), *call assignments* (e.g. `x = foo(y + 5);`), *branches* (e.g. `if (x) { ... } else { ... }`), *gotos* and *returns* (e.g. `return (*y + 5); `). The left hand side of assignments must always be either a variable or a single address dereference (such as `*v = 5;`). Note that in the resulting program loop statements such as `while` and `for` are substituted by appropriate `if` and `goto` statements. Moreover, we can assume that each variable has a unique scope, and each procedure always terminates with explicit `return` statements. These preprocessing steps are not very complicated, and are omitted here. The CFG for our example *proc* is shown in Figure 4.

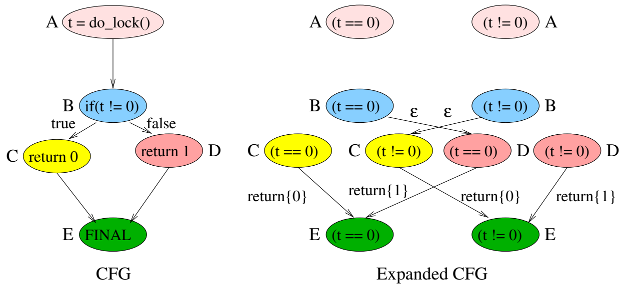


Figure 4. CFG and expanded CFG.

Control Locations. After this simplification, the definition of control locations becomes straightforward: Each normal assignment, call assignment, goto and return statement gives rise to a control location with a unique successor. In contrast, a branch yields a control location with exactly two successors. We assume that for each control location in the CFG, the associated conditions and statements are available. In addition, we introduce a unique final control location and make it the unique successor of all return control locations. Depending on the statement to which the control location is referring we speak of normal assignment locations, branch locations etc. Formally, $\mathcal{C}$ denotes the set of control locations of $proc$.

Expanding the Control Flow Graph. The CFG is the simplest reasonable finite model of $proc$. However, for verification purposes the CFG is too imprecise because it models only control flow, but ignores data (memory). On the other hand, it is computationally unfeasible to model the memory explicitly.

We will now show how to incorporate *abstract* memory state information into the CFG. To this end, we will consider a set of properties at each control location. These properties are described by C expressions similar to those used as branching conditions. Thus, if we have k data properties, each of which can be true or false, then each control location corresponds to 2^k possible states in our model, each of them corresponding to a particular valuation of the properties considered. Intuitively, the construction of the LTS M_{Imp} proceeds stepwise:

1. Construct the CFG.

2. Construct an expanded CFG $M_{Exp} = (S_{Exp}, S_{0,Exp}, Act_{Spec}, T_{Exp})$ as follows:

   - For each control location c, we include 2^k states in S_{Exp}; each state thus is a combination of a control location and a valuation of the considered properties.
   - Consider an edge (c_1, c_2) in the CFG. Then c_1 and c_2 correspond to 2^k states in S_{Exp} each. Consequently, (c_1, c_2) may be correspond to up to $2^k \times 2^k$ transitions in T_{Exp}. However, not all of them will be included because not all transitions are consistent with the abstract memory state information. We will use a theorem prover to determine which of these transitions indeed are admissible. We will only rule out transitions whose admissibility can be disproved by the theorem prover.

3. M_{Exp} is a more precise model of $proc$ than the CFG. However it does not model the behavior of the library routines called by $proc$. To achieve this we incorporate the assumption PAs into M_{Exp}. This step also requires the use of a theorem prover. The LTS obtained after incorporating the assumption PAs is M_{Imp}.

In the following two sections, we will describe step 2 of this process in detail; in Section 3.4, we will explain step 3.

3.1 Predicate Abstraction

Predicate abstraction is an approach to model abstractly the state of a system by a set of logical predicates [19, 37]. We use predicate abstraction techniques only to model the memory state. As we aim to verify C programs, it is natural to express these properties by pure (side-effect free) Boolean C expressions. Since we assumed the scope of each variable to be unique, this definition is unambiguous.

Thus, we shall use C expressions very much in the same way as quantifier-free first order formulas. Because of this conceptual proximity we will use logical connectives such as $\wedge$, $\vee$ and $\neg$ instead of their C equivalents `&&`, `||` and `!`.

In order to describe abstract memory states, let us fix a certain set $\mathcal{P} = \{P_1, \dots P_k\}$ of expressions which we call the *predicates*. Note that we do not use float and string constants in predicates at the current stage of the implementation. Given a concrete memory state m and a predicate P, we say that m satisfies P iff P evaluates to true (i.e., a nonzero numerical value) during the execution of *proc* when the memory state is m. A valuation for $\mathcal{P}$ is a vector $v_1, \dots v_k$ of Boolean values, such that v_i expresses the Boolean value of P_i. $\mathcal{V}$ denotes the set of all valuations, i.e., the set of abstract memory states. Intuitively, a concrete memory state m is modeled by $v_1, \dots v_k$ if for $1 \le i \le k$, m satisfies P_i iff v_i is true.

A valuation v typically models many concrete memory states. This set is characterized by a formula (called the *concretization* of v) which expresses truth or falsity of the involved properties in the natural way: Given a valuation $v = v_1, \dots v_k$, the concretization $\gamma(v)$ is defined as $\bigwedge_{i=1}^{k} P_i^{v_i}$ where $P_i^{v_i}$ is equal to P_i if v_i is true, and equal to $\neg P_i$ if v_i is false. Thus, $\gamma(v)$ describes the property captured by the valuation v; all memory states which are modeled by v satisfy $\gamma(v)$.

Example 1 *In our example, $\mathcal{P}$ contains a single predicate $(t == 0)$ and therefore has two valuations - [true] and [false]. Hence $\gamma([true]) := (t == 0)$ and $\gamma([false]) := \neg(t == 0)$. Thus [true] models all concrete memory states where the variable* t *is equal to* 0 *and false models all concrete memory states where the variable* t *is not equal to* 0*.*

State Space of M_{Exp}. We combine the control flow graph and the predicate abstraction to obtain the state space $S_{Exp} := \mathcal{C} \times \mathcal{V}$ of M_{Exp}. Thus a state of M_{Exp} is a pair $\langle c, v\rangle$ where $c \in \mathcal{C}$ and $v \in \mathcal{V}$. It models all execution states of *proc* where the control location is c and where the memory state is modeled by v. In Section 3.3 we show how to compute the transitions between states in S_{Exp}; in Section 3.4 we show how to extend M_{Exp} to incorporate the specification PAs. Before we can do this, we need to give some details about our use of theorem provers.

3.2 Application of Theorem Provers

We will use theorem provers to reason about C expressions. Since these expressions may involve integer as well as Boolean arithmetic, and, importantly, pointer arithmetic, the logic involved is quite complicated, and certainly undecidable.[1] It is therefore important to our approach that we use the theorem prover *conservatively*, i.e., we only assume logical relationships which the theorem prover actually can prove.

In this section, we will describe the principles of our usage of theorem provers so as to give an intuition of our approach. The logical intricacies involved (including, for example, uninterpreted function symbols to model the heap) are handled by our tool, but would exceed the scope of this paper, and are therefore omitted.

For the construction of the abstract transition relation it will often be necessary to determine whether two C expressions e_1 and e_2 are mutually exclusive. To this end, we use the theorem prover to compute a meta-predicate $\mathcal{A}(e_1, e_2)$ with the following properties:

- If $\mathcal{A}(e_1, e_2)$ is false, then e_1 and e_2 are provably mutually exclusive.
- If $\mathcal{A}(e_1, e_2)$ is true, then this indicates that the theorem prover could not prove that e_1 and e_2 are mutually exclusive, either because they indeed are not mutually exclusive, or because proving mutual exclusiveness was beyond the capabilities of the theorem prover.

The meta-predicate $\mathcal{A}$ has a crucial role in the definition of the transition relation. We will see that the definition of $\mathcal{A}$ ensures that the abstraction is safe. We illustrate the use of $\mathcal{A}$ by the following important example:

Given any C expression e_1 and a normal C assignment s, we define the *weakest precondition* of e_1 with respect to s in the same way as [11] and denote it by $\mathcal{WP}(s, e_1)$. Intuitively, $\mathcal{WP}(s, e_1)$ is a C expression which denotes the weakest assumption that has to be true *before* the execution of s in order for e_1 to become true *after* the execution of s. Given s and e_1, $\mathcal{WP}(s, e_1)$ can be computed as follows:

- If s is the assignment statement $v = e_2$ then $\mathcal{WP}(s, e_1)$ is obtained from e_1 by replacing all occurrences of v in e_1 with e_2.
- If s is an assignment statement of the form $*v = e_2$, then we have to take into account aliasing possibilities as well. For example if e_1 is the expression $a == 5$ then $\mathcal{WP}(s, e_1)$ is $((v == \&a) \wedge (e_2 == 5)) \vee ((v! = \&a) \wedge (a == 5))$.

[1] It follows from the famous negative solution to Hilbert's tenth problem [32] that checking the equivalence of two C expressions is undecidable even for simple integer arithmetic; even when we restrict the range of the variables to 32 bit, the question is computationally very hard.

Let $Asgn$ be the set of normal C assignments in P. The relation $Update \subseteq \mathcal{V} \times Asgn \times \mathcal{V}$ denotes how normal assignments affect the valuations, and is defined as follows.

$$Update = \{(v_1, s, v_2) \mid \quad \mathcal{A}(\gamma(v_1), \mathcal{WP}(s, \gamma(v_2))), \\ v_1, v_2 \in \mathcal{V}, s \in Asgn\}$$

Intuitively $Update(v_1, s, v_2)$ means that if *proc* is in a memory state modeled by v_1, and the assignment statement s is executed, then we need to admit the possibility that a memory state abstracted by v_2 can be reached.

3.3 Transition Relation T_{Exp}

We are now ready to define the transition relation $T_{Exp} \subseteq S_{Exp} \times Act_{Spec} \times S_{Exp}$ for M_{Exp}.

Consider any two states s and s' of M_{Exp}. We have already seen that state $s = (c, v)$ models all concrete states of *proc* whose control component is c and whose data component is modeled by v. Let t and t' denote two concrete states modeled by s and s' respectively. If there is a concrete transition for any such t and t' then we must include a transition (s, s') in T_{Exp}. This approach guarantees that M_{Exp} is a sound model of *proc*.

The rest of this section describes a procedure to decide whether to include such a transition (s, s') in T_{Exp} or not. We will make a case distinction by the type of c. If c is a final location, then there are no outgoing transitions from s. Otherwise c can be of five different types and we consider each type separately.

Goto. Let c' be the unique successor control location of c. Then we include $((c, v), \epsilon, (c', v))$ in T_{Exp}. Thus we change the control state according to the program flow and keep the data state unaltered.

Normal Assignment. Let z be the assignment statement at c and c' be the unique successor control location of c. For every valuation v' such that $Update(v, z, v')$, we include $((c, v), \epsilon, (c', v'))$ in T_{Exp}. Thus we let the control state change according to the control flow of the program and allow any data state change not disproved by the theorem prover.

Branch. Recall that in the CFG, every branch has two successors. Let e be the branching condition, and let c'_T and c'_E be the *true* and *false* successors of c. If $\mathcal{A}(\gamma(v), e)$, then we include $((c, v), \epsilon, (c'_T, v))$ in T_{Exp}. If $\mathcal{A}(\gamma(v), \neg e)$, then we include $((c, v), \epsilon, (c'_E, v))$ in T_{Exp}. Thus we allow for any successor control state that is not provably impossible and we keep the data state unchanged.

Return. Let e be the return expression at the return location c in *root* and c' be the unique successor location of c. Note that c' must be the final location. For all return actions $a \in Act_{Spec}$, if $\mathcal{A}(\gamma(v), (e == RetVal(a)))$ (i.e., if the return value described by the action is possibly equal to the value actually returned) then we include $((c, v), a, (c', v))$ in T_{Exp}. If there is no return action $a \in Act_{Spec}$, then we include $((c, v), \epsilon, (c', v))$ in T_{Exp}.

Call Assignment. Since we do not incorporate the specification PAs at this stage, we do not include any transitions originating at call assignment locations. These transitions will be explained in Section 3.4.

Initial States. The initial states are those states which are consistent with the guard G_{Spec}. Thus, $S_{0,ECFG}$ is the set of states (c, v) where $\mathcal{A}(\gamma(v), G_{Spec})$ and c is the initial location of CFG.

Example 2 *The expanded CFG for our example is shown in Figure 4. Since there are two valuations, there are two states corresponding to every control location in the actual CFG. The corresponding control locations and states in the M_{Exp} have the same color and are marked with the same letter. In addition every state corresponding to valuation v is marked with $\gamma(v)$ ($\neg(t == 0)$ is written $(t! = 0)$).)*

3.4 Inlining the Specification PAs

In this section, we show how to conclude the construction of M_{Imp} by incorporating the assumption PAs into M_{Exp}. Recall that the purpose of this step is to model calls made by *proc* to library routines. Intuitively, this is achieved by inline copies of appropriate LTSs between call assignment locations and their respective successors in the CFG.

M_{Imp} is obtained from M_{Exp} by adding new states and transitions: Consider a state (c, v) where c is a call assignment, and let c' be the unique successor of c in the CFG. Let $\mathtt{x} = \mathtt{lib}(\ldots)$; be the call assignment statement at c. Assume that *proc* is not a function pointer; we will deal with this special case later. 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 each i, we do the following:

1. Let g'_i be the guard obtained from g_i by replacing every parameter of *lib* by the corresponding argument passed to it at c. If $\mathcal{A}(g'_i, \gamma(v))$, then proceed, otherwise move on to the next guard.

2. Let $P_i = (S_i, Act_i, \{s_{0,i}\}, T_i)$. For each state $s \in S_i$ which is not *STOP*, introduce a new state $(c \cdot s, v)$ into S_{Imp}. These states represent the inlined states of P_i.

3. Add a transition $((c, v), \epsilon, (c \cdot s_{0,i}, v))$ into T_{Imp}. This transition connects the call location state to the initial inlined state.

4. For each transition $(s, a, t) \in T_i$ where t is different from STOP, add a transition $((c \cdot s, v), a, (c \cdot t, v))$ into T_{Imp}.

5. For each transition $(s, a, \text{STOP}) \in T_i$ where a is *not* a return action, and for each v' such that $\mathcal{A}(\gamma(v), \gamma(v'))$ is true, add $((c \cdot s, v), a, (c', v'))$ into T_{Imp}.

6. For each transition $(s, a, \text{STOP}) \in T_i$ where a is a return action, and for each v' such that $Update(v, x = RetVal(a), v')$ is true, add $((c \cdot s, v), \epsilon, (c', v'))$ into T_{Imp}.

If *lib* is a function pointer, then we repeat the construction described above for each possible target of *lib* listed by the user.

Example 3 *The assumption LTS for* `do_lock` *is shown at the top of Figure 5. The M_{Imp} obtained by incorporating the LTS for* `do_lock` *into M_{Exp} of Figure 4 is shown at the bottom. The corresponding states in M_{Exp} and M_{Imp} are colored identically. Similarly the states of the LTS for* `do_lock` *and the corresponding inlined states in M_{Imp} have identical colors. It is clear that M_{Imp} is simulated by the M_{Spec} in Figure 1.*

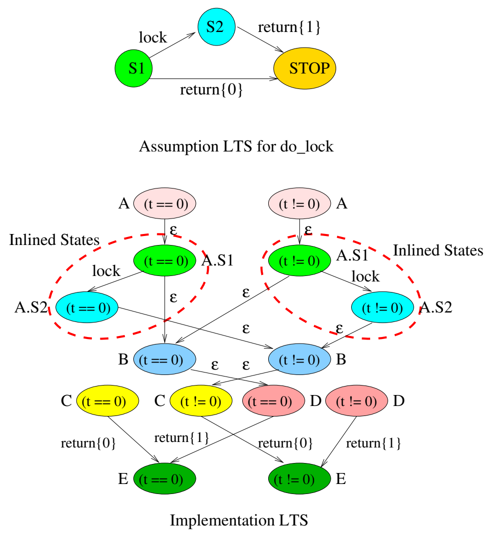


Figure 5. LTS for `do_lock` **and M_{Imp}.**

3.5 Enhancements and Implementation Issues

We now describe several enhancements to the above described basic framework that we have implemented in

MAGIC, but have been omitted to keep the presentation simple.

Making predicate abstraction more efficient. The set of valuations $\mathcal{V}$ is exponential in the number of predicates in $\mathcal{P}$. MAGIC also uses the theorem prover to group together predicates that are mutually exclusive. Since at most one predicate in such a group can be true at any time the number of possible valuations of that group is equal to the size of the group. This reduces the size of the state space dramatically. For example suppose we had four predicates originally and we formed two groups of two predicates each. Then the number of possible valuations reduces from sixteen to four.

Even though we have assumed a fixed set of predicates $\mathcal{P}$ in the above discussion, MAGIC allows different sets of predicates. Since not all predicates are *useful* to be abstracted at all control locations, using them indiscriminately would be inefficient. A similar method has been used by BLAST [27].

Automatic predicate discovery. The effectiveness of predicate abstraction relies critically on the set of predicates. The model extraction process described above requires that the predicates be supplied externally. However, if directed, MAGIC can also try to discover suitable sets of predicates. We do not discuss the full details of this predicate discovery process in this paper. However in almost all of our experiments MAGIC manages to automatically discover good predicate sets and correctly prove/disprove the simulation property with them.

Interfacing with theorem provers. As explained before, during the construction of M_{Imp} we use efficient theorem provers. We have integrated MAGIC with various publicly available theorem provers. In some cases, like Simplify [36], CVC [39] and ICS [23], the version of the software available to us can only be used via command line arguments. We run them as separate processes and interface with them via their standard inputs and outputs. In other cases, like CPROVER [29], the software is available as libraries with well-defined APIs and we link them directly with MAGIC. Also in all the cases we cache results to avoid redundant calls to the theorem prover.

4 Case Studies

Our experiments were guided by three general goals: First, we wanted to assure the correctness of the tool by experimenting with examples where the correct outcome was already known. Second, we wanted to evaluate the relative performances of various publicly available software (theorem provers, SAT solvers) that were integrated into our system. Third, we wished to validate the usefulness of our tool in handling large real life examples.

Regression Tests. The first two goals were achieved by a suite of 10 regression tests of small size. All these tests were derived from actual Linux kernel code. Figure 6 describes the source of each test briefly. LOC indicates the number of post-processed lines of C. The name of the procedure analysed is given in italics in the description. A modified procedure means that the source code was changed so that it would no longer be safely abstracted by the specification LTS. The library to which the procedure belongs is given in brackets after the procedure name.

Regression	LOC	Description
lock-y	27	*pthread_mutex_lock* (pthread)
unlock-y	24	*pthread_mutex_unlock* (pthread)
socket-y	60	*socket* (socket)
sock_alloc-y	24	*sock_alloc* (socket)
sys_send-y	4	*sys_send* (socket)
sock_sendmsg-y	11	*sock_sendmsg* (socket)
lock-n	27	modified *pthread_mutex_lock*
unlock-n	24	modified *pthread_mutex_unlock*
sock_alloc-n	24	modified *sock_alloc*
sock_sendmsg-n	11	modified *sock_sendmsg*

Figure 6. Descriptions of regression tests.

Verifying OpenSSL. To achieve the third goal we opted to work with OpenSSL [6], an open source implementation of the publicly available SSL [8] specification. This protocol is used by a client (typically a web browser) and a server to establish a secure socket connection over a malicious network using public and symmetric key cryptography.

A critical component of the protocol is the *handshake*. First we verified that the *openssl-0.9.6c* implementation of the server side of the handshake conforms to its specification. This implementation is encapsulated in a single procedure of about 347 lines of C. We constructed the target LTS M_{Spec} manually by reading the SSL specification [8]. The LTS had 28 states and 67 transitions. A total of 19 predicates and PAs for 14 library routines were supplied externally. We carried out two experiments. The first was done with the correct target LTS. The second was done with a modified the target LTS (of same size) so that a correct implementation would no longer be simulated by it. Next we repeated identical experiments with the client side implementation. It was encapsulated within a single procedure of 345 lines. The target LTS had 28 states and 60 transitions. A total of 18 predicates and PAs for 12 library routines were supplied externally.

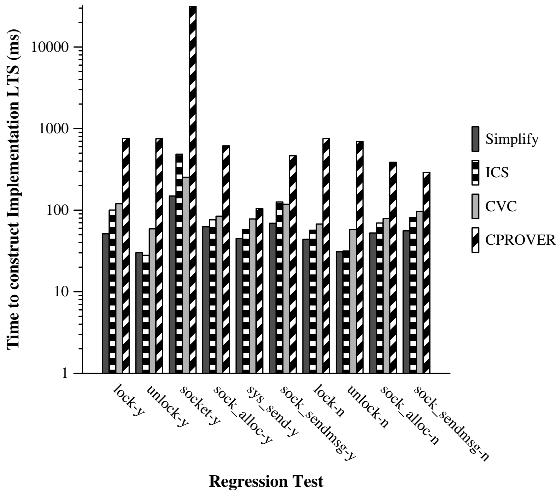


Figure 7. Time to construct M_{Imp}.

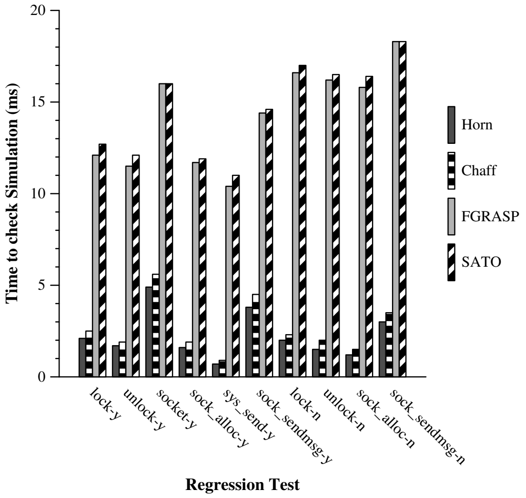


Figure 8. Time to check simulation.

Regression Test Results. All our experiments were done on a 1.4 GHz AMD Athlon machine with 1 GB of RAM running RedHat Linux 7.1. Figure 7 summarizes the performance results for various theorem provers obtained via the regression suite. The y-axis (drawn in log scale) shows the time needed to construct M_{Imp} in milliseconds which is a clear indicator of the performance of the theorem prover. Similarly, Figure 8 summarizes the performance results for various SAT solvers obtained via the regression suite. The y-axis indicates the time in milliseconds needed to check simulation since this is the step where the SAT solver is used.

OpenSSL Results. In the case of the OpenSSL server experiments, the fact that the correct specification LTS safely abstracts the OpenSSL implementation was then proved by our tool in 255 seconds using about 130 MB of memory. The tool also successfully verified that the modified specification LTS does not safely abstract the implementation in 247 seconds using 115 MB of memory. For the client experiments the corresponding figures were 226 seconds, 107MB and 227 seconds, 111MB. Owing to compositionality we did not have to verify the validity of the assumption PAs used for these experiments.

Comparison of Theorem Provers and SAT Tools. A closer look at the two bar graphs reveal several consistent trends. First, for the purposes of our tool, the theorem provers can be arranged in decreasing order of efficiency as follows: Simplify, ICS, CVC and CPROVER. The first three theorem provers have comparable efficiency and seem clearly superior to CPROVER. Second, the SAT solvers can also be arranged in decreasing order of efficiency as follows: Horn, Chaff, FGRASP and SATO. Of the external solvers we used Chaff seems to be easily the best, almost matching our own HORNSAT based implementation. FGRASP and SATO are less easily distinguishable.

The difference in performance between general SAT solvers and the HORNSAT solver we implemented becomes prominent for the larger OpenSSL example. The time required for checking simulation for the first OpenSSL server experiment and the first OpenSSL client experiment were 42 seconds and 32 seconds respectively when using our HORNSAT solver. In comparison the same figures for Chaff were 386 seconds and 265 seconds respectively.

Negative Results. The reported figures were obtained using user supplied predicates. When we repeated the experiments using automatically discovered predicates, higher execution times were observed. The reason is that our automatic predicate discovery process yields more predicates than are necessary. This leads to a larger number of states in M_{Imp} and hence to greater execution times. We believe that improving the predicate discovery technique is a good area for further research.

5 Future Work

There is enormous potential for extending the basic framework implemented by MAGIC. In conclusion we list notable areas for future research: (i) generation of diagnostic feedback and automatic model refinement, (ii) abstraction techniques for more precise modeling of the heap, (iii) extending the MAGIC infrastructure to OO languages like Java and C++, (iv) handling concurrency, and (v) automatic predicate discovery.

References

[1] Bandera. *http://www.cis.ksu.edu/santos/bandera*.

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

[3] Business Process Execution Language for Web Services. *http://www.oasis-open.org/cover/bpel4ws.html*.

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

[5] Java PathFinder. *http://ase.arc.nasa.gov/visser/jpf*.

[6] OpenSSL. *http://www.openssl.org*.

[7] SLAM. *http://research.microsoft.com/slam*.

[8] SSL 3.0 Specification. *http://wp.netscape.com/eng/ssl3*.

[9] Unified Modeling Language. *http://www.uml.org*.

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

[11] T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic predicate abstraction of c programs. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 203–213, 2001.

[12] T. Ball, A. Podelski, and S. K. Rajamani. Boolean and Cartesian abstraction for model checking C programs. *Lecture Notes in Computer Science*, 2031:268–283, 2001.

[13] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. *Lecture Notes in Computer Science*, 2057, 2001.

[14] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. *Lecture Notes in Computer Science*, 1579:193–207, 1999.

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

[16] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, pages 154–169, 2000.

[17] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and System (TOPLAS)*, 16(5):1512–1542, September 1994.

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

[19] S. Das, D. L. Dill, and S. Park. Experience with predicate abstraction. In *Computer Aided Verification*, pages 160–171, 1999.

[20] W. F. Dowling and J. H. Gallier. Linear time algorithms for testing the satisfiability of propositional horn formula. *Journal of Logic Programming*, 3:267–284, 1984.

[21] M. B. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. S. Pasareanu, H. Zheng, and W. Visser. Tool-supported program abstraction for finite-state verification. In *International Conference on Software engineering*, pages 177–187. IEEE Computer Society, 2001.

[22] 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.

[23] J.-C. Filliatre, S. Owre, H. Ruess, and N. Shankar. ICS: Integrated canonizer and solver. In *Computer-Aided Verification*, 2001.

[24] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *SIGPLAN Conference on Programming Language Design and Implementation*, 2002.

[25] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[26] K. Havelund and T. Pressburger. Model checking JAVA programs using JAVA pathfinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[27] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Symposium on Principles of Programming Languages*, pages 58–70, 2002.

[28] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM (CACM)*, 21(8):666–677, August 1978.

[29] D. Kroening. Application specific higher order logic theorem proving. In S. Autexier and H. Mantel, editors, *Proc. of the Verification Workshop - VERIFY'02*, pages 5–15, July 2002.

[30] J. Magee and J. Kramer. *Concurrency: State Models & Java Programs*. Wiley, 2000.

[31] J. P. Marques-Silva and K. A. Sakallah. GRASP – a new search algorithm for satisfiability. In *IEEE/ACM International Conference on Computer-Aided Design*, November 1996.

[32] Y. Matiyasevich. *Hilbert's Tenth Problem*. MIT Press, 1993.

[33] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[34] R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

[35] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient sat solver. In *Design Automation Conference*, June 2001.

[36] G. Nelson. *Techniques for Program Verification*. PhD thesis, Stanford University, 1980.

[37] 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.

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

[39] A. Stump, C. Barrett, and D. Dill. CVC: A cooperating validity checker. In *Conference on Computer-Aided Verification*, 2002.

[40] H. Zhang. SATO: An efficient propositional prover. In *Conference on Automated Deduction*, 1997.