Modular Verification of Software Components in C

Sagar Chaki Edmund Clarke Alex Groce

Carnegie Mellon University

{chaki|emc|agroce}@cs.cmu.edu

Somesh Jha, University of Wisconsin

jha@cs.wisc.edu

Helmut Veith, Technische Universität München

veith@in.tum.de

(Invited Paper)

This research was supported by the ONR under Grant No. N00014-01-1-0796, by the NSF under Grant No. CCR-9803774, CCR-0121547 and CCR-0098072, by the 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.

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 counterexample guided abstraction refinement (CEGAR) paradigm, our tool MAGIC first extracts a finite model from C source code using predicate abstraction and theorem proving. Subsequently, weak 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.

Index Terms

Software Engineering, Formal Methods, Verification.

I. Introduction

State machines have been recognized repeatedly as an important artifact in the software development process; in fact, variants of state machines have been proposed for virtually all software engineering methodologies, including, most notably, Statecharts [1] and the UML [2]. The sustained success of state machines in software engineering stems 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 have been 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 an 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 [3], [4], [5]. 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, on August 2002, IBM, Microsoft, and BEA announced the publication of three specifications (WS-Coordination, WS-Specification, BPEL4WS [6]) 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 being introduced increasingly into the public domain.

In this paper, we describe our tool MAGIC (**M**odular **A**nalysis of pro**G**rams **I**n **C**) [7] 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 formally verify 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 specifications can be plugged in for missing system components. Compositionality evidently fosters concurrent development by independent groups of developers.
- **Complexity.** State-space explosion [3] 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 *weak 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.

A. *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 [3] 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 over Kripke structures. 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. Some sample LTSs used in our framework are shown in Figure 4. A formal definition will be given in Section III.

The use of LTSs is also motivated by work in concurrency. Process algebras like CCS [8], CSP [9] and the π-calculus [10] have been used widely to formally reason 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.

B. *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 dag-like call graph as a single procedure by inlining; therefore, for simplicity, we speak only of single procedures in this paper.

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.

C. Compositionality

The general goal of MAGIC 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 1 illustrates this by depicting the call graph of an implementation and the steps involved in verifying it. In order to verify `baz` we need only assumption PAs for the other library routines. For `bar` we additionally use the PA for `baz` as an assumption PA while for `foo` we employ the PAs of both `bar` and `baz` as assumptions. Note that due to the sound compositional principles on which MAGIC is based upon, no particular ordering of these verification steps is required.

Assumption PAs are not only important for compositionality, they are in fact essential for handling recursive library routines. Since MAGIC inlines all library routines for which assumption PAs are unavailable, it would be unable to proceed if the assumption PA for a recursive library routine was absent. 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}.

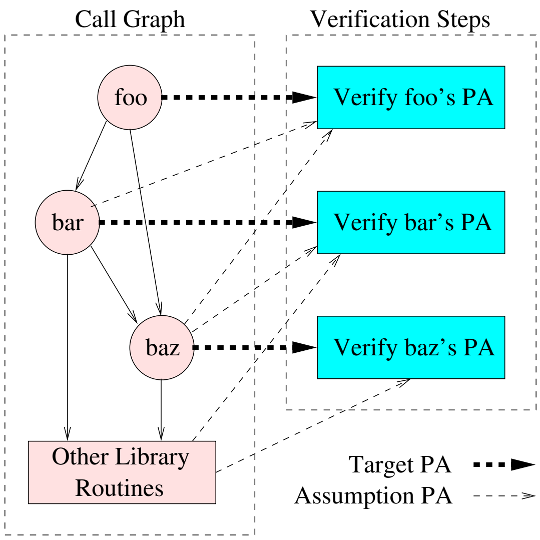

Fig. 1. Example of compositional verification.

D. Algorithms and Tool Description

The MAGIC tool follows the CEGAR paradigm [11], [12], [13], [14] that can be summarized as follows:

- **Step 1 : Model Creation.** Extract an LTS M_{Imp} from *proc* using the assumed PAs, the guard $\mathcal{G}_{Spec}$ and a set of predicates. In MAGIC, the model is computed from the control flow graph (CFG) of the program in combination with an abstraction method called *predicate abstraction* [12], [15], [16]. To decide properties such as equivalence of predicates, we use theorem provers. The details of this step are described in Section IV.
- **Step 2 : Verification.** Check if M_{Spec} safely abstracts M_{Imp}. If this is the case, the verification successfully terminates; otherwise, extract a counterexample and perform step 3. In MAGIC, the verification step amounts to checking whether a *weak simulation relation* (cf. Section III) holds between M_{Spec} and M_{Imp}. We reduce weak simulation to the satisfiability of a certain Boolean formula, thus utilizing highly efficient SAT procedures. The details of this step are described in Section V.

- **Step 3: Validation.** Check whether the counterexample extracted in step 2 is valid. If this is the case, then we have found an actual bug and the verification terminates unsuccessfully. Otherwise construct an explanation for the spuriousness of the counterexample and proceed to Step 4.
- **Step 4 : Refinement.** Use the explanation from the previous step to construct an improved set of predicates. Return to Step 1 to extract a more precise M_{Imp} using the new set of predicates instead of the old one. The new predicate set is constructed in such a way as to guarantee that all spurious counterexamples encountered so far will not appear in any future iteration of this loop.

At its current stage of development, MAGIC can perform all the above steps in an automated manner. 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}, $\mathcal{G}_{Spec}$ and a set of *predicates* for abstraction. The textual descriptions of LTSs are given using an extended version of the FSP notation by Magee and Kramer [17]. For example, the LTS Do_A shown in Figure 4 is described textually as follows:

```
A1 = (a -> A2),
A2 = (return {} -> STOP).
```

E. Tool Overview

The schematic in Figure 2 explains the software architecture of MAGIC. Model Creation is handled by Stage I of the program. In this stage, 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. Finally, M_{Imp} is extracted from the annotated CFG using the assumed PAs, $\mathcal{G}_{Spec}$ and the predicates. As described later, this process requires the use of theorem provers. MAGIC can interact with several public domain theorem provers, such as Simplify [18], CVC [19], ICS [20], CVC Lite [21], and CPROVER [22].

Verification is performed in Stage II. As mentioned above, weak simulation here is reduced to a form of Boolean satisfiability. MAGIC can interface with several publicly available SAT solvers, such as Chaff [23], FGRASP [24] and SATO [25]. We also have our own efficient SAT solver implementation which leverages the specific nature of SAT formulas that arise in this stage to

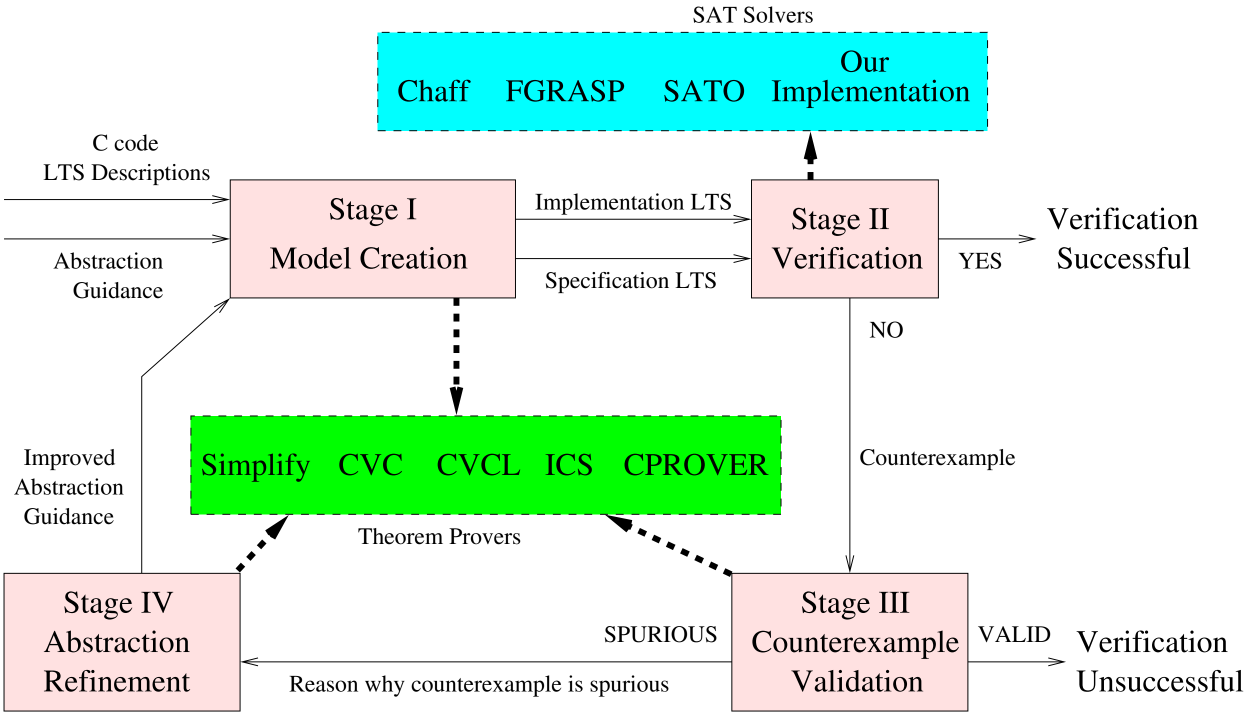

Fig. 2. Overall architecture of MAGIC.

deliver better performance than the public domain solvers. The verification process is presented in Section V in more detail.

If the verification step fails, MAGIC generates an appropriate counterexample and checks its validity in Stage III. If the counterexample is found to be spurious, an improved set of predicates is computed in Stage IV and the entire process is repeated from Stage I. Stages III and IV are completely automated and require the use of theorem provers. In this paper we focus on model creation and verification; details about counterexample validation and abstraction refinement are presented elsewhere [26].

The rest of this paper is organized as follows: In Section II we present related work. This is followed in Section III by some basic definitions that are used in the rest of this article. In Section IV we describe in detail the model construction procedure used in MAGIC to extract LTS models from C programs. Section V describes how we check weak simulation between M_{Spec} and M_{Imp} using Boolean satisfiability. In Section VI we present a broad range of benchmarks and results that we have used to evaluate MAGIC. Finally, in Section VII we give an overview of various ongoing and future research directions that are relevant to MAGIC.

II. 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 [27] and Java PathFinder [28], [29], ESC Java [30], SLAM [31], BLAST [32] and MC [33], [34] – are increasingly able to handle industrial software. Among the six mentioned systems, the first three focus on Java, while the last 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 analyze 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 weak simulation as the notion of conformance. This choice reflects the area of security currently being our primary application domain.

Except for MC and ESC Java, the above-mentioned tools are based on variations of *model checking* [3], [35], and they all require abstraction methods to alleviate the state explosion problem, most notably data abstraction [36] and the more generally predicate abstraction [16]. 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 [37]. Generally, we believe that the form of compositionality provided by MAGIC is unique among existing software verification systems.

Virtually all systems that use 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 which leverages the enormous success of SAT procedures in hardware verification [38] in software verification. SAT procedures have been successfully used for checking validity of software specifications (expressed in a relational calculus) [39], [40], [41].

III. Definitions

In this section we present some basic definitions that will be used in the rest of this article.

A. Labeled Transition Systems

A labeled transition system (LTS) M is a 4-tuple $\langle S, \mathit{init}, \Sigma, T\rangle$, where (i) S is a finite non-empty set of states, (ii) $\mathit{init} \in S$ is the initial state, (iii) Σ is a finite set of actions (alphabet) , and (iv) $T \subseteq S \times \Sigma \times S$ is the transition relation.

We assume that there is a distinguished state $\mathit{STOP} \in S$ which has no outgoing transitions, i.e., $\forall s \in S, \forall a \in \Sigma, (\mathit{STOP}, a, s) \notin T$. We will write $s \xrightarrow{a} t$ to mean $(s, a, t) \in T$ and denote the set $\{t \mid s \xrightarrow{a} t\}$ by $\mathit{Succ}(s, a)$.

B. 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. Often, the presence of an action indicates a certain behavior which is achieved by a sub-procedure in the implementation. Since we are analyzing C, 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 return values are integers by IntRet and the special return action whose return value is `void` by $\mathit{VoidRet}$.

All actions which are not return actions are called *basic actions*. A distinguished basic action τ denotes the occurrence of an unobservable internal event. In this article we 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 $\langle S, \mathit{init}, \Sigma, T\rangle$ must obey the following condition: $\forall s \in S, s \xrightarrow{a}$ STOP iff a is a return action.

C. Conformance via Weak Simulation

In the context of LTS, *simulation* [8] is the natural notion of conformance between a specification LTS and an implementation LTS. Compared to conformance notions based on trace containment [11], simulation has the additional advantage that it is computationally less expensive to check. Among the many technical variants of simulation [8], we choose *weak simulation* as our notion of conformance because it allows for asynchrony between the LTSs, i.e., one step of the specification LTS may correspond to 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}.

D. Weak Simulation

Let $M_1 = \langle S_1, init_1, \Sigma, T_1 \rangle$ and $M_2 = \langle S_2, init_2, \Sigma, T_2 \rangle$ be two LTSs with the same alphabet. A relation $R \subseteq S_1 \times S_2$ is called a *weak simulation* iff if obeys the following two conditions for all $s_1 \in S_1$, $t_1 \in S_1$ and $s_2 \in S_2$:

1) If $(s_1, s_2) \in R$, $a \neq \tau$ and $s_1 \xrightarrow{a} t_1$ then there exists $t_2 \in S_2$ such that $s_2 \xrightarrow{a} t_2$ and $(t_1, t_2) \in R$.
2) If $(s_1, s_2) \in R$ and $s_1 \xrightarrow{\tau} t_1$ then at least one of the following two conditions hold:
   a) $(t_1, s_2) \in R$
   b) There exists $t_2 \in S_2$ such that $s_2 \xrightarrow{\tau} t_2$ and $(t_1, t_2) \in R$

We say that LTS M_2 *weakly simulates* M_1 (denoted by $M_1 \preccurlyeq M_2$) if there exists a weak simulation relation $R \subseteq S_1 \times S_2$ such that $(init_1, init_2) \in R$.

E. Algorithm for Computing Weak Simulation

The existence of a weak simulation relation between M_1 and M_2 can be checked efficiently by reducing the problem to an instance of Boolean satisfiability [42]. 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 [43]. As part of MAGIC, we have implemented an online linear time HORNSAT algorithm [44]. MAGIC can also interface with public domain general SAT solvers like Chaff [23], FGRASP [24] and SATO [25].

IV. MODEL CONSTRUCTION

Let $M_{Spec} = \langle S_{Spec}, init_{Spec}, \Sigma_{Spec}, T_{Spec} \rangle$ 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 $\mathcal{G}_{Spec}$ 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 and 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 therefore requires the use of a theorem prover.

```
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; }
    }
```

Fig. 3. A simple *proc* we use as a running example.

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`. In our implementation, we use the CIL [45] tool to transform arbitrary C programs into the above format. Note that call-sites correspond to library routines called by *proc* for which assumed PAs are available. We assume the absence of indirect function calls and pointer dereferences in the lhs's of assignments.

In reality, MAGIC handles these constructs by using aliasing information conservatively [26]. We denote by $Stmt$ the set of statements of $proc$ and 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 3. It invokes two library routines `do_a` and `do_b`. Let the guard and LTS list in the assumption PA for `do_a` be $\langle \texttt{TRUE}, \texttt{Do\_A} \rangle$. This means that under all invocation conditions, `do_a` is safely abstracted by the LTS `Do_A`. Similarly the guard and LTS list in the assumption PA for `do_b` is $\langle \texttt{TRUE}, \texttt{Do\_B} \rangle$. The LTSs `Do_A` and `Do_B` are described in Figure 4. Also we use $\mathcal{G}_{Spec} = \texttt{TRUE}$ and $M_{Spec} = \texttt{Spec}$ (shown in Figure 4).

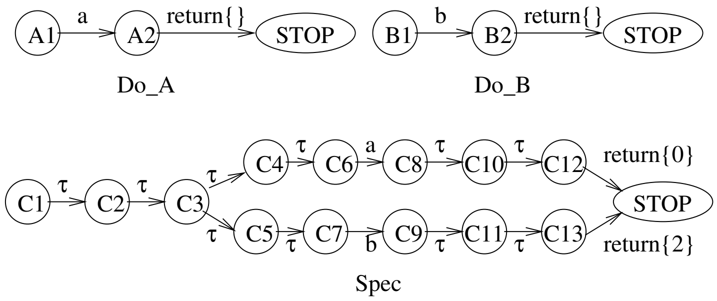

Fig. 4. The LTSs in the assumption PAs for `do_a` and `do_b`. The $VoidRet$ action is denoted by `return{}`.

A. 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 a CFA correspond to control points in the program. The transitions between states in the CFA correspond to the control flow between their associated control points in the program. Thus, a CFA of a program is a conservative abstraction of the program's control flow, i.e. it allows a superset of the possible traces of the program. Formally the CFA is a 4-tuple $\langle S_{CF}, I_{CF}, T_{CF}, \mathcal{L} \rangle$ where:

- S_{CF} is a set of states.
- $I_{CF} \in S_{CF}$ is an initial state.
- $T_{CF} \subseteq S_{CF} \times S_{CF}$ is a set of transitions.
- $\mathcal{L} : S_{CF} \setminus \{\text{FINAL}\} \rightarrow Stmt$ is a labeling function.

S_{CF} contains a distinguished FINAL state. The transitions between states reflect the flow of control between their labeling statements: $\mathcal{L}(I_{CF})$ is the initial statement of $proc$ and $(s_1, s_2) \in T_{CF}$ iff one of the following conditions hold:

- $\mathcal{L}(s_1)$ is an assignment, call-site or `goto` with $\mathcal{L}(s_2)$ as its unique successor.
- $\mathcal{L}(s_1)$ is a branch with $\mathcal{L}(s_2)$ as its `then` or `else` successor.
- $\mathcal{L}(s_1)$ is a `return` statement and $s_2 = \mathrm{FINAL}$.

Example 1: The CFA of our example program is shown in Figure 5. 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 5 are `S0 ...S9, final` with `S0` being the initial state.

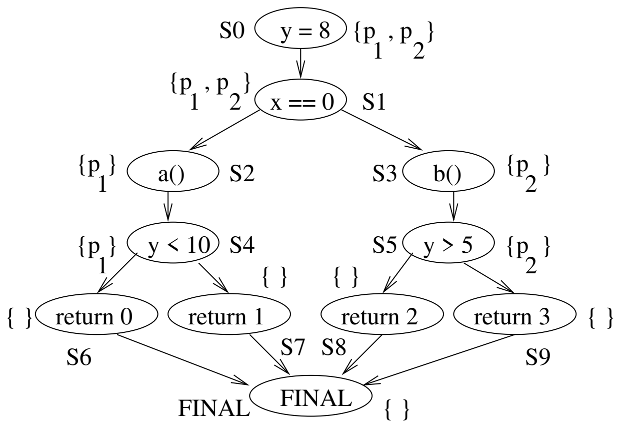

Fig. 5. The CFA for our example program. Each non-FINAL state is labeled the same as its corresponding statement. The initial state is labeled `S0`. The states are also labeled with inferred predicates when $\mathcal{P} = \{p_1, p_2\}$ where $p_1 = (y < 10)$ and $p_2 = (y > 5)$.

B. Predicate inference

Since the construction of M_{Imp} from $proc$ involves predicate abstraction, it is parameterized by a set of predicates $\mathcal{P}$. The main challenge in predicate abstraction is to identify the set $\mathcal{P}$ that is necessary for proving the given property. In our framework we require $\mathcal{P}$ to be a subset of the branch predicates in $proc$. Therefore we sometimes refer to $\mathcal{P}$ or subsets of $\mathcal{P}$ simply as a set of branches. The construction of M_{Imp} associates with each state s of the CFA a finite subset of Exp derived from $\mathcal{P}$, denoted by $\mathcal{P}_s$. The process of constructing the $\mathcal{P}_s$'s from $\mathcal{P}$

is known as *predicate inference* and is described by the algorithm *PredInfer* in Figure 6. Note that $\mathcal{P}_{\text{FINAL}}$ is always $\emptyset$.

The algorithm uses a procedure for computing the *weakest precondition* $\mathcal{WP}$ [11], [46], [47] of a predicate p relative to a given statement. Consider a C assignment statement a of the form $v = e$. Let φ be a pure C expression ($\varphi \in \mathit{Exp}$). Then the weakest precondition of φ with respect to a, denoted by $\mathcal{WP}[a]\{\varphi\}$ is obtained from φ by replacing every occurrence of v in φ with e. Note that we need not consider the case where a pointer appears in the lhs of a since we have disallowed such constructs from appearing in $proc$.

Input: `Set of branch statements` $\mathcal{P}$

Output: `Set of` $\mathcal{P}_s$`'s associated with each CFA state`

Initialize: $\forall s \in S_{CF}, \mathcal{P}_s := \emptyset$

Forever `do`

  For `each` $s \in S_{CF}$ `do`

   If $\mathcal{L}(s)$ `is an assignment statement and` $\mathcal{L}(s')$ `is its successor`

    For `each` $p' \in \mathcal{P}_{s'}$ `add` $\mathcal{WP}[\mathcal{L}(s)]\{p'\}$ `to` $\mathcal{P}_s$

   Else If $\mathcal{L}(s)$ `is a branch statement with condition` c

    If $\mathcal{L}(s) \in \mathcal{P}$`,` **then** `add` c `to` $\mathcal{P}_s$

    If $\mathcal{L}(s')$ `is a 'then' or 'else' successor of` $\mathcal{L}(s)$

     $\mathcal{P}_s$ `:=` $\mathcal{P}_s \cup \mathcal{P}_{s'}$

   Else If $\mathcal{L}(s)$ `is a call-site or a 'goto' statement with`
   `successor` $\mathcal{L}(s')$

    $\mathcal{P}_s$ `:=` $\mathcal{P}_s \cup \mathcal{P}_{s'}$

   Else If $\mathcal{L}(s)$ `returns expression` e `and` $r \in \Sigma_{Spec} \cap \mathit{IntRet}$

    `Add the expression` $(e == \mathit{RetVal}(r))$ `to` $\mathcal{P}_s$

  If `no` $\mathcal{P}_s$ `was modified in the For loop,` **then** `exit`

Fig. 6. The algorithm *PredInfer* that MAGIC uses for predicate inference.

The weakest precondition is clearly an element of *Exp* as well. Note that *PredInfer* may not terminate in the presence of loops in the CFA. However, this does not mean that our approach is incapable of handling C programs containing loops. In practice, we force termination of

PredInfer by limiting the maximum size of any $\mathcal{P}_s$. Using the resulting $\mathcal{P}_s$'s, we can compute the states and transitions of the abstract model as described later. Irrespective of whether *PredInfer* was terminated forcefully or not, M_{Imp} is guaranteed to be a safe abstraction of $proc$. We have found this approach to be very effective in practice. A similar algorithm was proposed by Dams and Namjoshi [48].

Example 2: Consider the CFA described in Example 1. Suppose $\mathcal{P}$ contains the two branches `S4` and `S5`. Then *PredInfer* begins with $\mathcal{P}_{\texttt{S4}} = \{(y < 10)\}$ and $\mathcal{P}_{\texttt{S5}} = \{(y > 5)\}$. From this it obtains $\mathcal{P}_{\texttt{S2}} = \{(y < 10)\}$ and $\mathcal{P}_{\texttt{S3}} = \{(y > 5)\}$. This leads to $\mathcal{P}_{\texttt{S1}} = \{(y < 10), (y > 5)\}$. Then $\mathcal{P}_{\texttt{S0}} = \{\mathcal{WP}[y = 8]\{y < 10\}, \mathcal{WP}[y = 8]\{y > 5\}\} = \{(8 < 10), (8 > 5)\}$. Since we ignore predicates that are trivially TRUE or FALSE, $\mathcal{P}_{\texttt{S0}} = \emptyset$. Since the return actions in `Spec` have return values $\{0, 2\}$, $\mathcal{P}_{\texttt{S6}} = \{(0 == 0), (0 == 2)\}$, which is again $\emptyset$. Similarly, $\mathcal{P}_{\texttt{S7}} = \mathcal{P}_{\texttt{S8}} = \mathcal{P}_{\texttt{S9}} = \mathcal{P}_{\text{FINAL}} = \emptyset$. Figure 5 shows the CFA with each state s labeled by $\mathcal{P}_s$.

C. Predicate valuation and concretization

So far we have described a method for computing the initial abstraction (the CFA) and a set of predicates associated with each location in the program. The states of the abstract system M_{Imp} correspond to the various possible valuations of the predicates in each location. Formally, for a CFA node s suppose $\mathcal{P}_s = \{p_1, \ldots, p_k\}$. Then a *valuation* V of $\mathcal{P}_s$ is a function from $\mathcal{P}_s$ to the set $\{\text{TRUE}, \text{FALSE}\}$. Alternately, one can view the valuation V as a Boolean vector $\langle v_1, \ldots, v_k \rangle$ of size k where each v_i is the result of applying the function V to the predicate p_i.

Let $\mathcal{V}_s$ be the set of all predicate valuations of $\mathcal{P}_s$. Note that the size of $\mathcal{V}_s$ is exponential in the size 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 $\Gamma_s(\perp) = \text{TRUE}$.

Example 3: Consider the CFA described in Example 1 and the inferred predicates as explained in Example 2. 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 $\Gamma_{\texttt{S1}}(V_2) = (y < 10) \wedge (\neg(y > 5))$.

D. States of M_{Imp}

Each state $s \in S_{CF}$ gives rise to a set of states of M_{Imp}, denoted by $\mathcal{IS}_s$. In addition, M_{Imp} has an unique initial state INIT. The definition of $\mathcal{IS}_s$ consists of the following sub-cases:

- $\mathcal{IS}_{\text{FINAL}} = \{\text{STOP}\}$.
- If $\mathcal{L}(s)$ is an assignment, branch, `goto` or `return` statement, then $\mathcal{IS}_s = \{s\} \times \mathcal{V}_s$.
- Suppose $\mathcal{L}(s)$ is a call-site for library routine `lib` and $\langle g_1, P_1\rangle, \ldots, \langle g_n, P_n\rangle$ is the guard and LTS list in the assumption PA for `lib`. For $1 \leq i \leq n$, let $P_i = (S_i, s_{\{0,i\}}, Act_i, T_i)$. Then:

$$\mathcal{IS}_s = (\cup_{i=1}^{n}(\{s\} \times \mathcal{V}_s \times S_i)) \cup (\{s\} \times \mathcal{V}_s)$$

In the rest of this article we shall refer to M_{Imp} states of the form (s, V) as *normal* states. Also we shall call M_{Imp} states of the form (s, V, c) *inlined* states since these states can be thought of as arising due to inlining of assumed PAs at call-sites.

E. Definition of M_{Imp}

Formally, M_{Imp} is an LTS $\langle S_{Imp}, init_{Imp}, \Sigma_{Spec}, T_{Imp}\rangle$ where:

- $S_{Imp} = \cup_{s \in S_{CF}} \mathcal{IS}_s \cup \{\text{INIT}\}$ is the set of states.
- $init_{Imp} = \text{INIT}$ is the initial state.
- $T_{Imp} \subseteq S_{Imp} \times \Sigma_{Spec} \times S_{Imp}$ is the transition relation.

Note that the set of actions of M_{Imp} is the same as that of M_{Spec}. In the worst case, the size of S_{Imp} is exponential in the size of $\mathcal{P}$. Therefore, the worst case space and time complexities of constructing M_{Imp} are exponential as well.

F. Computing T_{Imp}.

Computing the transitions between the states in M_{Imp} requires a theorem prover. We add a transition between two abstract states unless we can prove that there is no transition between their corresponding concrete states. We observe that this problem can be reduced to the problem of deciding whether $\neg(\psi_1 \wedge \psi_2)$ is valid, where ψ_1 and ψ_2 are first order formulas over the integers. In general this problem is known to be undecidable. However for our purposes it is sufficient that the theorem prover be sound and always terminate. Several publicly available theorem provers (such as Simplify [18]) have this characteristic.

Given arbitrary formulas ψ_1 and ψ_2, we say that the formulas are *admissible* if the theorem prover returns FALSE or UNKNOWN on $\neg(\psi_1 \wedge \psi_2)$. We denote this by $\mathcal{A}(\psi_1, \psi_2)$. Otherwise the formulas are inadmissible, which is denoted by $\neg\mathcal{A}(\psi_1, \psi_2)$. The definition of T_{Imp} consists of several sub-cases. First, we add a transition $\text{INIT} \xrightarrow{\tau} (I_{CF}, V)$ iff $\mathcal{A}(\Gamma_{I_{CF}}(V), \mathcal{G}_{Spec})$. Next we add $((s_1, V_1), \tau, (s_2, V_2))$ to T_{Imp} iff $(s_1, s_2) \in T_{CF}$ and one of the following conditions hold:

1. $\mathcal{L}(s_1)$ is an assignment statement and $\mathcal{A}(\Gamma_{s_1}(V_1), \mathcal{WP}[\mathcal{L}(s_1)]\{\Gamma_{s_2}(V_2)\})$.

2.1. $\mathcal{L}(s_1)$ is a branch statement with a branch condition c, $\mathcal{L}(s_2)$ is its `then` successor, $\mathcal{A}(\Gamma_{s_1}(V_1), \Gamma_{s_2}(V_2))$ and $\mathcal{A}(\Gamma_{s_1}(V_1), c)$.

2.2. $\mathcal{L}(s_1)$ is a branch statement with a branch condition c, $\mathcal{L}(s_2)$ is its `else` successor, $\mathcal{A}(\Gamma_{s_1}(V_1), \Gamma_{s_2}(V_2))$ and $\mathcal{A}(\Gamma_{s_1}(V_1), \neg c)$.

3. $\mathcal{L}(s_1)$ is a `goto` statement and $\mathcal{A}(\Gamma_{s_1}(V_1), \Gamma_{s_2}(V_2))$.

1) Handling `return` *statements:* We add $((s, V), a, \text{STOP})$ to T_{Imp} iff $\mathcal{L}(s)$ is a `return` statement, a is a return action, and either (i) $\mathcal{L}(s)$ returns expression e, $a \in \mathit{IntRet}$ and $\mathcal{A}(\Gamma_s(V), (e \;==\; \mathit{RetVal}(a)))$, or (ii) $\mathcal{L}(s)$ returns `void` and $a = \mathit{VoidRet}$. If $\mathcal{L}(s)$ returns expression e but condition (i) above is not applicable for any $a \in \mathit{IntRet}$, we add $((s, V), \mathit{VoidRet}, \text{STOP})$ to T_{Imp}. This ensures that from every ``return'' state there is at least one return action to STOP, and if an applicable return action cannot be determined, $\mathit{VoidRet}$ is used as the default.

2) Handling call-sites: Suppose $\mathcal{L}(s_1)$ is a call-site for library routine `lib` and $\langle g_1, P_1\rangle, \ldots, \langle g_n, P_n\rangle$ is the guard and LTS list in the assumption PA for `lib`. Also, let $(s_1, s_2) \in T_{CF}$, $V_1 \in \mathcal{V}_{s_1}$ and $V_2 \in \mathcal{V}_{s_2}$. Then for $1 \le i \le n$, 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 $\mathcal{L}(s_1)$. If $\mathcal{A}(g'_i, \Gamma_{s_1}(V_1))$, then let $P_i = (S_i, s_{\{0,i\}}, Act_i, T_i)$ and proceed to step 2, otherwise move on to the next i.

2. Add a transition $((s_1, V_1), \tau, (s_1, V_1, s_{\{0,i\}}))$ to T_{Imp}.

3. For each transition $(s, a, t) \in T_i$ where $t \neq \text{STOP}$, add a transition $((s_1, V_1, s), a, (s_1, V_1, t))$ to T_{Imp}.

4. If $\mathcal{L}(s_1)$ is a call-site with an assignment, i.e. of the form `x = lib(...)`, then:

- For each transition $(s, \mathit{VoidRet}, \text{STOP}) \in T_i$ such that $\mathcal{A}(\Gamma_{s_1}(V_1), \Gamma_{s_2}(V_2))$, add $((s_1, V_1, s), \tau, (s_2, V_2))$ to T_{Imp}.
- For each transition $(s, a, \text{STOP}) \in T_i$ such that $a \in \mathit{IntRet}$ and $\mathcal{A}(\Gamma_{s_1}(V_1), \mathcal{WP}[\mathtt{x} = \mathit{RetVal}(a)]\{\Gamma_{s_2}(V_2)\})$, add $((s_1, V_1, s), \tau, (s_2, V_2))$ to T_{Imp}.

5. If $\mathcal{L}(s_1)$ is a call-site without an assignment, i.e. of the form `lib(...)`, then for each transition $(s, a, \mathrm{STOP}) \in T_i$ such that $\mathcal{A}(\Gamma_{s_1}(V_1), \Gamma_{s_2}(V_2))$, add $((s_1, V_1, s), \tau, (s_2, V_2))$ to T_{Imp}.

Example 4: Recall the CFA from Example 1 and the predicates corresponding to CFA nodes discussed in Example 2. The models obtained with the set of predicates $\mathcal{P} = \emptyset$ and $\mathcal{P} = \{(y < 10), (y > 5)\}$ are shown in Figure 7(a) and 7(b) respectively. Note that 7(a) is not weakly simulated by `Spec`, while 7(b) is weakly simulated by `Spec`.

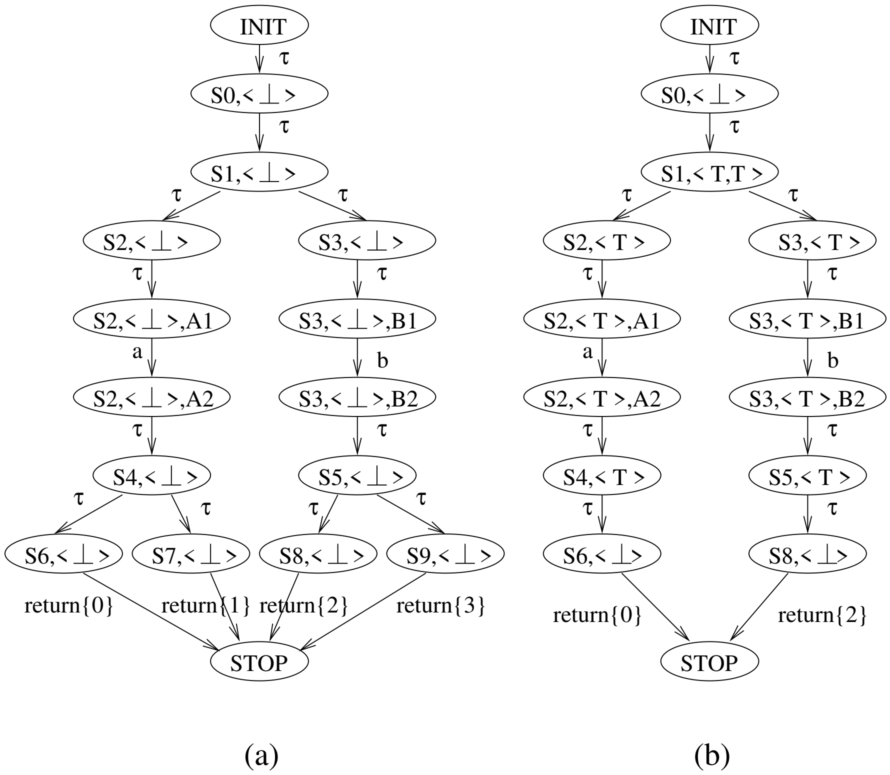

Fig. 7. (a) example M_{Imp} with $\mathcal{P} = \emptyset$; (b) example M_{Imp} with $\mathcal{P} = \{(y < 10), (y > 5)\}$.

V. CHECKING WEAK SIMULATION

In this section, we describe how to check weak simulation between two LTSs. In particular we show how this can be achieved by reducing the problem to one of satisfiability of a weakly-negated HORN formula. We begin with a few preliminary definitions.

A. Definitions

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

A *valuation* is a function from boolean variables to $\{\text{TRUE}, \text{FALSE}\}$. A valuation $\mathcal{V}$ automatically induces a function $\mathcal{V}$ from literals to $\{\text{TRUE}, \text{FALSE}\}$ as follows: (i) $\mathcal{V}(l) = \mathcal{V}(b)$ if l is of the form b and (ii) $\mathcal{V}(l) = \neg\mathcal{V}(b)$ if l is of the form $\neg b$. A valuation $\mathcal{V}$ automatically induces a function $\mathcal{V}$ from clauses to $\{\text{TRUE}, \text{FALSE}\}$ as follows. Let $c = (l_1 \vee \ldots \vee l_m)$ be a clause. Then $\mathcal{V}(c) = \vee_{i=1}^{m} \mathcal{V}(l_i)$. In the same spirit, a valuation $\mathcal{V}$ automatically induces a function $\mathcal{V}$ from CNF formula to $\{\text{TRUE}, \text{FALSE}\}$ as follows. Let $\phi = (c_1 \wedge \ldots \wedge c_n)$ be a CNF formula. Then $\mathcal{V}(\phi) = \wedge_{i=1}^{n} \mathcal{V}(c_i)$. A CNF formula ϕ is said to be satisfiable iff there exists a valuation $\mathcal{V}$ such that $\mathcal{V}(\phi) = \text{TRUE}$.

A CNF formula $(c_1 \wedge \ldots \wedge c_n)$ is said to be a weakly negated HORN (N-HORN) formula iff each c_i contains at most one negative literal for $1 \leq i \leq n$. The problem of checking the satisfiability of an arbitrary N-HORN formula is known as N-HORNSAT. There exists a well-known algorithm [44] for solving the N-HORNSAT problem that requires linear time and space in the size of the input formula. We are now ready to present the N-HORNSAT based weak simulation checking algorithm used by MAGIC.

B. Reducing weak simulation to N-HORNSAT

Let $M_1 = \langle S_1, init_1, \Sigma, T_1 \rangle$ and $M_2 = \langle S_2, init_2, \Sigma, T_2 \rangle$ be two LTSs with the same alphabet. Our goal is to create a weakly negated HORN formula $\phi(M_1, M_2)$ such that $\phi(M_1, M_2)$ is satisfiable iff $M_1 \preccurlyeq M_2$. For each $s_1 \in S_1$ and $s_2 \in S_2$ we introduce a boolean variable that we denote $BV(s_1, s_2)$. Intuitively, $BV(s_1, s_2)$ stands for the proposition that there exists a weak simulation relation R such that $(s_1, s_2) \in R$. We then generate a set of clauses that constrain the various boolean variables according to the definition of a weak simulation relation.

In particular suppose $BV(s_1, s_2)$ is TRUE. Then there exists a weak simulation relation R such that $(s_1, s_2) \in R$. Now suppose $s_1 \xrightarrow{a} t_1$ where $a \neq \tau$. Then from the definition of a weak simulation relation, there must exist some state t_2 such that $s_2 \xrightarrow{a} t_2$ and further $(t_1, t_2) \in R$. In other words, if $Succ(s_2, a) = \{t_2^1, \ldots, t_2^n\}$, then it must be the case that $\{(t_1, t_2^1), \ldots, (t_1, t_2^n)\} \cap R \neq \emptyset$. But this argument can be expressed formally by the following clause: $BV(s_1, s_2) \Rightarrow \vee_{i=1}^n BV(t_1, t_2^i)$.

If $a = \tau$, then we have to allow for the additional possibility that $(t_1, s_2) \in R$. Hence, the corresponding clause is: $BV(s_1, s_2) \Rightarrow \vee_{i=1}^n BV(t_1, t_2^i) \vee BV(t_1, s_2)$. In essence, our target formula $\phi(M_1, M_2)$ is composed of such clauses, one for each appropriate choice of s_1, s_2, a and t_1. The following algorithm describes precisely how these clauses are generated.

For `each` $s_1 \in S_1$
  For `each` $s_2 \in S_2$
    For `each` $a \in \Sigma$
      For `each` $t_1 \in Succ(s_1, a)$
        If $(a \neq \tau)$ **then output** $BV(s_1, s_2) \Rightarrow \vee_{t_2 \in Succ(s_2,a)} BV(t_1, t_2)$
        Else Output $BV(s_1, s_2) \Rightarrow \vee_{t_2 \in Succ(s_2,a)} BV(t_1, t_2) \vee BV(t_1, s_2)$

As a special case, when $Succ(s_2, a) = \emptyset$, the generated clause is simply $\neg BV(s_1, s_2)$. Finally we have to express the constraint that there exists a weak simulation relation R such that $(init_1, init_2) \in R$. But this can be done precisely by the singleton clause $BV(init_1, init_2)$. In summary, our target formula $\phi(M_1, M_2)$ consists of the clauses generated by the algorithm given above, along with the singleton clause $BV(init_1, init_2)$. Note that $\phi(M_1, M_2)$ is a N-HORN formula.

The above method of checking weak simulation via N-HORNSAT is well-known [42]. Further, N-HORNSAT can be solved in linear time and space [43]. This yields extremely efficient algorithms for checking weak simulation between two LTSs. In addition, recall that the CEGAR loop used by MAGIC makes it necessary to construct a counterexample if the weak simulation check fails. As part of MAGIC we have implemented an extended version of the N-HORNSAT algorithm presented by Ausiello and Italiano [44] to achieve precisely this goal. In other words, not only does our algorithm check for satisfiability of N-HORN formulas, but it also constructs a counterexample for the weak simulation relation if the formula is found to be unsatisfiable. To the best of our knowledge, ours is the first attempt to construct counterexamples in the context of weak simulation using SAT procedures.

C. Counterexamples from N-HORNSAT

In general, counterexamples to weak simulation can be viewed as winning strategies of simulation games between the implementation and the specification. In this section we will give a brief overview of the process by which we construct a counterexample. A thorough formal treatment of this procedure, along with detailed algorithms for counterexample validation and abstraction refinement, have been presented elsewhere [26]. We begin with a few simple definitions.

Let $\hat{R}$ be the set $\{R \subseteq S_1 \times S_2 \mid R$ is a weak simulation relation $\}$. In other words $\hat{R}$ is the set of all weak simulation relations between M_1 and M_2. Suppose we order the elements of $\hat{R}$ by the subset ordering, i.e., for any two elements $R_1 \in \hat{R}$ and $R_2 \in \hat{R}$, $R_1 \leq R_2$ iff $R_1 \subseteq R_2$. Then it is well-known that there exists a unique maximal element of $\hat{R}$ called the maximal weak simulation relation. Let us denote this maximal element by $\preccurlyeq_m$. From this it follows that $M_1 \preccurlyeq M_2$ iff $init_1 \preccurlyeq_m init_2$. In other words, one can check if $M_1 \preccurlyeq M_2$ in two steps: (i) compute $\preccurlyeq_m$; (ii) check if $(init_1, init_2) \in \preccurlyeq_m$.

A standard algorithm to compute $\preccurlyeq_m$ works as follows: (i) start with $S_1 \times S_2$ as the initial guess for $\preccurlyeq_m$; (ii) repeatedly eliminate elements from the current guess till a fixed point is reached. In each iteration, elements are eliminated on the basis of elements already eliminated in previous iterations and following the definition of a weak simulation relation.

For example, suppose that state t_1 has an outgoing transition $t_1 \xrightarrow{a} w_1$ (where $a \neq \tau$), but state t_2 has no outgoing transitions labeled with a. Then the pair (t_1, t_2) cannot belong to $\preccurlyeq_m$ and hence must be eliminated. Now suppose, in addition, that the only outgoing transition from states s_1 and s_2 are $s_1 \xrightarrow{a} t_1$ and $s_2 \xrightarrow{a} t_2$ respectively. Then, since the pair (t_1, t_2) has been eliminated already, as per the definition of a weak simulation relation, $(s_1, s_2) \not\in \preccurlyeq_m$. Hence the pair (s_1, s_2) must also be eliminated from the guess.

The correctness of the above algorithm is well-known. Clearly, it can be extended to not only compute $\preccurlyeq_m$, but also to record the order in which elements were eliminated from the guess. We have shown [26] that this order, which we call the *elimination order*, can be used to construct a counterexample in case the weak simulation check fails. An elimination order explains why the initial state of the implementation cannot be simulated by the initial state of the specification. In other words, it explains why the specification fails to simulate the implementation. However, as described earlier, we check weak simulation by solving for the satisfiability of an N-HORNSAT formula, and not by computing $\preccurlyeq_m$ explicitly. We must therefore devise a way to construct an elimination order, when necessary, from our N-HORNSAT solver. In the rest of this section we describe the process by which we achieve this goal.

Recall that in order to check weak simulation between two LTSs M_1 and M_2, we first construct an N-HORNSAT formula $\phi(M_1, M_2)$ such that $\phi(M_1, M_2)$ is satisfiable iff $M_1 \preccurlyeq M_2$. We then solve for the satisfiability of $\phi(M_1, M_2)$. In the rest of this section we shall denote $\phi(M_1, M_2)$ as simply ϕ. The satisfiability check occurs in two phases. In the first phase, a directed hypergraph, $\mathcal{HG}_\phi$ is constructed on the basis of the clauses in ϕ. The nodes of $\mathcal{HG}_\phi$ correspond to the Boolean variables in ϕ. We shall denote the node corresponding to Boolean variable b as simply n_b. Additionally there are two special nodes called n_{TRUE} and n_{FALSE}. The edges of $\mathcal{HG}_\phi$ are constructed as follows:

- For each clause of the form $\neg b$ in ϕ, we add an edge from node n_{FALSE} to node n_b.
- For each clause of the form $(b_1 \vee \ldots \vee b_k)$ in ϕ, we add a hyper-edge from the hyper-node $\{n_{b_1}, \ldots, n_{b_k}\}$ to node n_{TRUE}.
- Finally, for each clause of the form $(\neg b_0 \vee b_1 \vee \ldots \vee b_k)$ in ϕ, we add a hyper-edge from the hyper-node $\{n_{b_1}, \ldots, n_{b_k}\}$ to node n_{b_0}.

In the second phase of our N-HORNSAT satisfiability algorithm, we compute the set of nodes of $\mathcal{HG}_\phi$ that are reachable from n_{FALSE}. Let us denote this set of nodes by $\mathcal{R}_{\text{FALSE}}$. It can be shown [49] that ϕ is satisfiable iff $n_{\text{TRUE}} \notin \mathcal{R}_{\text{FALSE}}$. Also $\mathcal{R}_{\text{FALSE}}$ can be computed using linear time and space in the size of $\mathcal{HG}_\phi$ (and hence ϕ). This set $\mathcal{R}_{\text{FALSE}}$ has an additional significance. Recall that the boolean variables in ϕ are of the form $BV(s_1, s_2)$. It can be shown that the following holds:

$$\forall s_1 \in S_1, \forall s_2 \in S_2, n_{BV(s_1,s_2)} \in \mathcal{R}_{\text{FALSE}} \Leftrightarrow (s_1, s_2) \notin \preccurlyeq_m$$

In other words, the elements in $\mathcal{R}_{\text{FALSE}}$ are exactly those nodes that correspond to boolean variables $BV(s_1, s_2)$ such that (s_1, s_2) would have been eliminated from $\preccurlyeq_m$. Additionally, the following can also be shown to be true: suppose that, while computing $\mathcal{R}_{\text{FALSE}}$, elements gets included in it in the following order: $\langle n_{BV(p_1)}, \ldots, n_{BV(p_k)} \rangle$ where each $p_i \in S_1 \times S_2$; then $\langle p_1, \ldots, p_k \rangle$ is a valid elimination order. Our N-HORNSAT based elimination order computation therefore works as follows: when computing $\mathcal{R}_{\text{FALSE}}$ we record the order in which nodes get added to $\mathcal{R}_{\text{FALSE}}$. We output the corresponding order of state pairs. Once an elimination order has been obtained, the counterexample construction can proceed as usual.

VI. Experimental Evaluation

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. All our experiments were performed on a 1.4 GHz AMD Athlon machine with 1 GB of RAM running RedHat Linux 7.1.

A. 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 8 describes the source of each test briefly. LOC indicates the number of post-processed lines of C. The name of the procedure analyzed 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.

B. Regression Test Results.

Figure 10 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 9 summarizes the performance results for various SAT solvers obtained via the regression suite. The y-axis indicates the time in milliseconds needed to check weak simulation since this is the step where the SAT solver is used.

C. Verifying OpenSSL.

To achieve the third goal we opted to work with OpenSSL [50], an open source implementation of the publicly available SSL [51] specification. This protocol is used by a client (typically a

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*

Fig. 8. Descriptions of regression tests.

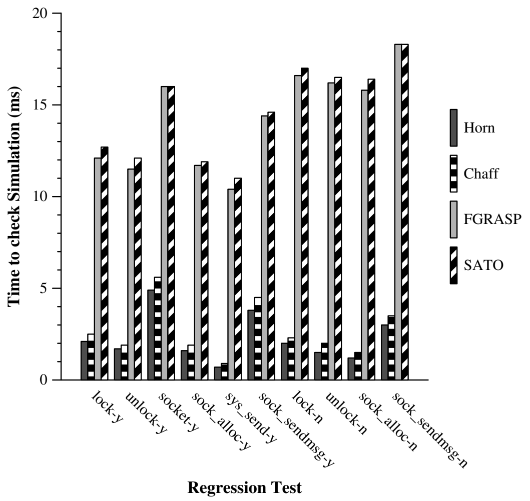

Fig. 9. Time to check weak simulation.

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 [51]. The LTS had 28 states and

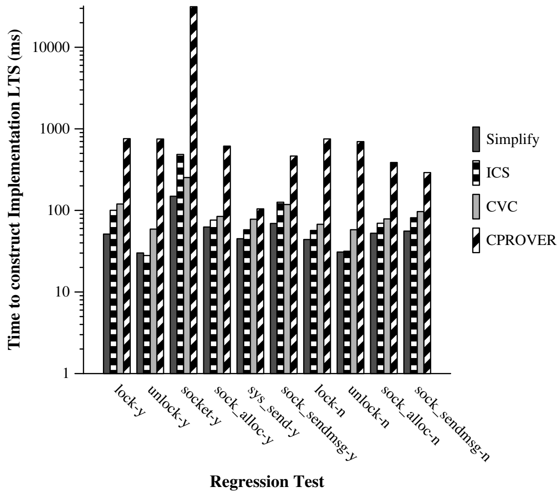

Fig. 10. Time to construct M_{Imp}.

67 transitions. A total of 19 predicates and PAs for 14 library routines were externally supplied. 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 weakly 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 externally supplied.

D. 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.

E. 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, CVC Lite and CPROVER. The first four 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 weak 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.

VII. ONGOING AND FUTURE WORK

We are currently extending and enhancing the MAGIC framework presented in this paper in several important directions. As mentioned previously, the process of extracting a finite model from a C program using predicate abstraction can be exponential in the number of predicates used. Thus, in order to make model construction effective and scalable, one must attempt to keep the set of predicates as small as possible. MAGIC uses a sophisticated predicate minimization [52] scheme based on solving pseudo-Boolean constraints to achieve this goal with encouraging results.

Another important feature not discussed in the current paper is MAGIC's capability of verifying concurrent C programs where the various components communicate among themselves by blocking message passing. In the context of concurrent programs, the state-space grows exponentially with the number of components. As part of MAGIC, we have implemented an automated, compositional, two-level abstraction refinement [53] technique to alleviate the state-space explosion problem and improve scalability. Experimental results indicate that our approach enables us to verify non-trivial concurrent programs using MAGIC.

Finally, we are investigating temporal logic based specification mechanisms which complement the ones presented in this paper. In particular, we are looking into linear [54] and branching-time temporal logics that allow for specifications involving both state propositions and actions. We believe that such state/event based logics will serve as more natural specification mediums for software than corresponding logics based purely on either states or events.

There are many interesting research directions for further work. First, we will look into more expressive abstraction techniques for more precise modeling of the heap and dynamically allocated data structures. Second, we envision an extension of the MAGIC infrastructure to other imperative languages such as Java and C++. Third, it is important that MAGIC be capable of handling concurrent programs that communicate via shared memory as opposed to message passing. A vast majority of multi-threaded C programs fall under this category and we aim at analyzing such programs.

Another important aspect is the analysis of parameterized systems that can consist of an arbitrary number of concurrent components. Currently MAGIC can only handle concurrent programs with an apriori fixed degree of concurrency. Several classes of software systems – most notably libraries implementing shared data structures like trees and priority queues – are however designed to support simultaneous access by an arbitrary number of threads and/or processes. It is important to verify that such systems are implemented in a *thread-safe* manner. In other words, we need to verify that such libraries satisfy certain safety criteria (e.g. absence of data races and stack overflow) irrespective of the number of clients accessing them simultaneously.

ACKNOWLEDGMENT

Several people have contributed, and continue to contribute, critically to the progress of the MAGIC project. In particular, we would like to express our gratitude to Joel Ouaknine, Nishant Sinha, Natasha Sharygina, Ofer Strichman and Karen Yorav for making MAGIC happen.

REFERENCES

[1] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, June 1987. [Online]. Available: http://citeseer.nj.nec.com/harel87statecharts.html

[2] "Unified Modeling Language," http://www.uml.org.

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

[4] E. Clarke, S. Jha, and W. Marrero, "Verifying security protocols with Brutus," *ACM Transactions in Software Engineering Methodology (TOSEM)*, vol. 9, no. 4, 2000.

[5] G. Lowe, S. A. Schneider, B. Roscoe, M. H. Goldsmith, P. Y. A. Ryan, and A. Roscoe, *Modelling and Analysis of Security Protocols*. Addison-Wesley Pub Co, December 2000.

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

[7] "MAGIC," http://www.cs.cmu.edu/∼chaki/magic.

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

[9] C. A. R. Hoare, "Communicating sequential processes," *Communications of the ACM (CACM)*, vol. 21, no. 8, pp. 666–677, August 1978.

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

[11] T. Ball and S. K. Rajamani, "Automatically validating temporal safety properties of interfaces," *Lecture Notes in Computer Science*, vol. 2057, 2001. [Online]. Available: http://citeseer.nj.nec.com/ball01automatically.html

[12] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *Computer Aided Verification*, 2000, pp. 154–169, extended version to appear in J. ACM. [Online]. Available: http://citeseer.nj.nec.com/clarke00counterexampleguided.html

[13] 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*. IEEE Computer Society, 2001, pp. 177–187.

[14] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Lazy abstraction," in *Symposium on Principles of Programming Languages*, 2002, pp. 58–70. [Online]. Available: http://citeseer.nj.nec.com/524901.html

[15] S. Das, D. L. Dill, and S. Park, "Experience with predicate abstraction," in *Computer Aided Verification*, 1999, pp. 160–171. [Online]. Available: http://citeseer.nj.nec.com/das99experience.html

[16] S. Graf and H. Saidi, "Construction of abstract state graphs with PVS," in *Computer Aided Verification*, O. Grumberg, Ed., vol. 1254. Springer Verlag, 1997, pp. 72–83. [Online]. Available: http://citeseer.nj.nec.com/graf97construction.html

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

[18] G. Nelson, "Techniques for program verification," Ph.D. dissertation, Stanford University, 1980.

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

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

[21] "CVC Lite," http://chicory.stanford.edu/CVCL.

[22] D. Kroening, "Application specific higher order logic theorem proving," in *Proc. of the Verification Workshop - VERIFY'02*, S. Autexier and H. Mantel, Eds., July 2002, pp. 5–15.

[23] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient SAT solver," in *Design Automation Conference*, June 2001.

[24] 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.

[25] H. Zhang, "SATO: An efficient propositional prover," in *Conference on Automated Deduction*, 1997.

[26] S. Chaki, E. Clarke, S. Jha, and H. Veith, "Strategy Guided Abstraction Refinement," Carnegie Mellon University, Tech. Rep. CMU-CS-03-188, 2003.

[27] "Bandera," http://www.cis.ksu.edu/santos/bandera.

[28] "Java PathFinder," http://ase.arc.nasa.gov/visser/jpf.

[29] K. Havelund and T. Pressburger, "Model checking JAVA programs using JAVA pathfinder," *International*

Journal on Software Tools for Technology Transfer, vol. 2, no. 4, pp. 366–381, 2000. [Online]. Available: http://citeseer.nj.nec.com/article/havelund99model.html

[30] "ESC-Java," http://www.research.compaq.com/SRC/esc.

[31] "SLAM," http://research.microsoft.com/slam.

[32] "BLAST," http://www-cad.eecs.berkeley.edu/∼rupak/blast.

[33] 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. [Online]. Available: http://citeseer.nj.nec.com/article/engler00checking.html

[34] 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. [Online]. Available: http://citeseer.nj.nec.com/529542.html

[35] 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)*, vol. 8, no. 2, pp. 244–263, April 1986.

[36] E. M. Clarke, O. Grumberg, and D. E. Long, "Model checking and abstraction," *ACM Transactions on Programming Languages and System (TOPLAS)*, vol. 16, no. 5, pp. 1512–1542, September 1994.

[37] T. Ball, A. Podelski, and S. K. Rajamani, "Boolean and Cartesian abstraction for model checking C programs," *Lecture Notes in Computer Science*, vol. 2031, pp. 268–283, 2001. [Online]. Available: http://citeseer.nj.nec.com/ball01boolean.html

[38] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," *Lecture Notes in Computer Science*, vol. 1579, pp. 193–207, 1999. [Online]. Available: http://citeseer.nj.nec.com/article/biere99symbolic.html

[39] D. Jackson, "Automating relational logic," in *ACM SIGSOFT Conference on Foundations of Software Engineering (FSE), San Diego, CA*, November 2000.

[40] D. Jackson and K. Sullivan, "COM revisited: Tool-assisted modelling and analysis of complex software structures," in *ACM SIGSOFT Conference on Foundations of Software Engineering, San Diego, CA*, November 2000.

[41] S. Khurshid and D. Jackson, "Exploring the design of an intentional naming scheme with an automatic constraint analyzer," in *15th IEEE International Conference on Automated Software Engineering (ASE), Grenoble, France*, September 2000.

[42] S. K. Shukla, "Uniform approaches to the verification of finite state systems," Ph.D. dissertation, SUNY, Albany, 1997.

[43] W. F. Dowling and J. H. Gallier, "Linear time algorithms for testing the satisfiability of propositional Horn formula," *Journal of Logic Programming*, vol. 3, pp. 267–284, 1984.

[44] G. Ausiello and G. F. Italiano, "On-line algorithms for polynomially solvable satisfiability problems," *Journal of Logic Programming*, vol. 10, no. 1,2,3 & 4, pp. 69–90, January 1991.

[45] "CIL," http://manju.cs.berkeley.edu/cil.

[46] E. W. Dijkstra, "A simple axiomatic basis for programming language constructs," 1973, lecture notes from the International Summer School on Structured Programming and Programmed Structures. [Online]. Available: http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD372.PDF

[47] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[48] D. Dams and K. S. Namjoshi, "Shape analysis through predicate abstraction and model checking," in *VMCAI*, 2003.

[49] S. K. Shukla, H. B. H. III, and D. J. Rosenkrantz, "HORNSAT, model checking, verification and games," State University of New York, Albany, Tech. Rep. TR-95-8, 1995.

[50] "OpenSSL," http://www.openssl.org.

[51] "SSL 3.0 Specification," http://wp.netscape.com/eng/ssl3.

[52] S. Chaki, E. Clarke, A. Groce, and O. Strichman, "Predicate abstraction with minimum predicates," in *Proceedings of CHARME*, 2003.

[53] S. Chaki, J. Ouaknine, K. Yorav, and E. Clarke, "Automated compositional abstraction refinement for concurrent C programs: A two-level approach," in *Electronic Notes in Theoretical Computer Science*, B. Cook, S. Stoller, and W. Visser, Eds., vol. 89. Elsevier, 2003.

[54] S. Chaki, E. Clarke, J. Ouaknine, N. Sinha, and N. Sharygina, "State/Event-based Software Model Checking," submitted.