Efficient Verification of Sequential and Concurrent C Programs [*]

S. Chaki (chaki@cs.cmu.edu)
E. Clarke (emc@cs.cmu.edu)
A. Groce (agroce@cs.cmu.edu)
J. Ouaknine (ouaknine@cs.cmu.edu)
Carnegie Mellon University, Pittsburgh, USA

O. Strichman (ofers@ie.technion.ac.il)
Technion, Haifa, Israel

K. Yorav (kareny@cs.cmu.edu)
Carnegie Mellon University, Pittsburgh, USA

Abstract. There has been considerable progress in the domain of software verification over the last few years. This advancement has been driven, to a large extent, by the emergence of powerful yet automated abstraction techniques such as predicate abstraction. However, the state-space explosion problem in model checking remains the chief obstacle to the practical verification of real-world distributed systems. Even in the case of purely sequential programs, a crucial requirement to make predicate abstraction effective is to use as few predicates as possible. This is because, in the worst case, the state-space of the abstraction generated (and consequently the time and memory complexity of the abstraction process) is exponential in the number of predicates involved. In addition, for concurrent programs, the number of reachable states could grow exponentially with the number of components.

We attempt to address these issues in the context of verifying concurrent (message-passing) C programs against safety specifications. More specifically, we present a fully automated compositional framework which combines two orthogonal abstraction techniques (predicate abstraction for data and *action-guided abstraction* for events) within a counterexample-guided abstraction refinement scheme. In this way, our algorithm incrementally increases the granularity of the abstractions until the specification is either established or refuted. Additionally, a key feature of our approach is that if a property can be proved to hold or not hold based on a given finite set of predicates $\mathcal{P}$, the predicate refinement procedure we propose in this article finds automatically a minimal subset of $\mathcal{P}$ that is sufficient for the proof. This, along with our explicit use of compositionality, delays the onset of state-space explosion for as long as possible. We describe our approach in detail, and report on some very encouraging experimental results obtained with our tool MAGIC.

Keywords: Software verification, Concurrency, Predicate abstraction, Process algebra, Abstraction refinement

[*] 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

© 2004 *Kluwer Academic Publishers. Printed in the Netherlands.*

1. Introduction

Critical infrastructures in several domains, such as medicine, power, telecommunications, transportation and finance are highly dependent on computers. Disruption or malfunction of services due to software failures (accidental or malicious) can have catastrophic effects, including loss of human life, disruption of commercial activities, and huge financial losses. The increased reliance of critical services on software infrastructure and the dire consequences of software failures have highlighted the importance of software reliability, and motivated systematic approaches for asserting software correctness. While testing is very successful for finding simple, relatively shallow errors, testing cannot guarantee that a program conforms to its specification. Consequently, testing by itself is inadequate for critical applications, and needs to be complemented by automated verification.

Although software verification has been the subject of ambitious projects for several decades, and this research tradition has provided us with important fundamental notions of program semantics and structure, software verification tools have, until recently, not attained the level of practical applicability required by industry. Motivated by urgent industrial need, the success and maturity of formal methods in hardware verification, and by the arrival of new techniques such as predicate abstraction [34], several research groups [1, 3, 5, 7, 32, 33, 35] have started to develop a new generation of software verification tools. A common feature of all these tools is that they operate directly on programs written in a general purpose programming language such as C or Java instead of those written in a more restricted modeling language such as Promela [8]. In addition, all of them are characterized by an extended model checking [22] algorithm which interacts with theorem provers and decision procedures to reason about software abstractions, in particular about abstractions of data types.

Our own tool MAGIC, (**M**odular **A**nalysis of pro**G**rams **I**n **C**), [6, 15–17] also belongs to this family and focuses on modular verification of C code. In general, C programs can be concurrent, i.e., consist of multiple communicating processes and/or threads. Therefore, in order to avoid confusion, we shall adopt the following terminology for the remainder of this article. We shall refer to an arbitrary C program as simply a *program*. Each process/thread of which the program is comprised will be referred to as a *component*. Thus, in particular, for a purely sequential C program, the two terms "program" and "component"

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.

are synonymous. Also, we shall usually denote a program by Π and a component by $\mathcal{C}$ (with appropriate subscripts where applicable).

The general architecture of the MAGIC tool has been presented recently [16] along with an in-depth description of its modular verification approach. In MAGIC, both components and programs are described by labeled transition systems (LTSs), a form of state machines. The goal of MAGIC is to verify whether the implementation Π of a program conforms to its specification *Spec* (which is again expressed using an LTS), i.e., whether all possible behaviors of the implementation are subsumed by the specification.

Several notions of conformance have been proposed in the literature. In the context of this article, we use *trace containment* (denoted by $\preceq$) as our notion of conformance. In the rest of this paper, we shall use "conformance" and "trace containment" synonymously. MAGIC can accept non-deterministic LTSs as *Spec*. It checks trace containment by the standard approach of determining whether the languages of the implementation and the negation of *Spec* have a non-empty intersection. As usual, *Spec* must be determinized before it can be negated. Language intersection is computed by a Boolean satisfiability based symbolic exploration technique.

Since a program can, in general, give rise to an infinite-state system, we will not directly check Π against *Spec*. Rather we will extract an intermediate abstract model $A(\Pi)$ such that Π is guaranteed *by construction* to conform to $A(\Pi)$, and then verify whether $A(\Pi)$, in turn, conforms to *Spec*, i.e.,

$$\Pi \preceq A(\Pi) \preceq \mathit{Spec}$$

The evident problem in this approach is to find a good abstraction $A(\Pi)$. If $A(\Pi)$ is too close to the original system Π, then the computational cost for checking $A(\Pi) \preceq \mathit{Spec}$ may be prohibitively expensive. On the other hand, if $A(\Pi)$ is very coarse then it may well be that relevant features of Π are abstracted away, i.e., $A(\Pi) \not\preceq \mathit{Spec}$, even though $\Pi \preceq \mathit{Spec}$ actually holds. However, an inspection of $A(\Pi) \not\preceq \mathit{Spec}$ provides a counterexample *CE*. In general though, this counterexample may not be grounded on any real behavior of Π and consequently could be *spurious*. This motivates the use of a framework known as **C**ounter**E**xample **G**uided **A**bstraction **R**efinement (CEGAR) [23, 42], which works as follows.

- **Step 1 (Abstract).** Create an abstraction $A(\Pi)$ of the program such that Π conforms to $A(\Pi)$ by construction.

- **Step 2 (Verify).** Verify that $A(\Pi) \preceq \mathit{Spec}$, i.e., $A(\Pi)$ conforms to the specification *Spec*. If this is the case, the verification is successful.

Otherwise, i.e., if $A(\Pi)$ does not conform to *Spec*, obtain a possibly spurious counterexample *CE*. Determine whether *CE* is spurious. If *CE* is not spurious, report the counterexample and stop; otherwise go to the next step.

- **Step 3 (Refine).** Use the spurious counterexample *CE* to refine the abstraction $A(\Pi)$ so as to eliminate *CE* and go to step 1.

Despite the advent of automation via paradigms such as CEGAR, the biggest challenge in making model checking effective remains the problem of state-space explosion. In the context of MAGIC, this problem manifests itself in two forms. First, even in the purely sequential case, state explosion could occur during predicate abstraction. This is because the process of predicate abstraction, in the worst case, requires exponential time and memory in the number of predicates. Second, the state-space size of a concurrent system increases exponentially with the number of components. Hence there is an obvious possibility of state-space explosion for concurrent programs. In this article we present two orthogonal, yet smoothly integrated, techniques developed within MAGIC to tackle the state explosion problem that can occur due to these factors while verifying both sequential and concurrent C programs.

1.1. Predicate Minimization

A fundamental underlying technique used in MAGIC (as well as SLAM [12] and BLAST [37]) is predicate abstraction [34]. Given a (possibly infinite-state) component $\mathcal{C}$ and a set of predicates $\mathcal{P}$, verification with predicate abstraction consists of constructing and analyzing an automaton $\mathcal{A}$, a conservative abstraction of $\mathcal{C}$ relative to $\mathcal{P}$. However, as mentioned before, the process of constructing $\mathcal{A}$ is in the worst case exponential, both in time and space, in $|\mathcal{P}|$. Therefore a crucial point in deriving efficient algorithms based on predicate abstraction is the choice of a *small* set of predicates. In other words, one of the main challenges in making predicate abstraction effective is identifying a small set of predicates that are sufficient for determining whether a property holds or not. The first technique we present is aimed at finding automatically such a minimal set from a given set of candidate predicates [15].

1.2. Compositional Two-level Abstraction Refinement

The second technique we consider is aimed at solving the state-space explosion problem resulting from concurrent systems. We propose a fully automated *compositional two-level CEGAR* scheme to verify that a parallel composition $\Pi = \mathcal{C}_1 || \ldots || \mathcal{C}_n$ of n sequential C components conforms to its specification *Spec* [17]. The basic idea is to extract as small a model as possible from Π by employing two orthogonal abstraction schemes. To this end we use predicate abstraction to handle data and an action-guided abstraction to handle events. Each type of abstraction is also associated with a corresponding refinement scheme. The action-guided abstraction-refinement loop is embedded naturally within the predicate abstraction-refinement cycle, yielding a two-level framework. In addition, abstraction, counterexample validation and refinement can each be carried out component-wise, making our scheme compositional and scalable. More precisely, the steps involved in the two-level CEGAR algorithm presented in this article can be summarized as follows:

- **Step 1: Two-Level Model Construction.** In this stage we use a combination of two abstraction techniques for extracting an LTS model from the concurrent C program $\Pi = \mathcal{C}_1 || \dots || \mathcal{C}_n$.
  - *Step 1.1: Predicate Abstraction.* Initially, we use predicate abstraction to conservatively transform each (infinite-state) C component $\mathcal{C}_i$ into a finite LTS $\mathcal{MP}_i$. In the rest of this article, first level LTS models (obtained after predicate abstraction) will be denoted by $\mathcal{MP}$ (*model-predicate*), with appropriate subscripts where applicable.
  - *Step 1.2: Action-Guided Abstraction.* Since the parallel composition of these $\mathcal{MP}_i$'s may well still have an unmanageably large state-space, we further reduce each $\mathcal{MP}_i$ by conservatively aggregating states together, yielding a smaller LTS $\mathcal{MA}_i$; only then is the model constructed by the much coarser parallel composition $\mathcal{MA} = \mathcal{MA}_1 || \dots || \mathcal{MA}_n$. In the rest of this article, second level LTS models (obtained after action-guided abstraction) will be denoted by $\mathcal{MA}$ (*model-action*), with appropriate subscripts where applicable.
- **Step 2 : Verification.** By construction, the extracted model $\mathcal{MA}$ exhibits all of the original system's behaviors, and usually many more. We now check whether $\mathcal{MA} \preceq \mathit{Spec}$. If successful, we conclude that our original system Π also conforms to its specification *Spec*.
- **Step 3: Refinement.** Otherwise, we must examine the counterexample obtained to determine whether it is valid or not. This validity check is again performed in two stages: first at the level of the $\mathcal{MP}_i$'s and, if that succeeds, at the level of $\mathcal{C}_i$'s. It is also important to note that this validation can be carried out component-wise, without it ever being necessary to construct in full the large state-spaces of the lower-level parallel systems (either the $\mathcal{MP}_i$'s or the $\mathcal{C}_i$'s). A valid counterexample shows *Spec* to be violated and thus terminates

the procedure. Otherwise, a (component-specific) refinement of the appropriate abstracted system is carried out, eliminating the spurious counterexample, and we proceed with a new iteration of the verification cycle. Depending on the counterexample, the refinement process could have two possible outcomes:

- *Case 3.1: Predicate Refinement.* It produces a new set of predicates, leading to a refined $\mathcal{MP}_i$. In this case the new iteration of the two-level CEGAR loop starts with step 1.1 above. It is important to note that the predicate minimization algorithm mentioned above is invoked each time a new set of predicates is generated, leading to a smooth integration of the two techniques we present in this article.

- *Case 3.2: Action-Guided Refinement.* The refinement yields a finer state aggregation, leading to a refined $\mathcal{MA}_i$. In this case the new iteration of the two-level CEGAR loop starts with step 1.2 above.

The verification procedure is fully automated, and requires no user input beyond supplying the C programs, assumptions about the environment, and the specification to be verified. We have implemented the algorithm within MAGIC and have carried out a number of case studies, which we report here. Restrictions on the nature of C programs that MAGIC can handle are discussed in Section 3. To our knowledge, our algorithm is the first to invoke CEGAR over more than a single abstraction refinement scheme (and in particular over *action-based* abstractions), and also the first to combine CEGAR with fully automated compositional reasoning for concurrent systems. In summary, the crucial features of our approach consist of the following:

- We leverage two very different kinds of abstraction to reduce a parallel composition of sequential C programs to a very coarse parallel composition of finite-state processes. The first abstraction partitions the (potentially infinite) state-space according to possible values of predicates over state variables, whereas the second abstraction lumps these resulting states together according to events that they can communicate.

- We invoke a predicate minimization algorithm to compute a minimal set of predicates sufficient to validate or invalidate the specification.

- A counterexample-guided abstraction refinement scheme incrementally refines these abstractions until the right granularity is achieved to decide whether the specification holds or not. We note that while termination cannot be guaranteed, all of our experimental benchmarks could be handled without requiring human intervention.

- Our use of compositional reasoning, grounded in standard process algebraic techniques, enables us to perform most of our analysis component by component, without ever having to construct global state-spaces except at the highest (most abstract) level.

The experiments we have carried out range over a variety of sequential and concurrent examples, and indicate that both the techniques we present, either combined or separately, increase the capacity of MAGIC to verify large C programs. For example, in some cases, predicate minimization can improve the time consumption of MAGIC by over two orders of magnitude and the memory consumption by over an order of magnitude. With the smaller examples we find that our two-level approach constructs models that are 2 to 11 times smaller than those generated by predicate abstraction alone. These ratios increase dramatically as we consider larger and larger examples. In some of our instances MAGIC constructs models that are more than two orders of magnitude smaller than those created by mere predicate abstraction. Full details are presented in Section 9.

The rest of this article is organized as follows. In Section 2 we discuss related work. In section 3 we present some preliminary definitions. In Section 4 we formally define the two-level CEGAR algorithm. In Section 5 we describe the process of constructing the LTS $\mathcal{MP}_i$ from the component $\mathcal{C}_i$ using predicate abstraction. In Section 6 we define the process of checking if a counterexample is valid at the level of the $\mathcal{C}_i$'s. We also present our approach for refining the appropriate $\mathcal{MP}_i$ if the counterexample is found to be spurious. Recall that this is achieved by constructing a minimal set of predicates sufficient to eliminate spurious counterexamples. In Section 7 we present the action-guided abstraction used to obtain $\mathcal{MA}_i$ from $\mathcal{MP}_i$. In Section 8 we define the process of checking if a counterexample is valid at the level of the $\mathcal{MP}_i$'s. We also show how to refine the appropriate $\mathcal{MA}_i$ by constructing a finer state aggregation if the counterexample is found to be spurious. Finally, we present experimental evaluation of our ideas in Section 9 and conclude in Section 10.

2. Related Work

Predicate abstraction [34] was introduced as a means to conservatively transform infinite-state systems into finite-state ones, so as to enable the use of finitary techniques such as model checking [18, 21]. It has since been widely used [11, 12, 26, 27, 29, 30, 37, 46].

The formalization of the more general notion of abstraction was first given by Cousot et al. [28]. We distinguish between *exact* abstractions, which preserve all properties of interest of the system, and *conservative* abstractions—used in this paper—which are only guaranteed to preserve 'undesirable' properties of the system [24, 41]. The advantage of the latter is that they usually lead to much greater reductions in the state-space than their exact counterparts. However, conservative abstractions in general require an iterated abstraction refinement mechanism (such as CEGAR [23]) in order to establish specification satisfaction.

The abstractions we use on finite-state processes essentially lump together states that have the same set of enabled actions, and gradually refine these partitions according to reachable successor states. Our refinement procedure can be seen as an atomic step of the Paige-Tarjan algorithm [49], and therefore yields successive abstractions which converge in a finite number of steps to the bisimulation quotient of the original process.

Counterexample-guided abstraction refinement [23, 42], or CEGAR, is an iterative procedure whereby spurious counterexamples for a specification are repeatedly eliminated through incremental refinements of a conservative abstraction of the system. Both the abstraction and refinement techniques for such systems, as applied elsewhere [23, 42], are essentially different than the predicate abstraction approach we follow. For example, Kurshan [42] abstracts by assigning non-deterministic values to selected sets of variables, while refinement corresponds to gradually returning to the original definition of these variables. CEGAR has been used in many cases, some non-automated [47], others at least partly automated [12, 13, 37, 43, 50]. The problem of finding small sets of predicates has also been investigated in the context of hardware designs [19, 20].

Compositionality, which features crucially in our work, enables the verification of a large system via the verification of its smaller components. In other words, it allows us to decompose a complex verification problem into a set of simpler, more tractable, sub-problems. Compositionality has been most extensively studied in process algebra [40, 45, 51], particularly in conjunction with abstraction. Bensalem et al. [14] have presented a compositional framework for (non-automated) CEGAR over data-based abstractions. Their approach differs from ours in that communication takes place through shared variables (rather than blocking message-passing), and abstractions are refined by eliminating spurious transitions, rather than by splitting abstract states.

A technique closely related to compositionality is that of assume-guarantee reasoning [38, 44]. It was originally developed to circumvent the difficulties associated with generating exact abstractions, and has recently been implemented as part of a fully automated and incremental verification framework [25].

Among the works most closely resembling ours we note the following. The Bandera project [27] offers tool support for the automated verification of Java programs based on abstract interpretation; there is no automated CEGAR and no explicit compositional support for concurrency.Păsăreanu et al. [50] have imported Bandera-derived abstractions into an extension of Java PathFinder [35] which incorporates CEGAR. However, once again no use is made of compositionality, and only a single level of abstraction is considered. Stoller [52] has implemented another tool in Java PathFinder which explicitly supports concurrency; it uses datatype abstraction on the first level, and partial order reduction with aggregation of invisible transitions on the second level. Since all abstractions are exact it does not require the use of CEGAR. The SLAM project [7, 11, 12] has been very successful in analyzing interfaces written in C. It is built around a single-level predicate abstraction and automated CEGAR treatment, and offers no explicit compositional support for concurrency. Lastly, the BLAST project [1, 36, 37] proposes a single-level lazy (on-the-fly) predicate abstraction scheme together with CEGAR and thread-modular assume-guarantee reasoning. The BLAST framework is based on shared variables rather than message-passing as the communication mechanism.

3. Background

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

DEFINITION 1 (**Labeled Transition Systems**). *A labeled transition system (LTS) M is a quadruple $(S, init, Act, T)$, where (i) S is a finite non-empty set of states, (ii) $init \in S$ is the initial state, (iii) Act is a finite set of actions (alphabet), and (iv) $T \subseteq S \times Act \times S$ is the transition relation.*

We assume that there is a distinguished state $\mathrm{STOP} \in S$ which has no outgoing transitions, i.e., $\forall s' \in S, \forall a \in Act, (\mathrm{STOP}, a, s') \notin T$. If $(s, a, s') \in T$, then (s, s') will be referred to as an a-transition and will be denoted by $s \xrightarrow{a} s'$. For a state s and action a, we define $\mathrm{Succ}(s, a) = \{s' \mid s \xrightarrow{a} s'\}$. Action a is said to be *enabled* at state s iff $\mathrm{Succ}(s, a) \neq \emptyset$. For $s \in S$ we write $export(s) = \{a \in Act \mid \mathrm{Succ}(s, a) \neq \emptyset\}$ to denote the set of actions enabled in state s.

3.1. Actions

In accordance with existing practice, we use actions to denote externally visible behaviors of systems being analyzed, e.g., acquiring a lock. Actions are atomic, and are distinguished simply by their names. Since we are analyzing C, 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 $RetVal(r)$. 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 *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.

DEFINITION 2 (**Traces**). *A trace π is a finite (possibly empty) sequence of actions. We define the language $L(M)$ of the LTS M to be the set of all traces $a_1 \ldots a_n \in Act^*$ such that, for some sequence $s_0 \ldots s_n$ of states of M (with $s_0 = init$) we have $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} s_n$. We refer to the underlying sequence of states $s_0 \ldots s_n$ as the path in M corresponding to the trace $a_1 \ldots a_n$.*

DEFINITION 3 (**Reachability**). *For a trace $\pi = a_1 \ldots a_n \in Act^*$ and $s, t \in S$ two states of M, we write $s \stackrel{\pi}{\Longrightarrow} t$ to indicate that t is reachable from s through π, i.e., that there exist states $s_0 \ldots s_n$ with $s = s_0$ and $t = s_n$, such that $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} s_n$. Given a state $s \in S$ and a trace $\pi \in Act^*$, we let $Reach(M, s, \pi) = \{t \in S \mid s \stackrel{\pi}{\Longrightarrow} t\}$ stand for the set of states reachable from s through π. We overload this notation by setting, for a set of states $Q \subseteq S$, $Reach(M, Q, \pi) = \bigcup_{q \in Q} Reach(M, q, \pi)$; this represents the set of states reachable through π from some state in Q.*

DEFINITION 4 (**Projection**). *Let $\pi \in Act^*$ be a trace over Act, and let Act' be another (not necessarily disjoint) alphabet. The projection $\pi \restriction Act'$ of π on Act' is the sub-trace of π obtained by simply removing all actions in π that are not in Act'.*

DEFINITION 5 (**Parallel Composition**). *Let $M_1 = \langle S_1, init_1, Act_1, T_1 \rangle$ and $M_2 = \langle S_2, init_2, Act_2, T_2 \rangle$ be two LTSs. Their parallel composition $M_1 || M_2 = \langle S_1 \times S_2, (init_1, init_2), Act_1 \cup Act_2, T_1 || T_2 \rangle$ is defined so that $((s_1, s_2), a, (t_1, t_2)) \in T_1 || T_2$ iff one of the following holds:*

1. *$a \in Act_1 \setminus Act_2$ and $s_1 \xrightarrow{a} t_1$ and $s_2 = t_2$.*

2. $a \in Act_2 \setminus Act_1$ *and* $s_2 \xrightarrow{a} t_2$ *and* $s_1 = t_1$.

3. $a \in Act_1 \cap Act_2$ *and* $s_1 \xrightarrow{a} t_1$ *and* $s_2 \xrightarrow{a} t_2$.

In other words, components must synchronize on shared actions and proceed independently on local actions. This notion of parallel composition originates from CSP. The following results are well-known [40, 51] and we state them here without proof:

THEOREM 1.

1. *Parallel composition is associative and commutative as far as the accepted language is concerned. Thus, in particular, no bracketing is required when combining more than two LTSs.*

2. *Let* $M_1, \ldots, M_n$ *and* $M'_1, \ldots, M'_n$ *be LTSs with every pair of LTSs* M_i, M'_i *sharing the same alphabet* $Act_i = Act'_i$. *If, for each* $1 \leqslant i \leqslant n$, *we have* $L(M_i) \subseteq L(M'_i)$, *then* $L(M_1 || \ldots || M_n) \subseteq L(M'_1 || \ldots || M'_n)$. *In other words, parallel composition preserves language containment.*

3. *Let* $M_1, \ldots, M_n$ *be LTSs with respective alphabets* $Act_1, \ldots, Act_n$, *and let* $\pi \in (Act_1 \cup \ldots \cup Act_n)^*$ *be any trace. Then* $\pi \in L(M_1 || \ldots || M_n)$ *iff, for each* $1 \leqslant i \leqslant n$, *we have* $\pi \restriction Act_i \in L(M_i)$. *In other words, whether a trace belongs to a parallel composition of LTSs can be checked by projecting and examining the trace on each individual component separately.*

3.2. Restrictions on C Programs

We developed the C parser used by MAGIC using the standard ANSI C grammar. Hence MAGIC can only analyze pure ANSI C programs. To analyze a C program Π with GCC and MSVC extensions, we first use the CIL [2] tool to translate Π into an equivalent pure ANSI C program Π_{ANSI} and subsequently use MAGIC on Π_{ANSI}. Also, MAGIC can only analyze non-recursive C programs and uses inlining to convert a set of procedures with a DAG-like call graph into a single procedure.

Finally, before applying predicate abstraction, MAGIC conservatively eliminates two commonly used constructs in C programs: (i) indirect function calls via function pointers and (ii) indirect assignments to variables using pointer dereferences, i.e., pointer dereferences on the left hand sides (LHS's) of assignments. Therefore, without loss of generality, we will assume that the C programs on which predicate abstraction is used do not contain these two constructs. The details of the process by which MAGIC obtains and uses alias information are presented next.

3.3. Using Alias Analysis

In essence, we rewrite Π to obtain a new program Π' with two crucial properties:

AL1. Π' does not contain any indirect function calls or pointer dereferences.

AL2. Π' exhibits every behavior of Π (and possibly some more), i.e. it is a safe abstraction of Π.

The obtained Π' can then be subjected to the model construction algorithm (to be presented later.) The resulting model will be a safe abstraction of Π', and hence a safe abstraction of Π. Π' is obtained from Π in two stages. First, Π is rewritten to eliminate all indirect function calls to obtain Π''. Next Π'' is rewritten to eliminate pointer dereferences on the LHS's of assignments to obtain Π'. We now describe these two stages in more detail.

3.3.1. *Handling Function Pointers*

Let $\mathtt{x} = *\mathtt{fp}(\ldots)$ be an arbitrary call-site of Π. Let $\{\mathtt{lib}_1, \ldots, \mathtt{lib}_k\}$ be the set of possible target procedures of `fp` obtained by alias analysis. We rewrite the call-site to the following `for` statement.

```
if (fp == lib_1) x = lib_1(...);
          .........
else if (fp == lib_{k-1}) x = lib_{k-1}(...);
else x = lib_k(...);
```

3.3.2. *Handling Pointer Dereferences*

Let $*p = e$ be an arbitrary assignment in Π'' and let $\{\mathtt{v}_1, \ldots, \mathtt{v}_k\}$ be the set of possible target locations of `p` obtained by alias analysis. We rewrite the assignment to the following `for` statement.

```
if (p == &v_1) v_1 = e;
      .........
else if (p == &v_{k-1}) v_{k-1} = e;
else v_k = e;
```

It is easy to see that if the alias analysis used is conservative (commonly called a "may" alias analysis), then the resulting Π' obeys both conditions **AL1** and **AL2** mentioned above. We use PAM [4] to obtain such "may" alias information from C programs[1]. PAM works in two stages. In the first stage it performs a conservative alias analysis based on Anderson's [10] technique to create a *points-to database*. In the second stage, it provides a C procedure based API for querying the points-to database. This API can be used by external tools to obtain alias information. We have integrated MAGIC with PAM, allowing MAGIC to obtain the required alias information using PAM in a completely automated manner.

3.4. Procedure Abstraction

Because of the specific restrictions discussed in the previous section and MAGIC's use of inlining, we can assume that a software component consists of a single (non-recursive) procedure. In other words, in our approach, procedures and components are synonymous. As mentioned before, MAGIC allows specifications to be supplied for components (i.e., procedures), as well as for programs. As the goal of MAGIC is to verify whether a C program conforms to its specification, the need for program specifications is obvious.

Procedure specifications serve another crucial purpose. They are used as assumptions while constructing component models via predicate abstraction. More precisely, suppose we are trying to construct $\mathcal{MP}_i$ from component $\mathcal{C}_i$. In general $\mathcal{C}_i$ may invoke several library routines and it is quite common for the actual implementation of these routines to be unavailable during verification. In such situations, a specification for these routines is used instead to construct $\mathcal{MP}_i$ conservatively.

In practice, it often happens that a procedure performs quite different tasks for various calling contexts, i.e., values of its parameters and global variables. In our approach, this phenomenon is accounted for by allowing multiple specification LTSs for a single procedure. The selection among these LTSs is achieved by *guards*, i.e., formulas, which describe the conditions on the procedure's parameters and globals under which a certain LTS is applicable. This gives rise to the notion of *procedure abstraction* (PA). Formally, a PA for a procedure $\mathcal{C}$ is a tuple $\langle d, l \rangle$ where:

- d is the declaration for $\mathcal{C}$, as it appears in a C header file.
- l is a finite list $\langle g_1, M_1 \rangle, \ldots, \langle g_k, M_k \rangle$ where each g_i is a guard formula ranging over the parameters and globals of $\mathcal{C}$, and each M_i is an LTS.

[1] PAM produces may alias information modulo some reasonable assumptions. For example it assumes that pointer values cannot arise as a result of bitwise operations.

The procedure abstraction expresses that $\mathcal{C}$ conforms to one LTS chosen among the M_i's. More precisely, $\mathcal{C}$ conforms to M_i if the corresponding guard g_i evaluates to true over the *actual arguments* and globals passed to $\mathcal{C}$. We require that the guard formulas g_i be mutually exclusive and complete (i.e., cover all possibilities) so that the choice of M_i is always unambiguously defined. Since PAs are used as assumptions during model construction, they are often referred to as assumption PAs. Note that both guards and LTSs of assumption PAs must be completely specified by the user.

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 $(S, \mathit{init}, \mathit{Act}, T)$ must obey the following condition: $\forall s \in S, s \xrightarrow{a} \text{STOP}$ iff a is a return action.

3.5. Program Abstraction

Just as a PA encapsulates a component's specification, a program's specification is expressed through a program abstraction. Program abstractions are defined in a similar manner as PAs. Formally, let Π be a program consisting of n components $\mathcal{C}_1, \ldots, \mathcal{C}_n$. Then a program abstraction for Π is a tuple $\langle d, l \rangle$ where:

- $d = \langle d_1, \ldots, d_n \rangle$ is the list of declarations for $\mathcal{C}_1, \ldots, \mathcal{C}_n$.
- l is a finite list $\langle g_1, M_1 \rangle, \ldots, \langle g_k, M_k \rangle$ where each $g_i = \langle g_i^1, \ldots, g_i^n \rangle$ such that g_i^j is a guard formula ranging over the parameters and globals of $\mathcal{C}_j$, and each M_i is an LTS.

Without loss of generality we will assume throughout this paper that our target program abstraction contains only one guard $\langle \mathit{Guard}_1, \ldots, \mathit{Guard}_n \rangle$ and one LTS *Spec*. To achieve the result in full generality, the described algorithm can be iterated over each element of the target abstraction.

3.6. Concurrency and Communication

We consider a concurrent version of the C programming language in which a fixed number of sequential components $\mathcal{C}_1, \ldots, \mathcal{C}_n$ are run concurrently on independent platforms. Each component $\mathcal{C}_i$ has an associated alphabet of actions Act_i, and can communicate a particular event a in its alphabet only if all other programs having a in their alphabets are willing to synchronize on this event. Actions are realized using calls to library routines. Programs have local variables but no shared variables. In other words, we are assuming blocking message-passing (i.e., ‘send’ and ‘receive’ statements) as the sole communication mechanism. When there is no communication, the components execute asynchronously.

4. Two-Level Counterexample-Guided Abstraction Refinement

Consider a concurrent C program $\Pi = \mathcal{C}_1 || \ldots || \mathcal{C}_n$ and a specification *Spec*. Our goal is to verify that the concurrent C program Π conforms to the LTS *Spec*. Since we use trace containment as our notion of conformance, the concurrent program meets its specification iff $L(\Pi) \subseteq L(\mathit{Spec})$.

Theorem 1 forms the basis of our compositional approach to verification. We first invoke predicate abstraction to reduce each (infinite-state) program $\mathcal{C}_i$ into a finite LTS (or process) $\mathcal{MP}_i$ having the same alphabet as $\mathcal{C}_i$. The initial abstraction is created with a relatively small set of predicates, and further predicates are then added as required to refine the $\mathcal{MP}_i$'s and eliminate spurious counterexamples. This procedure may add a large number of predicates, yielding an abstract model with a potentially huge state-space. We therefore seek to further reduce each $\mathcal{MP}_i$ into a smaller LTS $\mathcal{MA}_i$, again having the same alphabet as $\mathcal{C}_i$. Both abstractions are such that they maintain the language containment $L(\mathcal{C}_i) \subseteq L(\mathcal{MP}_i) \subseteq L(\mathcal{MA}_i)$. Theorem 1 then immediately yields the rule:

$$L(\mathcal{MA}_1 || \ldots || \mathcal{MA}_n) \subseteq L(\mathit{Spec}) \Rightarrow L(\mathcal{C}_1 || \ldots || \mathcal{C}_n) \subseteq L(\mathit{Spec})$$

The converse need not hold: it is possible for a trace $\pi \notin L(\mathit{Spec})$ to belong to $L(\mathcal{MA}_1 || \ldots || \mathcal{MA}_n)$ but not to $L(\mathcal{C}_1 || \ldots || \mathcal{C}_n)$. Such a spurious counterexample is then eliminated, either by suitably refining the $\mathcal{MA}_i$'s (if $\pi \notin L(\mathcal{MP}_1 || \ldots || \mathcal{MP}_n)$), or by refining the $\mathcal{MP}_i$'s (and subsequently adjusting the $\mathcal{MA}_i$'s to reflect this change). The chief property of our refinement procedure (whether at the $\mathcal{MA}_i$ or the $\mathcal{MP}_i$ level) is that it purges the spurious counterexample by restricting the accepted language yet maintains the invariant $L(\mathcal{C}_i) \subseteq L(P_i') \subseteq L(\mathcal{MA}_i') \subset L(\mathcal{MA}_i)$, where primed terms denote refined processes. Note that, according to Theorem 1, we can check whether $\pi \in L(\mathcal{MP}_1 || \ldots || \mathcal{MP}_n)$ and whether $\pi \in L(\Pi)$ one component at a time, without it ever being necessary to construct the full state-spaces of the parallel compositions. This iterated process forms the basis of our two-level CEGAR algorithm.

We describe this algorithm in Figure 1. The predicate abstraction and refinement procedures are detailed in Section 5 and Section 6. We present our action-guided abstraction and refinement steps (marked † and ‡ respectively) in Section 7 and Section 8.

```
Input: C components $\mathcal{C}_1,\ldots,\mathcal{C}_n$ and specification $Spec$
Output: '$\mathcal{C}_1||\ldots||\mathcal{C}_n$ satisfies $Spec$' or
        counterexample $\pi \in L(\mathcal{C}_1||\ldots||\mathcal{C}_n) \setminus L(Spec)$

  predicate abstraction: create LTSs $\mathcal{MP}_1,\ldots,\mathcal{MP}_n$ with
                                      $L(\mathcal{C}_i) \subseteq L(\mathcal{MP}_i)$
† action-guided abstraction: create LTSs $\mathcal{MA}_1,\ldots,\mathcal{MA}_n$ with
                                      $L(\mathcal{MP}_i) \subseteq L(\mathcal{MA}_i)$
  for $1 \le i \le n$, let $Act_i$ be the alphabet of $\mathcal{MP}_i$ and $\mathcal{MA}_i$
  repeat
     if $L(\mathcal{MA}_1||\ldots||\mathcal{MA}_n) \subseteq L(Spec)$
        return '$\mathcal{C}_1||\ldots||\mathcal{C}_n$ satisfies $Spec$'
     else
        extract counterexample $\pi \in L(\mathcal{MA}_1||\ldots||\mathcal{MA}_n) \setminus L(Spec)$
        if for $1 \le i \le n$, $\pi \restriction Act_i \in L(\mathcal{MP}_i)$
           if for $1 \le i \le n$, $\pi \restriction Act_i \in L(\mathcal{C}_i)$ return $\pi$
           else
              let $1 \le j \le n$ be an index such that $\pi \restriction Act_j \notin L(\mathcal{C}_j)$
              do predicate abstraction refinement of $\mathcal{MP}_j$
                 to eliminate $\pi$
†             adjust or create new abstraction $\mathcal{MA}_j$
        else
‡          let $1 \le j \le n$ be an index such that $\pi \restriction Act_j \notin L(\mathcal{MP}_j)$
           do action-guided refinement of $\mathcal{MA}_j$ to eliminate $\pi$
  endrepeat.
```

Figure 1. Two-level CEGAR algorithm.

5. Predicate Abstraction

Let $Spec = (S_S, init_S, Act_S, T_S)$ and the assumption PAs be $\{PA_1,\ldots,PA_k\}$. In this section we show how to extract $\mathcal{MP}_i$ from $\mathcal{C}_i$ using the assumption PAs, the guard $Guard_i$ and the predicates (in the first abstraction, the set of predicates used in empty). This is a brief description of the model construction technique described by Chaki et al. [16]. Since in this section we shall be dealing with a single component, we shall refer to $\mathcal{C}_i$, $\mathcal{MP}_i$ and $Guard_i$ as simply $\mathcal{C}$, $\mathcal{MP}$ and $Guard$ respectively. The extraction of $\mathcal{MP}$ relies on several principles:

- Every state of $\mathcal{MP}$ models a state during the execution of $\mathcal{C}$; 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 control flow graph (CFG) of $\mathcal{C}$.
- The data components are *abstract representations* of the memory state of $\mathcal{C}$. These abstract representations are obtained using predicate abstraction.
- The transitions between states in $\mathcal{MP}$ are derived from the transitions in the CFG, 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.

Without loss of generality, we can assume that there are only five kinds of statements in $\mathcal{C}$: assignments, call-sites, `if-then-else` branches, `goto` and `return`. Note that call-sites correspond to library routines called by $\mathcal{C}$ whose code is unavailable and therefore cannot be inlined. We can also assume, without loss of generality, that all expressions in $\mathcal{C}$ are side-effect free. Recall, from Section 3.2, that we have assumed the absence of indirect function calls and pointer dereferences on the LHS's of assignments. We denote by *Stmt* the set of statements of $\mathcal{C}$ and by *Expr* the set of all C expressions over the variables of $\mathcal{C}$.

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

Figure 2. The example C component $\mathcal{C}$.

As a running example of $\mathcal{C}$, we use the C procedure shown in Figure 2. It invokes two library routines `do_a` and `do_b`. Let the guard and LTS list in the assumption PA for `do_a` be $\langle \text{TRUE}, \texttt{Do\_A} \rangle$. This means that under all invocation conditions, `do_a` conforms to the LTS `Do_A`. Similarly the guard and LTS list in the assumption PA for `do_b` is

$\langle \text{TRUE}, \texttt{Do\_B} \rangle$. The LTSs `Do_A` and `Do_B` are described in Figure 3. We also use *Guard* = TRUE and *Spec* = `Spec` (shown in Figure 3).

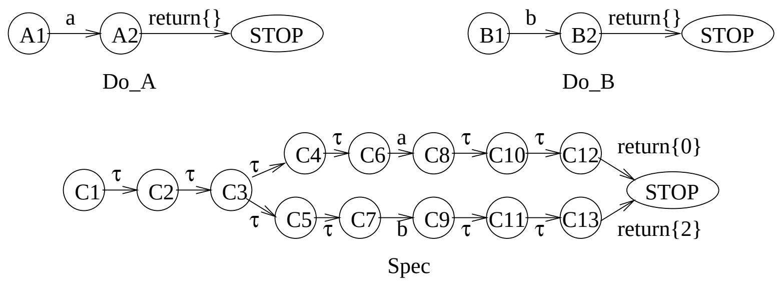

Figure 3. The LTSs in the assumption PAs for `do_a` and `do_b`. The return action with return value `void` is denoted by **`return`**{}.

5.1. Control Flow Automata

The construction of $\mathcal{MP}$ begins with the construction of the control flow automaton (CFA) of $\mathcal{C}$. 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. Intuitively, the CFA can be obtained by viewing the CFG of $\mathcal{C}$ as an automata. Therefore, the CFA is a conservative abstraction of $\mathcal{C}$'s control flow, i.e., it allows a superset of the possible traces of $\mathcal{C}$. 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.

– $\mathcal{L} : S_{CF} \setminus \{\text{FINAL}\} \to \mathit{Stmt}$ 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 $\mathcal{C}$ and $(s_1, s_2) \in T_{CF}$ iff one of the following conditions hold:

– $\mathcal{L}(s_1)$ is an assignment or call-site with $\mathcal{L}(s_2)$ as its unique successor.

– $\mathcal{L}(s_1)$ is a `goto` with $\mathcal{L}(s_2)$ as its target.

– $\mathcal{L}(s_1)$ 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 = FINAL.

EXAMPLE 1. *The CFA of our example program is shown in Figure 4. 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 4 are* `S0, ..., S9,` FINAL *with* `S0` *being the initial state.*

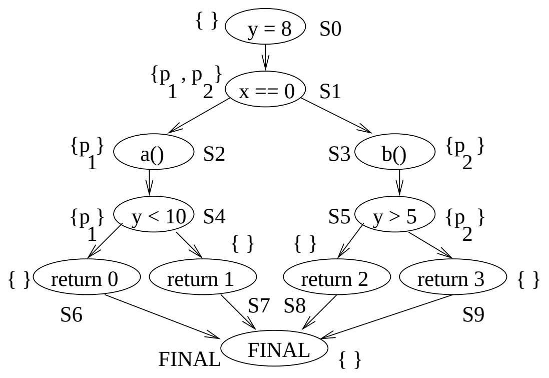

Figure 4. The CFA for our example program. Each non-FINAL state is labeled the same as its corresponding statement. The initial state is labeled `S0`. The states are also labeled with inferred predicates when $\mathcal{P} = \{p_1, p_2\}$ where $p_1 = (y < 10)$ and $p_2 = (y > 5)$. Note that, in general, the set of inferred predicates are not necessarily equal at all states of the CFA. Rather they are computed as the weakest preconditions of the inferred predicates at their successor states.

5.2. Predicate Inference

Since the construction of $\mathcal{MP}$ from $\mathcal{C}$ involves predicate abstraction, it is parameterized by a set of predicates $\mathcal{P}$. We will often indicate this explicitly by referring to $\mathcal{MP}$ as $\mathcal{MP}(\mathcal{P})$. In particular we will write $\mathcal{MP}(\mathcal{P}_1)$ and $\mathcal{MP}(\mathcal{P}_2)$ to denote the LTSs obtained via predicate abstraction from $\mathcal{C}$ using two sets of predicates $\mathcal{P}_1$ and $\mathcal{P}_2$. 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 statements in $\mathcal{C}$. Therefore we sometimes refer to $\mathcal{P}$ or subsets of $\mathcal{P}$ simply as a set of branches. The construction of $\mathcal{MP}$ associates with each state s of the CFA a finite subset of *Expr* 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 5. Note that $\mathcal{P}_{\text{FINAL}}$ is always $\emptyset$.

The algorithm uses a procedure for computing the *weakest precondition* $\mathcal{WP}$ [12, 31, 39] of a predicate p relative to a given statement.

Consider a C assignment statement a of the form $v = e$. Let φ be a C expression. 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 on the LHS of a since we have disallowed such constructs from appearing in $\mathcal{C}$ (see Section 3.2).

```
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 $\mathcal{L}(s) \in \mathcal{P}$, 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 $r \in Act_S \cap IntRet$
      Add the expression $(e == RetVal(r))$ to $\mathcal{P}_s$
  If no $\mathcal{P}_s$ was modified in the 'for' loop, exit
```

Figure 5. Algorithm *PredInfer* for predicate inference.

The weakest precondition is clearly an element of *Expr* as well. The purpose of predicate inference is to create $\mathcal{P}_s$'s that lead to a very precise abstraction of the program relative to the predicates in $\mathcal{P}$. Intuitively, this is how it works. Let $s, t \in S_{CF}$ be such that $\mathcal{L}(s)$ is an assignment statement and $(s,t) \in T_{CF}$. Suppose a predicate p_t gets inserted in $\mathcal{P}_t$ at some point during the execution of *PredInfer* and suppose $p_s = \mathcal{WP}[\mathcal{L}(s)]\{p_t\}$. Now consider any execution state of $\mathcal{C}$ where the control has reached $\mathcal{L}(t)$ after the execution of $\mathcal{L}(s)$. It is obvious that p_t will be true in this state iff p_s was true before the execution of $\mathcal{L}(s)$. In terms of the CFA, this means that the value of p_t after a transition from s to t can be determined precisely on the basis of the value of p_s before the transition. This motivates the inclusion of p_s in $\mathcal{P}_s$. The cases in which $\mathcal{L}(s)$ is not an assignment statement can be explained analogously.

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 in the next section. Irrespective of whether *PredInfer* was terminated forcefully or not, $\mathcal{C}$ is guaranteed to conform to $\mathcal{MP}$. We have found this approach to be very effective in practice. Similar algorithms have been proposed by Namjoshi et al. [29, 46].

EXAMPLE 2. *Consider the CFA described in Example 1. Suppose* $\mathcal{P}$ *contains the two branches* `S4` *and* `S5`*. Then PredInfer begins with* $\mathcal{P}_{\mathtt{S4}} = \{(y < 10)\}$ *and* $\mathcal{P}_{\mathtt{S5}} = \{(y > 5)\}$*. From this it obtains* $\mathcal{P}_{\mathtt{S2}} = \{(y < 10)\}$ *and* $\mathcal{P}_{\mathtt{S3}} = \{(y > 5)\}$*. This leads to* $\mathcal{P}_{\mathtt{S1}} = \{(y < 10), (y > 5)\}$*. Then* $\mathcal{P}_{\mathtt{S0}} = \{\mathcal{WP}[y = 8]\{y < 10\}, \mathcal{WP}[y = 8]\{y > 5\}\} = \{(8 < 10), (8 > 5)\}$*. Since we ignore predicates that are trivially* TRUE *or* FALSE*,* $\mathcal{P}_{\mathtt{S0}} = \emptyset$*. Since the return actions in* `Spec` *have return values* $\{0, 2\}$*,* $\mathcal{P}_{\mathtt{S6}} = \{(0 == 0), (0 == 2)\}$*, which is again* $\emptyset$*. Similarly,* $\mathcal{P}_{\mathtt{S7}} = \mathcal{P}_{\mathtt{S8}} = \mathcal{P}_{\mathtt{S9}} = \mathcal{P}_{\mathrm{FINAL}} = \emptyset$*. Figure 4 shows the CFA with each state* s *labeled by* $\mathcal{P}_s$*.*

So far we have described a method for computing the CFA and a set of predicates associated with each state of the CFA. The states of $\mathcal{MP}$ correspond to the different states of the CFA along with various possible valuations of the predicates inferred at these states. We now define the notions of a predicate valuation and its concretization formally.

DEFINITION 6 (**Predicate valuation and concretization**). *For a CFA node* s *suppose* $\mathcal{P}_s = \{p_1, \ldots, p_k\}$*. Then a valuation of* $\mathcal{P}_s$ *is a Boolean vector* $v_1, \ldots, v_k$*. Let* $\mathcal{V}_s$ *be the set of all valuations of* $\mathcal{P}_s$*. The predicate concretization function* $\Gamma_s : \mathcal{V}_s \rightarrow Expr$ *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* $p_i^{\mathrm{TRUE}} = p_i$ *and* $p_i^{\mathrm{FALSE}} = \neg p_i$*. As a special case, if* $\mathcal{P}_s = \emptyset$*, then* $\mathcal{V}_s = \{\perp\}$ *and* $\Gamma_s(\perp) =$ TRUE*.*

EXAMPLE 3. *Consider the CFA described in Example 1 and the inferred predicates as explained in Example 2. Recall that* $\mathcal{P}_{\mathtt{S1}} = \{(y < 10), (y > 5)\}$*. Suppose* $V_1 = \{0, 1\}$ *and* $V_2 = \{1, 0\}$*. Then* $\Gamma_{\mathtt{S1}}(V_1) = (\neg(y < 10)) \wedge (y > 5)$ *and* $\Gamma_{\mathtt{S1}}(V_2) = (y < 10) \wedge (\neg(y > 5))$*.*

As mentioned before, the states of $\mathcal{MP}$ correspond to the states of the CFA and possible valuations of the predicates associated with these states. In other words, one can view the states of $\mathcal{MP}$ as being obtained by *splitting up* each state s of the CFA in accordance with various valuations of the predicate set $\mathcal{P}_s$ inferred at s. We now define this process more formally.

DEFINITION 7 (**Splitting up states of the CFA**). *Each state $s \in S_{CF}$ gives rise to a set of states of $\mathcal{MP}$, denoted by $\mathcal{S}_s$. In addition, $\mathcal{MP}$ has a unique initial state* INIT*. The definition of $\mathcal{S}_s$ consists of the following sub-cases:*

- $\mathcal{S}_{\text{FINAL}} = \{\text{STOP}\}$.
- *If $\mathcal{L}(s)$ is an assignment, branch,* `goto` *or* `return` *statement, then $\mathcal{S}_s = \{s\} \times \mathcal{V}_s$.*
- *Suppose $\mathcal{L}(s)$ is a call-site for a 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 $P_i = (S_i, init_i, Act_i, T_i)$. Then $\mathcal{S}_s = (\cup_{i=1}^{n} \{s\} \times \mathcal{V}_s \times S_i) \cup (\{s\} \times \mathcal{V}_s)$.*

In the rest of this article we shall refer to states of $\mathcal{MP}$ of the form (s, V) as *normal* states. Also we shall call states of $\mathcal{MP}$ of the form (s, V, c) as *inlined* states since these states can be thought of as arising due to inlining of assumed PAs at call-sites. We are now ready to define $\mathcal{MP}$ precisely.

DEFINITION 8 ($\mathcal{MP}$). *Formally, $\mathcal{MP}$ is an LTS $\langle S_I, init_I, Act_I, T_I \rangle$ where:*

– $S_I = \cup_{s \in S_{CF}} \mathcal{S}_s \cup \{\text{INIT}\}$ *is the set of states.*

– $init_I = \text{INIT}$ *is the initial state.*

– Act_I *is the alphabet.*

– $T_I \subseteq S_I \times Act_I \times S_I$ *is the transition relation.*

5.3. Alphabet of $\mathcal{MP}$

So far we have defined every component of $\mathcal{MP}$ except Act_I and T_I. In this section we shall describe Act_I and in the next section present how T_I can be computed. Intuitively the alphabet of $\mathcal{MP}$ contains every event in the alphabet of *Spec* along with the events in the alphabets of inlined LTSs.

Formally, let *CallSites* $\subseteq S_{CF}$ be the set of states of the CFA that are labeled by call-sites. Let *cs* be an arbitrary element of *Call-Sites* such that $\mathcal{L}(cs)$ is a call-site for library routine `lib`. Suppose $\langle g_1, P_1 \rangle, \ldots, \langle g_n, P_n \rangle$ is the guard and LTS list in the assumption PA for `lib`. For $1 \leq i \leq n$, let $P_i = (S_i, init_i, Act_i, T_i)$. Then we define the function *CallAct* as: $CallAct(cs) = \cup_{i=1}^{n} Act_i$. Once we have defined the function *CallAct*, Act_I is simply equal to $(\cup_{cs \in CallSites} CallAct(cs)) \cup Act_S$.

5.4. Computing T_I

We now describe how to compute T_I. Intuitively, we add a transition between two abstract states unless we can prove that there is no transition between their corresponding concrete states. If we cannot prove this, we say that the two states (or the two formulas representing them) are *admissible*. As we shall soon see, 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. Solving it, therefore, requires the use of a theorem prover. In general the 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 [48]) have this characteristic.

Formally, 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 $Adm(\psi_1, \psi_2)$. Otherwise the formulas are inadmissible, denoted by $\neg Adm(\psi_1, \psi_2)$. The computation of T_I consists of several sub-cases and we shall consider each separately.

- **Transitions from** INIT**.** Recall that *Guard* represents guard qualifying the initial states. First, we add a transition $(\mathrm{INIT}, \tau, (I_{CF}, V))$ to T_I iff $Adm(\Gamma_{I_{CF}}(V), Guard)$.

- **Assignments, `goto`s and branches.** Next, $((s_1, V_1), \tau, (s_2, V_2)) \in T_I$ 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 $Adm(\Gamma_{s_1}(V_1), \mathcal{WP}[\mathcal{L}(s_1)]\{\Gamma_{s_2}(V_2)\})$.
  2. $\mathcal{L}(s_1)$ is a branch statement with a branch condition c, and:
     1. Either $\mathcal{L}(s_2)$ is its `then` successor, $Adm(\Gamma_{s_1}(V_1), \Gamma_{s_2}(V_2))$ and $Adm(\Gamma_{s_1}(V_1), c)$.
     2. Or $\mathcal{L}(s_2)$ is its `else` successor, $Adm(\Gamma_{s_1}(V_1), \Gamma_{s_2}(V_2))$ and $Adm(\Gamma_{s_1}(V_1), \neg c)$.
  3. $\mathcal{L}(s_1)$ is a `goto` statement and $Adm(\Gamma_{s_1}(V_1), \Gamma_{s_2}(V_2))$.

- **Handling return statements.** $((s, V), a, \mathrm{STOP}) \in T_I$ iff $\mathcal{L}(s)$ is a `return` statement, a is a return action, and either (i) $\mathcal{L}(s)$ returns the expression e, $a \in IntRet$ and $Adm(\Gamma_s(V), (e == RetVal(a)))$, or (ii) $\mathcal{L}(s)$ returns `void` and $a = VoidRet$. If $\mathcal{L}(s)$ returns the expression e but condition (i) above is not applicable for any $a \in IntRet$, we add $((s, V), \mathit{VoidRet}, \text{STOP})$ to T_I. 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, *VoidRet* is used as the default[2].

- **Handling call-sites.** Suppose $\mathcal{L}(s_1)$ is a call-site for a 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 $\mathit{Adm}(g'_i, \Gamma_{s_1}(V_1))$, then let $P_i = (S_i, \mathit{init}_i, \mathit{Act}_i, T_i)$ and proceed to step 2, otherwise move on to the next i.
  2. Add a transition $((s_1, V_1), \tau, (s_1, V_1, \mathit{init}_i))$ into T_I.
  3. For each transition $(s, a, t) \in T_i$ where $t \neq \text{STOP}$, add a transition $((s_1, V_1, s), a, (s_1, V_1, t))$ into T_I.
  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 $\mathit{Adm}(\Gamma_{s_1}(V_1), \Gamma_{s_2}(V_2))$, add $((S_1, V_1, s), \tau, (s_2, V_2))$ into T_I.
     - For each transition $(s, a, \text{STOP}) \in T_i$ such that $a \in \mathit{IntRet}$ and $\mathit{Adm}(\Gamma_{s_1}(V_1), \mathcal{WP}[\texttt{x} = \mathit{RetVal}(a)]\{\Gamma_{s_2}(V_2)\})$, add $((S_1, V_1, s), \tau, (s_2, V_2))$ into T_I.
  5. If $\mathcal{L}(s_1)$ is a call-site without an assignment, i.e., of the form `lib(...)`, then for each transition $(s, a, \text{STOP}) \in T_i$ such that $\mathit{Adm}(\Gamma_{s_1}(V_1), \Gamma_{s_2}(V_2))$, add $((S_1, V_1, s), \tau, (s_2, V_2))$ into T_I.

Clearly, $|S_I|$ is exponential in $|\mathcal{P}|$, as are the worst case space and time complexities of constructing $\mathcal{MP}$.

EXAMPLE 4. *Recall the CFA from Example 1 and the predicates corresponding to CFA nodes discussed in Example 2. The $\mathcal{MP}$'s obtained with $\mathcal{P} = \emptyset$ and $\mathcal{P} = \{(y < 10), (y > 5)\}$ are shown in Figure 6(a) and 6(b) respectively.*

[2] In reality MAGIC uses a special non-simulatable action in such situations to maintain soundness. We use *VoidRet* in this article for the sake of simplicity

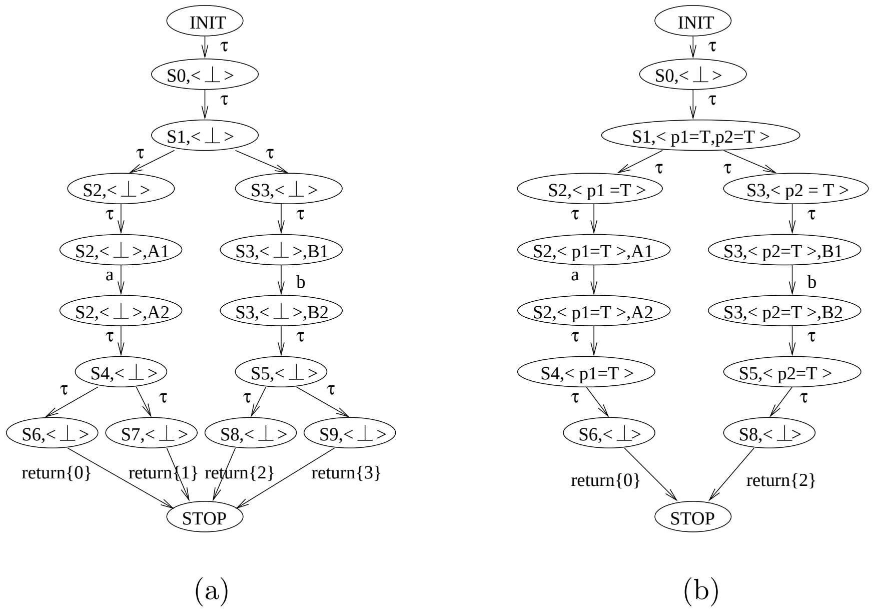

Figure 6. (a) example $\mathcal{MP}$ with $\mathcal{P} = \emptyset$; (b) example $\mathcal{MP}$ with $\mathcal{P} = \{p_1, p_2\}$ where $p_1 = (y < 10)$ and $p_2 = (y > 5)$.

6. Predicate Minimization

Given a counterexample *CE* to the trace containment check, i.e., a trace of $\mathcal{MP}$ that is not a trace of *Spec*, we have to perform the following steps: (i) check if *CE* is a real counterexample, i.e., whether it is a concrete trace of $\mathcal{C}$, and (ii) if *CE* is not a real counterexample, construct a minimal set of predicates that will prevent *CE*, *along with every other spurious counterexample discovered so far*, from arising in future iterations of the two-level CEGAR loop. In this section, we present precisely how these two steps are performed by MAGIC [15]. We begin with the definition of a counterexample.

DEFINITION 9 (**Counterexample**). *A counterexample is a finite sequence* $\langle \hat{s}_1, a_1, \hat{s}_2, a_2, \ldots, a_{n-1}, \hat{s}_n \rangle$ *such that:*

– *For* $1 \le i \le n$, $\hat{s}_i \in S_I$

– *For* $1 \le i < n$, $a_i \in Act_I$

– $\hat{s}_1 = \text{INIT}$

– *For* $1 < i < n$, $\hat{s}_i$ *is of the form* (s_i, V_i) *or* (s_i, V_i, c_i)

– $\hat{s}_n$ *is of the form* (s_i, V_i) *or* (s_i, V_i, c_i) *or* $\hat{s}_n = \text{STOP}$

– *For* $1 \leq i < n$, $\hat{s}_i \xrightarrow{a_i} \hat{s}_{i+1}$

6.1. Counterexample Checking

The *CECheck* algorithm, described in Figure 7, takes $\mathcal{C}$ and a counterexample *CE* as inputs and returns TRUE if *CE* is a valid counterexample of $\mathcal{C}$. Intuitively it computes the weakest precondition of *CE* and then checks if this weakest precondition is satisfiable. This is a backward traversal based algorithm. There is an equivalent algorithm [13] that is based on a forward traversal and uses strongest postconditions instead of weakest preconditions.

```
Input: A counterexample $CE = \langle \hat{s}_1, a_1, \hat{s}_2, a_2, \ldots, a_{n-1}, \hat{s}_n \rangle$ of $\mathcal{MP}$
Output: TRUE iff $CE$ is valid (can be simulated
        on the concrete system)
Variable: $X$ of type formula
Initialize: $X$ := TRUE
For $i$ = $n-1$ to 1
  If $\hat{s}_i$ is of the form $(s,V)$
   If $\mathcal{L}(s)$ is an assignment
    $X$ := $\mathcal{WP}[\mathcal{L}(s)]\{X\}$
   Else If $\mathcal{L}(s)$ is a branch with condition $c$
    If (i < n)
     //At this point, we know that $\hat{s}_{i+1}$ must be of the form
     //$(s',V')$ and further that $\mathcal{L}(s')$ must be the 'then' or
     //'else' successor of $\mathcal{L}(s)$
     If $\mathcal{L}(s')$ is the 'then' successor of $\mathcal{L}(s)$, $X$ := $X \wedge c$
     Else $X$ := $X \wedge \neg c$
  Else If $\hat{s}_i$ is of the form $(s,V,c)$
   //At this point, we know that $\mathcal{L}(s)$ is a call-site
   If $\mathcal{L}(s)$ is of the form x = lib(...)
    If ((i < n) and ($\hat{s}_{i+1}$ is of the form $(s',V')$))
     //At this point we know that some return action was
     //inlined at $\hat{s}_i$
     Let $R = \{r \in Act_S \cap IntRet \mid \exists c'$ s.t. $c \xrightarrow{r} c'\}$
     $X = \vee_{r \in R}\ \mathcal{WP}[x = RetVal(r)]\{X\}$
  If ($X \equiv$ FALSE) return FALSE
Return TRUE
```

Figure 7. Algorithm *CECheck* to check the validity of a counterexample of $\mathcal{C}$.

6.2. Counterexample Elimination

As we shall see in the next section, the process of predicate minimization requires us to solve the following problem: given a spurious counterexample *CE* and a set of branches $\mathcal{P}$, determine if $\mathcal{P}$ eliminates *CE*. This can be achieved in two broad steps: (i) construct $\mathcal{MP}(\mathcal{P})$ and (ii) determine if there exists a counterexample CE' of $\mathcal{MP}(\mathcal{P})$ such that *CE* is *consistent* with CE'. Algorithm *CEEliminate*, described in Figure 8, formally presents how these two steps can be performed. Note that, in practice, *CEEliminate* can proceed in an on-the-fly manner without constructing the full $\mathcal{MP}(\mathcal{P})$ upfront.

To understand *CEEliminate* we need to understand the concept of consistency between states. Let $\mathcal{P}_1$ and $\mathcal{P}_2$ be two sets of branches of $\mathcal{C}$. Let $\mathcal{MP}(\mathcal{P}_1) = \langle S_1, init_1, Act_I, T_1 \rangle$ and $\mathcal{MP}(\mathcal{P}_2) = \langle S_2, init_2, Act_I, T_2 \rangle$ be the two LTSs obtained by predicate abstraction using $\mathcal{P}_1$ and $\mathcal{P}_2$ respectively. Let $s_1 \in S_1$ and $s_2 \in S_2$ be two arbitrary states of $\mathcal{MP}(\mathcal{P}_1)$ and $\mathcal{MP}(\mathcal{P}_2)$ respectively. Recall that either $s_1 = \mathrm{INIT}$ or $s_1 = \mathrm{STOP}$ or s_1 is of the form (s, V) or of the form (s, V, c) where s is a state of the CFA, $V \in \mathcal{V}_s$ is a predicate valuation and c is an inlined LTS state. The same holds true for s_2 as well.

Intuitively s_1 and s_2 are consistent if they differ at most in their predicate valuations. Formally, s_1 and s_2 are said to be consistent (denoted by $Cons(s_1, s_2)$) iff one of the conditions hold:

– $s_1 = s_2 = \mathrm{INIT}$

– $s_1 = s_2 = \mathrm{STOP}$

– $s_1 = (s, V_1)$ and $s_2 = (s, V_2)$ for some V_1 and V_2

– $s_1 = (s, V_1, c)$ and $s_2 = (s, V_2, c)$ for some V_1 and V_2

6.3. Minimizing the Eliminating Set

In this section we solve the following problem: given a set of spurious counterexamples T and a set of candidate predicates $\widehat{\mathcal{P}}$, find a minimal set $\widehat{\mathcal{P}}_{min} \subseteq \widehat{\mathcal{P}}$ which eliminates all the counterexamples in T. Note that, in our context, T will contain every spurious counterexample encountered so far in the CEGAR loop, while $\widehat{\mathcal{P}}$ will be all the branches of $\mathcal{C}$. We present a three step algorithm for solving this problem. **First**, find a mapping $T \mapsto 2^{2^{\widehat{\mathcal{P}}}}$ between each counterexample in T and the set of subsets $\widehat{\mathcal{P}}$ that eliminate it. This can be achieved by iterating through every $\widehat{\mathcal{P}}_{sub} \subseteq \widehat{\mathcal{P}}$ and $CE \in T$, using *CEEliminate* to determine

```
Input: Spurious counterexample $CE = \langle \hat{s}_1, a_1, \hat{s}_2, a_2, \ldots, a_{n-1}, \hat{s}_n \rangle$,
       set of predicates $\mathcal{P}$
Output: TRUE if $CE$ is eliminated by $\mathcal{P}$ and FALSE otherwise
Compute $\mathcal{MP}(\mathcal{P}) = \langle S_{\mathcal{P}}, I_{\mathcal{P}}, Act_{\mathcal{P}}, T_{\mathcal{P}} \rangle$
Variable: $X, Y$ of type subset of $S_{\mathcal{P}}$
Initialize: $X := \{\text{INIT}\}$
If $(X = \emptyset)$ return TRUE
For $i$ = 2 to $n$ do
   $Y$ := $\{\hat{s}' \in S_{\mathcal{P}} \mid Cons(\hat{s}_i, \hat{s}') \wedge \exists \hat{s} \in X$ s.t. $\hat{s} \xrightarrow{a_{i-1}} \hat{s}'\}$
   If $(Y = \emptyset)$ return TRUE
   $X := Y$
Return FALSE
```

Figure 8. Algorithm *CEEliminate* to check if a spurious counterexample can be eliminated.

if $\widehat{\mathcal{P}}_{sub}$ can eliminate CE. This approach is exponential in $|\widehat{\mathcal{P}}|$ but below we list several ways to reduce the number of attempted combinations[3]:

– Limit the size of attempted combinations to a small constant, e.g. 5, assuming that most counterexamples can be eliminated by a small set of predicates.

– Stop after reaching a certain size of combinations if any eliminating solutions have been found.

– Break up the CFG into blocks and only consider combinations of predicates within blocks (keeping combinations in other blocks fixed).

– For any $CE \in T$, if a set $\widehat{\mathcal{P}}'_{sub}$ eliminates CE, ignore all supersets of $\widehat{\mathcal{P}'}_{sub}$ with respect to CE (as we are seeking a minimal solution). This last optimization preserves optimality in all cases.

Second, encode each predicate $p_i \in \widehat{\mathcal{P}}$ with a new Boolean variable p_i^b. We use the terms 'predicate' and 'the Boolean encoding of the predicate' interchangeably. **Third**, derive a Boolean formula σ, based on the predicate encoding, that represents all the possible combinations of predicates that eliminate the elements of T. We use the following notation in the description of σ. Let $CE \in T$ be a counterexample:

– k_{CE} denotes the number of sets of predicates that eliminate CE $(1 \leq k_{CE} \leq 2^{|\widehat{\mathcal{P}}|})$.

[3] While some of these reductions may cause sub-optimality, our experimental results indicate that this seldom occurs in practice.

- $s(CE, i)$ denotes the i-th set ($1 \leq i \leq k_{CE}$) of predicates that eliminates CE. We use the same notation for the conjunction of the predicates in this set.

The formula σ is defined as follows:

$$\sigma \stackrel{\text{def}}{=} \bigwedge_{CE \in T} \bigvee_{i=1}^{k_{CE}} s(CE, i) \tag{1}$$

For any satisfying assignment to σ, the predicates whose Boolean encodings are assigned TRUE are sufficient for eliminating all elements of T. Note that the above described process requires us to store all counterexamples found so far. This is potentially expensive memory-wise, especially if the verification takes a large number of iterations and the counterexamples are large. However our experiments (see Figures 11, 12 and 13) indicate that the memory needed to store the counterexamples does not become a bottleneck in practical scenarios.

From the various possible satisfying assignments to σ, we look for the one with the smallest number of positive assignments. This assignment represents the minimal number of predicates that are sufficient for eliminating T. One is initially inclined to solve σ via 0-1 Integer Linear Programming (ILP) since we are essentially trying to optimize a function given some constraints over Boolean variables. However this is not directly possible because ILP only allows constraints that are implicitly conjuncted while σ includes disjunctions. If we attempt to expand out σ into a conjuncted form using standard distributive laws, we could face an exponential blowup in the size of σ. We therefore use PBS [9], a solver for Pseudo Boolean Formulas, to solve σ directly.

A pseudo-Boolean formula is of the form $\sum_{i=1}^{n} c_i \cdot b_i \bowtie k$, where b_i is a Boolean variable, c_i is a rational constant for $1 \leq i \leq n$, k is a rational constant and $\bowtie$ represents one of the inequality or equality relations ($\{<, \leq, >, \geq, =\}$). Each such constraint can be expanded to a CNF formula (hence the name pseudo-Boolean), but this expansion can be exponential in n. PBS does not perform this expansion, but rather uses an algorithm designed in the spirit of the Davis-Putnam-Loveland algorithm that handles these constraints directly. PBS accepts as input standard CNF formulas augmented with pseudo-Boolean constraints. Given an objective function in the form of a pseudo-Boolean formula, PBS finds an optimal solution by repeatedly tightening the constraint over the value of this function until it becomes unsatisfiable. That is, it first finds a satisfying solution and calculates the value of the objective function according to this solution. It then adds a constraint that the value of the objective function should be smaller by one[4]. This process is repeated until the formula becomes unsatisfiable. The objective function in our case is to minimize the number of chosen predicates (by minimizing the number of variables that are assigned TRUE):

$$\min \sum_{i=1}^{n} p_i^b \tag{2}$$

EXAMPLE 5. *Suppose that the counterexample CE_1 is eliminated by either $\{p_1, p_3, p_5\}$ or $\{p_2, p_5\}$ and that the counterexample CE_2 can be eliminated by either $\{p_2, p_3\}$ or $\{p_4\}$. The objective function is* $\min \sum_{i=1}^{5} p_i^b$ *and is subject to the constraint:*

$$\begin{aligned}\sigma \ = \ &((p_1^b \wedge p_3^b \wedge p_5^b) \vee (p_2^b \wedge p_5^b)) \wedge \\ &((p_2^b \wedge p_3^b) \vee (p_4^b))\end{aligned}$$

The minimal satisfying assignment in this case is $p_2^b = p_5^b = p_4^b =$ TRUE. □

Other techniques for solving this optimization problem are possible, including minimal hitting sets and logic minimization. The PBS step, however, has not been a bottleneck in any of our experiments.

7. Action-Guided Abstraction

We now present a CEGAR on LTSs [17]. Let $\mathcal{MP} = \langle S_I, init_I, Act_I, T_I \rangle$ be an LTS obtained via predicate abstraction from a component $\mathcal{C}$, as described in Section 5. We first create an LTS $\mathcal{MA}^0 = \langle S_A^0, init_A^0, Act_I, T_A^0 \rangle$ such that (i) $L(\mathcal{MP}) \subseteq L(\mathcal{MA}^0)$ and (ii) $\mathcal{MA}^0$ contains at most as many states as $\mathcal{MP}$ (and typically many fewer). Given an abstraction $\mathcal{MA} = \langle S_A, init_A, Act_I, T_A \rangle$ of $\mathcal{MP}$ and a spurious trace $\pi \in L(\mathcal{MA}) \setminus L(\mathcal{MP})$, our refinement procedure produces a refined abstraction $\mathcal{MA}' = \langle S'_A, init'_A, Act_I, T'_A \rangle$ such that (i) $L(\mathcal{MP}) \subseteq L(\mathcal{MA}') \subset L(\mathcal{MA})$, (ii) $\pi \notin L(\mathcal{MA}')$, and (iii) $\mathcal{MA}'$ contains at most as many states as $\mathcal{MP}$. It is important to note that we require throughout that $\mathcal{MP}$, $\mathcal{MA}^0$, $\mathcal{MA}$, and $\mathcal{MA}'$ all share the same alphabet. We also remark that iterating this refinement procedure must converge in a finite number of steps to an LTS that accepts the same language as $\mathcal{MP}$.

Let us write $B = \langle S_B, init_B, Act_I, T_B \rangle$ to denote a generic abstraction of $\mathcal{MP}$. States of B are called *abstract* states, whereas states of

[4] A possible improvement is to do a binary search. In none of our experiments, however, was this stage a bottleneck.

$\mathcal{MP}$ are called *concrete* states. In the context of action-guided abstraction, abstract states are always disjoint sets of concrete states that partition S_I, and our abstraction refinement step corresponds precisely to a refinement of the partition. For $s \in S_I$ a concrete state, the unique abstract state of B to which s belongs is denoted by $[s]_B$.

In any abstraction B that we generate, a partition S_B of the concrete states of $\mathcal{MP}$ uniquely determines the abstract model B: the initial state $init_B$ of B is simply $[init_I]_B$, and for any pair of abstract states $u, v \in S_B$ and any action $a \in Act_I$, we include a transition $u \xrightarrow{a} v \in T_B$ iff there exist concrete states $s \in u$ and $t \in v$ such that $s \xrightarrow{a} t$. This construction is an instance of an *existential abstraction* [24]. It is straightforward to show that it is sound, i.e., that $L(\mathcal{MP}) \subseteq L(B)$ holds for any abstraction B.

The initial partition S_A^0 of concrete states identifies two states $s, t \in S_I$ if they share the same set of immediately enabled actions: $t \in [s]_A^0$ iff $export(t) = export(s)$. We then let $S_A^0 = \{[s]_A^0 \mid s \in S_I\}$. Again, this uniquely defines our initial abstraction $\mathcal{MA}^0$, the construction marked † in Figure 1 (c.f. Section 4).

8. Action-Guided Refinement

In order to describe the refinement step, we need an auxiliary definition. Given an abstract state $u \in S_B$ and an action $a \in Act_I$, we construct a refined partition $S'_B = Split(S_B, u, a)$ of S_I which agrees with S_B outside of u, but distinguishes concrete states in u if they have different abstract a-successors in S_B. More precisely, for any $s \in S_I$, if $s \notin u$, we let $[s]_{B'} = [s]_B$. Otherwise, for $s, t \in u$, we let $[s]_{B'} = [t]_{B'}$ iff $\bigcup\{[s']_B \mid s' \in Reach(\mathcal{MP}, s, a)\} = \bigcup\{[t']_B \mid t' \in Reach(\mathcal{MP}, t, a)\}$. We then let $Split(S_B, u, a) = \{[s]_{B'} | s \in S_I\}$. This refined partition uniquely defines a new abstraction, which we write $Abs(Split(S_B, u, a))$. Note that in order to compute the transition relation of $Abs(Split(S_B, u, a))$ it suffices to adjust only those transitions in T_B that have u either as a source or a target.

The refinement step takes as input a spurious counterexample $\pi \in L(\mathcal{MA}) \backslash L(\mathcal{MP})$ and returns a refined abstraction $\mathcal{MA}'$ which does not accept π. This is achieved by repeatedly splitting states of $\mathcal{MA}$ along abstract paths which accept π. The algorithm in Figure 9 (marked ‡ in Figure 1) describes this procedure in detail.

THEOREM 2. *The algorithm described in Figure 9 is correct and always terminates.*

```
Input: abstraction $\mathcal{MA}$ of $\mathcal{MP}$ (with $L(\mathcal{MP}) \subseteq L(\mathcal{MA})$) and
       trace $\pi = a_1 \dots a_m \in L(\mathcal{MA}) \setminus L(\mathcal{MP})$
Output: refined abstraction $\mathcal{MA}'$
        (with $L(\mathcal{MP}) \subseteq L(\mathcal{MA}') \subset L(\mathcal{MA})$) and $\pi \notin L(\mathcal{MA}')$

while there exists some abstract path $u_0 \xrightarrow{a_1} \dots \xrightarrow{a_m} u_m$ in $\mathcal{MA}$ do
   let $\mathit{reachable\_states} = \{\mathit{init}\}$ //$\mathit{init}$ = initial state of $\mathcal{MP}$
   let $j = 1$
   while $\mathit{reachable\_states} \neq \emptyset$ do
      let $\mathit{reachable\_states} = \mathit{Reach}(\mathcal{MP}, \mathit{reachable\_states}, a_j) \cap u_j$
      let $j = j + 1$
   endwhile
   let $\mathcal{MA} = \mathit{Abs}(\mathit{Split}(S_A, u_{j-2}, a_{j-1}))$ //$S_A$ = states of $\mathcal{MA}$
endwhile
let $\mathcal{MA}' = \mathcal{MA}$
return $\mathcal{MA}'$.
```

Figure 9. Action-guided CEGAR algorithm on LTS.

PROOF 1. *We first note that it is obvious that whenever the algorithm terminates it returns an abstraction $\mathcal{MA}'$ with $\pi \notin L(\mathcal{MA}')$. It is equally clear, since $\mathcal{MA}'$ is obtained via successive refinements of $\mathcal{MA}$, that $L(\mathcal{MP}) \subseteq L(\mathcal{MA}') \subset L(\mathcal{MA})$. It remains to show that every splitting operation performed by the algorithm results in a proper partition refinement; termination then follows from the fact that the set of states of $\mathcal{MP}$ is finite.*

Observe that, since $\pi \notin L(\mathcal{MP})$, $\mathit{Reach}(\mathcal{MP}, \mathit{init}, \pi) = \emptyset$, and therefore the inner while loop always terminates. At that point, we claim that (i) there is an abstract transition $u_{j-2} \xrightarrow{a_{j-1}} u_{j-1}$; (ii) there are some concrete states in u_{j-2} reachable (in $\mathcal{MP}$) from init; and (iii) none of these reachable concrete states have concrete a_{j-1}-successors in u_{j-1}. Note that (ii) follows from the fact that the inner loop is entered with $\mathit{reachable\_states} = \{\mathit{init}\}$, whereas (i) and (iii) are immediate. Because of the existential definition of the abstract transition relation, we conclude that u_{j-2} contains two kinds of concrete states: some having concrete a_{j-1}-successors in u_{j-1}, and some not. Splitting the state u_{j-2} according to action a_{j-1} therefore produces a proper refinement. □

We remark again that each splitting operation corresponds to a unit step of the Paige-Tarjan algorithm [49]. Iterating our refinement procedure therefore converges to the bisimulation quotient of $\mathcal{MP}$. Note however that, unlike the Paige-Tarjan algorithm, our refinement process is counterexample driven and not aimed at computing the bisimulation quotient. In practical verification instances, we usually stop well before reaching this quotient.

We stress that the CEGAR algorithm described in Figure 1 never invokes the above abstraction refinement routine with the full parallel composition $\mathcal{MA} = \mathcal{MA}_1 || \ldots || \mathcal{MA}_n$ as input. Indeed, this would be very expensive, since the size of the global state-space grows exponentially with the number of concurrent processes. It is much cheaper to take advantage of compositionality: by Theorem 1, $\pi \in L(\mathcal{MA}_1 || \ldots || \mathcal{MA}_n) \setminus L(\mathcal{MP}_1 || \ldots || \mathcal{MP}_n)$ iff, for some i, $\pi \restriction Act_i \in L(\mathcal{MA}_i) \setminus L(\mathcal{MP}_i)$. It then suffices to apply abstraction refinement to this particular $\mathcal{MA}_i$, since $\pi \restriction Act_i \notin L(\mathcal{MA}'_i)$ implies that $\pi \notin L(\mathcal{MA}_1 || \ldots || \mathcal{MA}'_i || \ldots || \mathcal{MA}_n)$. The advantage of this approach follows from the fact that the computational effort required to identify $\mathcal{MA}_i$ grows only linearly with the number of concurrent components.

9. Experimental Evaluation

We implemented our technique inside MAGIC and experimented with three broad goals in mind. The first goal was to check the effectiveness of predicate minimization by itself on purely sequential benchmarks. The second goal was to compare the overall effectiveness of the proposed two-level CEGAR approach, particularly insofar as memory usage is concerned. The third goal was to verify the effectiveness of our action-guided abstraction scheme by itself. We carried out experiments with a wide range of benchmarks, both sequential and concurrent. Each benchmark consisted of an implementation (a C program) and a specification (provided separately as an LTS). All of the experiments were carried out on an AMD Athlon 1600 XP machine with 900 MB RAM running RedHat 7.1.

```
**Input:** Set of predicates $\mathcal{P}$
**Output:** Subset of $\mathcal{P}$ that eliminates all spurious
        counterexamples so far
**Variable:** $X$ of type set of predicates
LOOP: Create a random ordering $\langle p_1, \ldots, p_k \rangle$ of $\mathcal{P}$
For $i$ = 1 to $k$ do
  $X := \mathcal{P} \setminus \{p_i\}$
  If $X$ can eliminate every spurious counterexample seen so far
    $\mathcal{P} := X$
    Goto LOOP
Return $\mathcal{P}$
```

Figure 10. Greedy predicate minimization algorithm.

9.1. Predicate Minimization Results

In this section we describe our results in the context of the first of the three goals mentioned above, i.e., checking the effectiveness of predicate minimization by itself. We also present results comparing our predicate minimization scheme with a greedy predicate minimization strategy implemented on top of MAGIC. In each iteration, this greedy strategy first adds predicates sufficient to eliminate the spurious counterexample to the predicate set $\mathcal{P}$. Then it attempts to reduce the size of the resulting $\mathcal{P}$ by using the algorithm described in Figure 10. The advantage of this approach is that it requires only a small overhead (polynomial) compared to *Sample-and-Eliminate*, but on the other hand it does not guarantee an optimal result. Further, we performed experiments with Berkeley's BLAST [37] tool. BLAST also takes C programs as input, and uses a variation of the standard CEGAR loop based on *lazy abstraction*, but without minimization. Lazy abstraction refines an abstract model while allowing different degrees of abstraction in different parts of a program, without requiring recomputation of the entire abstract model in each iteration. Laziness and predicate minimization are, for the most part, orthogonal techniques. In principle a combination of the two might produce better results than either in isolation.

9.1.1. *Benchmarks*

We used two kinds of benchmarks. A small set of relatively simple benchmarks were derived from the examples supplied with the BLAST distribution and regression tests for MAGIC. The difficult benchmarks were derived from the C source code of `OpenSSL-0.9.6c`, several thousand lines of code implementing the SSL protocol used for secure transfer of information over the Internet. A critical component of this protocol is the initial *handshake* between a server and a client. We verified different properties of the main routines that implement the handshake. The names of benchmarks that are derived from the server routine and client routine begin with `ssl-srvr` and `ssl-clnt` respectively. In all our benchmarks, the properties are satisfied by the implementation. The server and client routines have roughly 350 lines each but, as our results indicate, are non-trivial to verify. Note that all these benchmarks involved purely sequential C code.

9.1.2. *Results Summary*

Figure 11 summarizes the comparison of our predicate minimization strategy with the greedy approach. Time consumptions are given in seconds. For predicate minimization, instead of solving the full optimization problem, we simplified the problem as described in sec-

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

Figure 11. Comparison of MAGIC with the greedy approach. '*' indicates run-time longer than 3 hours. '×' indicates negligible values. Best results are emphasized.

tion 6. In particular, for each trace we only considered the first 1,000 combinations and only generated 20 eliminating combinations. The combinations were considered in increasing order of size. After all combinations of a particular size had been tried, we checked whether at least one eliminating combination had been found. If so, no further combinations were tried. In the smaller examples we observed no loss of optimality due to these restrictions. We also studied the effect of altering these restrictions on the larger benchmarks and we report our findings later.

Figure 12 shows the improvement observed in MAGIC upon using predicate minimization while Figure 13 shows the comparison between predicate minimization and BLAST. Once again, time consumptions are reported in seconds. The column Iter reports the number of iterations through the CEGAR loop necessary to complete the proof. Predicates are listed differently for the two tools. For BLAST, the first

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

Figure 12. Results for MAGIC with and without minimization. '*' indicates run-time longer than 3 hours. '×' indicates negligible values. Best results are emphasized.

number is the total number of predicates discovered and used and the second number is the number of predicates active at any one point in the program (due to lazy abstraction this may be smaller). In order to force termination we imposed a limit of three hours on the running time. We denote by '*' in the Time column examples that could not be solved in this time limit. In these cases the other columns indicate relevant measurements made at the point of forceful termination.

For MAGIC, the first number is the total number of expressions used to prove the property, i.e., $|\cup_{s \in S_{CF}} \mathcal{P}_s|$. The number of predicates (the second number) may be smaller, as MAGIC combines multiple mutually exclusive expressions (e.g., $x == 1$, $x < 1$, and $x > 1$) into a single, possibly non-binary predicate, having a number of values equal to the number of expressions (plus one, if the expressions do not cover all possibilities.) The final number for MAGIC is the size of the final $\mathcal{P}$. For

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

Figure 13. Results for BLAST and MAGIC with predicate minimization. '*' indicates run-time longer than 3 hours. '×' indicates negligible values. Best results are emphasized.

experiments in which memory usage was large enough to be a measure of state-space size rather than overhead, we also report memory usage (in megabytes).

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

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

		ssl-srvr-4						ssl-srvr-15					
ELM	SUB	Ti	It	$\|\mathcal{P}\|$	M	T	G	Ti	It	$\|\mathcal{P}\|$	M	T	G
50	250	656	8	2	64	34	1	1170	15	3	72	86	1
100	250	656	8	2	64	34	1	1169	15	3	72	86	1
150	250	657	8	2	64	34	1	1169	15	3	72	86	1
200	250	656	8	2	64	34	1	1170	15	3	72	86	1
250	250	656	8	2	64	34	1	1168	15	3	72	86	1

		ssl-clnt-1					
ELM	SUB	Ti	It	$\|\mathcal{P}\|$	M	T	G
50	250	1089	13	2	67	66	1
100	250	1089	13	2	67	66	1
150	250	1090	13	2	67	66	1
200	250	1089	13	2	67	66	1
250	250	1090	13	2	67	66	1

Figure 14. Results for optimality. ELM = MAXELM, SUB = MAXSUB, Ti = Time in seconds, It = number of iterations, M = Memory, T = total number of eliminating subsets generated, and G = maximum size of any eliminating subset generated.

9.1.3. *Optimality*

We experimented with two of the parameters that affect the optimality of our predicate minimization algorithm: (i) the maximum number of examined subsets (MAXSUB) and (ii) the maximum number of eliminating subsets generated (MAXELM) (that is, the procedure stops the search if MAXELM eliminating subsets were found, even if less than MAXSUB combinations were tried). We first kept MAXSUB fixed and took measurements for different values of MAXELM on a subset of our benchmarks viz. `ssl-srvr-4`, `ssl-srvr-15` and `ssl-clnt-1`. Our results, shown in Figure 14, clearly indicate that the optimality is practically unaffected by the value of MAXELM.

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

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

Figure 15. Results for optimality. SUB = MAXSUB, Time is in seconds, It = number of iterations, T = total number of eliminating subsets generated, M = maximum size of subsets tried, and G = maximum size of eliminating subsets generated.

Next we experimented with different values of MAXSUB (the value of MAXELM was set equal to MAXSUB). The results we obtained are summarized in Figure 15. It appears that, at least for our benchmarks, increasing MAXSUB leads only to increased execution time without reduced memory consumption or number of predicates. The additional number of combinations attempted or constraints allowed does not lead to improved optimality. The most probable reason is that, as shown by our results, even though we are trying more combinations, the actual number or maximum size of eliminating combinations generated does not increase significantly. It would be interesting to investigate whether this is a feature of most real-life programs. If so, it would allow us, in most cases, to achieve near optimality by trying out only a small number of combinations or only combinations of small size.

9.2. Two-Level CEGAR Results

In this section we present our results with regard to the effectiveness of our proposed two-level and only action-guided CEGAR schemes. To this end, we carried out experiments on 36 benchmarks, of which 26 were sequential programs and 10 were concurrent programs. Each example was verified twice, once with only the low-level abstraction, and once with the full two-level algorithm. Tests that used only the low-level predicate abstraction refinement scheme are marked by *PredOnly* in our results tables, whereas tests that also incorporated our LTS action-guided abstraction refinement procedure are marked by *Both-Abst*. Both schemes started out with the same initial sets of predicates. For each experiment we measured several quantities: (i) the size of the final state-space on which the property was proved/disproved,[5] (ii) the number of predicate refinement iterations required, (iii) the number of LTS refinement iterations required, (iv) the total number of refinement iterations required, and (v) the total time required. In the tables summarizing our results, these measurements are reported in columns named respectively *St, PIt, LIt, It* and *T*. Note that predicate minimization was turned on during all the experiments described in this section.

LOC	**Description**	**PredOnly**			**BothAbst**		
		St	*It*	*T*	*St*	*It*	*T*
27	*pthread_mutex_lock* (pthread)	26	1	52	16	3	54
24	*pthread_mutex_unlock* (pthread)	27	1	51	13	2	56
60	*socket* (socket)	187	3	1752	44	25	2009
24	*sock_alloc* (socket)	50	2	141	14	4	154
4	*sys_send* (socket)	7	1	92	6	1	93
11	*sock_sendmsg* (socket)	23	1	108	14	3	113
27	modified *pthread_mutex_lock*	23	1	59	14	2	61
24	modified *pthread_mutex_unlock*	27	1	61	12	2	66
24	modified *sock_alloc*	47	1	103	9	1	106
11	modified *sock_sendmsg*	21	1	96	10	1	97

Figure 16. Summary of results for Linux Kernel code. **LOC** and **Description** denote the number of lines of code and a brief description of the benchmark source code. The measurements for *PIter* and *LIter* have been omitted because they are insignificant. All times are in milliseconds.

9.2.1. *Unix Kernel Benchmarks*

The first set of examples was designed to examine how our approach works on a wide spectrum of implementations. The summary of our results on these examples is presented in Figure 16. We chose ten code fragments from the Linux Kernel 2.4.0. Corresponding to each code fragment we constructed a specification from the Linux manual pages. For example, the specification in the third benchmark[6] states that the socket system call either properly allocates internal data structures for

[5] Note that, since our abstraction-refinement scheme produces increasingly refined models, and since we reuse memory from one iteration to the next, the size of the final state-space represents the *total* memory used.

[6] This benchmark was also used as `socket-y` in the predicate minimization experiments described in the previous section.

a new socket and returns 1, or fails to do so and returns an appropriate negative error value.

9.2.2. *OpenSSL Benchmarks*

The next set of examples was aimed at verifying larger pieces of code. Once again we used OpenSSL handshake implementation to design a set of 26 benchmarks. However, unlike the previous OpenSSL benchmarks, some of these benchmarks were concurrent and comprised of both a client and a server component executing in parallel. The specifications were derived from the official SSL design documents. For example, the specification for the first concurrent benchmark states that the handshake is always initiated by the client.

PredOnly			**BothAbst**					**Gain**
St(S1)	*It*	*T*	*St(S2)*	*PIt*	*LIt*	*It*	*T*	*S1/S2*
563	7	127	151	7	191	198	142	3.73
323	9	134	172	9	307	316	156	1.89
362	21	212	214	20	850	870	263	1.69
227	1	25	19	1	0	1	23	11.94
3204	98	1284	878	53	6014	6067	6292	3.65
2614	121	1418	559	113	9443	9556	6144	4.68
2471	40	517	662	34	3281	3315	2713	3.73
2614	60	750	455	37	3158	3195	1992	5.75
402	18	174	176	19	506	525	209	2.28
408	18	194	185	16	651	667	217	2.21
633	51	405	263	58	3078	3136	688	2.41
369	28	232	193	33	987	1020	306	1.91
318	15	166	172	13	398	411	182	1.85
323	20	190	236	21	644	665	242	1.37
323	20	188	160	20	556	576	221	2.02
314	16	168	264	16	570	586	215	1.19

Figure 17. Summary of results for sequential OpenSSL examples. The first eight are server benchmarks while the last eight are client benchmarks. Note that for the **PredOnly** case, *LIt* is always zero and *PIt* = *It*. All times are in seconds.

The first 16 examples are sequential implementations, examining different properties of `SrvrCode` and `ClntCode` separately. Each of these examples contains about 350 comment-free LOC. The results for these are summarized in Figure 17. The remaining 10 examples test various properties of `SrvrCode` and `ClntCode` when executed together. These examples are concurrent and consist of about 700 LOC. The results for them are summarized in Figure 18. All OpenSSL benchmarks other than the seventh server benchmark passed the property. In terms of state-space size, the two-level refinement scheme outperforms the one-level scheme by factors ranging from 2 to 136. The savings for the

PredOnly			**BothAbst**					**Gain**
St(S1)	*It*	*T*	*St(S2)*	*PIt*	*LIt*	*It*	*T*	*S1/S2*
108659	8	243	16960	8	268	276	529	6.41
95535	9	226	15698	9	331	340	608	6.09
69866	24	449	23865	19	828	847	1831	2.93
43811	1	51	323	1	0	1	55	135.64
108659	7	217	16006	6	186	192	384	6.79
162699	12	366	18297	9	375	384	792	8.89
167524	23	599	31250	24	1441	1465	4492	5.36
60602	9	227	17922	10	434	444	852	3.38
313432	115	3431	50274	63	3660	3723	15860	6.23
123520	23	430	23460	21	926	947	2139	5.27

Figure 18. Summary of results for concurrent OpenSSL examples. Note that for the **PredOnly** case, *LIt* is always zero and *PIt* = *It*. All times are in seconds.

concurrent examples are significantly higher than for the sequential ones. We expect these savings to increase with the number of concurrent components in the implementation. The fourth server and the fourth concurrent benchmarks show particular improvement with the two-level approach. In these two benchmarks, the property holds on the very initial abstraction, thereby requiring no refinement and letting us achieve maximum reduction in state-space.

Although our goal of reducing the size of the state-space was achieved, our implementation of the two-level algorithm shows an increase in time over that of the one-level scheme. However, we believe that this situation can be redressed through engineering optimizations of MAGIC. For instance, not only is MAGIC currently based on explicit state enumeration, but also in each iteration it performs the entire verification from scratch. As is evident from our results, the majority of iterations involve LTS refinement. Since the latter only induces a local change in the transition system, the refined model is likely to differ marginally from the previous one. Therefore much of the work done during verification in the previous iteration could be reused.

10. Conclusions and Future Work

Despite significant recent advancement, automated verification of concurrent programs remains an elusive goal. In this paper we presented an approach to automatically and compositionally verify concurrent C programs against safety specifications. These concurrent implementations consist of several sequential C programs which communicate via blocking message-passing. Our approach is an instantiation of the

CEGAR paradigm, and incorporates two levels of abstraction. The first level uses predicate abstraction to handle data while the second level aggregates states according to the values of observable events. In addition, our predicate refinement scheme is aimed at discovering a minimal set of predicates that suffice to prove/disprove the property of interest.

Experimental results with our tool MAGIC suggest that this scheme effectively combats the state-space explosion problem. In all our benchmarks, the two-level algorithm achieved significant reductions in state-space (in one case by over two orders of magnitude) compared to the single-level predicate abstraction scheme. The reductions in the number of predicates required (and thereby in the time and memory consumption) due to our predicate minimization technique were also very encouraging.

We are currently engaged in extending MAGIC to handle the proprietary implementation of a large industrial controller for a metal casting plant. This code consists of over 30,000 lines of C and incorporates up to 25 concurrent threads which communicate through shared variables. Adapting MAGIC to handle shared memory, without sacrificing compositionality, is therefore one of our priorities. Not only will this enable us to test our tool on the many available shared-memory-based benchmarks, but it will also allow us to compare MAGIC with other similar tools (such as BLAST) which also use shared memory for communication.

We also wish to investigate the possibility of performing incremental verification in the context of action-guided abstraction refinement. Since the successive $\mathcal{MA}_i$'s obtained during this process can be expected to differ only marginally from each other, we expect incremental model checking to speed up the verification process by a significant factor. In addition, we are working on extending MAGIC to handle state/event based LTL-like specifications. Lastly, we intend to explore the possibility of adapting the two-level CEGAR scheme to different types of conformance relations such as simulation and bisimulation, so as to handle a wider range of specifications.

Acknowledgements

We thank Rupak Majumdar and Ranjit Jhala for their help with BLAST.

References

1. 'BLAST website'. `http://www-cad.eecs.berkeley.edu/~rupak/blast`.

2. ‘CIL website’. `http://manju.cs.berkeley.edu/cil`.
3. ‘ESC-Java website’. `http://www.research.compaq.com/SRC/esc`.
4. ‘Grammatech, Inc.’. `http://www.grammatech.com`.
5. ‘Java PathFinder website’. `http://ase.arc.nasa.gov/visser/jpf`.
6. ‘MAGIC website’. `http://www.cs.cmu.edu/∼chaki/magic`.
7. ‘SLAM website’. `http://research.microsoft.com/slam`.
8. ‘SPIN website’. `http://spinroot.com/spin/whatispin.html`.
9. Aloul, F., A. Ramani, I. Markov, and K. Sakallah: 2002, ‘PBS: A backtrack search pseudo Boolean solver’. In: *Symposium on the theory and applications os satisfiability testing (SAT)*. pp. 346–353.
10. Anderson, L.: 1994, ‘Program analysis and specialization for the C programming language’. Ph.D. thesis, Datalogisk Intitut, Univ. of Copenhagen, Copenhagen, Denmark.
11. Ball, T., R. Majumdar, T. D. Millstein, and S. K. Rajamani: 2001, ‘Automatic Predicate Abstraction of C Programs’. In: *SIGPLAN Conference on Programming Language Design and Implementation*. pp. 203–213.
12. Ball, T. and S. K. Rajamani: 2001, ‘Automatically Validating Temporal Safety Properties of Interfaces’. In: *Proceedings of SPIN*, Vol. 2057. pp. 103–122.
13. Ball, T. and S. K. Rajamani: 2002, ‘Generating Abstract Explanations of Spurious Counterexamples in C Programs’. Technical Report MSR-TR-2002-09, Microsoft Research, Redmond.
14. Bensalem, S., Y. Lakhnech, and S. Owre: 1998, ‘Computing Abstractions of Infinite State Systems Compositionally and Automatically’. In: *Proceedings of CAV*, Vol. 1427. pp. 319–331.
15. Chaki, S., E. Clarke, A. Groce, and O. Strichman: 2003a, ‘Predicate Abstraction with Minimum Predicates’. In: *Proceedings of CHARME*. To appear.
16. Chaki, S., E. M. Clarke, A. Groce, S. Jha, and H. Veith: 2003b, ‘Modular Verification of Software Components in C’. In: *Proceedings of ICSE*. pp. 385–395.
17. Chaki, S., J. Ouaknine, K. Yorav, and E. Clarke: 2003c, ‘Automated compositional abstraction refinement for concurrent C programs: A two-level approach’. In: *Proceedings of SoftMC*.
18. Clarke, E., O. Grumberg, and D. Peled: 1999, *Model Checking*. MIT Press.
19. Clarke, E., O. Grumberg, M. Talupur, and D. Wang: 2003, ‘Making predicate abstraction efficient: eliminating redundant predicates’. In: *Proceedings of Computer Aided Verification (CAV)*.
20. Clarke, E., A. Gupta, J. Kukula, and O. Strichman: 2002, ‘SAT based Abstraction - Refinement using ILP and Machine Learning Techniques’. In: E. Brinksma and K. Larsen (eds.): *Proceedings of CAV*, Vol. 2404 of *LNCS*. Copenhagen, Denmark, pp. 265–279.
21. Clarke, E. M. and E. A. Emerson: 1982, ‘Synthesis of Synchronization Skeletons from Branching Time Temporal Logic’. In: *Proceedings of the Workshop on Logics of Programs*, Vol. 131. pp. 52–71.
22. Clarke, E. M., E. A. Emerson, and A. P. Sistla: 1986, ‘Automatic verification of finite-state concurrent systems using temporal logic specifications’. *ACM Transactions on Programming Languages and System (TOPLAS)* **8**(2), 244–263.
23. Clarke, E. M., O. Grumberg, S. Jha, Y. Lu, and H. Veith: 2000, ‘Counterexample-Guided Abstraction Refinement’. In: *Proceedings of CAV*, Vol. 1855. pp. 154–169.

24. Clarke, E. M., O. Grumberg, and D. E. Long: 1994, 'Model Checking and Abstraction'. *Proceedings of TOPLAS* pp. 1512–1542.
25. Cobleigh, J. M., D. Giannakopoulou, and C. S. Păsăreanu: 2003, 'Learning Assumptions for Compositional Verification'. In: *Proceedings of TACAS*, Vol. 2619. pp. 331–346.
26. Colón, M. and T. E. Uribe: 1998, 'Generating Finite-State Abstractions of Reactive Systems Using Decision Procedures'. In: *Proceedings of CAV*. pp. 293–304.
27. Corbett, J. C., M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng: 2000, 'Bandera: extracting finite-state models from Java source code'. In: *Proceedings of ICSE*. pp. 439–448.
28. Cousot, P. and R. Cousot: 1977, 'Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints'. In: *Proceedings of the SIGPLAN Conference on Programming Languages*. pp. 238–252.
29. Dams, D. and K. S. Namjoshi: 2003, 'Shape Analysis through Predicate Abstraction and Model Checking'. In: *Proceedings of VMCAI*, Vol. 2575.
30. Das, S., D. L. Dill, and S. Park: 1999, 'Experience with Predicate Abstraction'. In: *Computer Aided Verification*. pp. 160–171.
31. Dijkstra, E. W.: 1973, 'A simple axiomatic basis for programming language constructs'. Lecture notes from the International Summer School on Structured Programming and Programmed Structures.
32. Dwyer, M. B., J. Hatcliff, R. Joehanes, S. Laubach, C. S. Pasareanu, H. Zheng, and W. Visser: 2001, 'Tool-supported program abstraction for finite-state verification'. In: *International Conference on Software engineering*. pp. 177–187.
33. Engler, D., B. Chelf, A. Chou, and S. Hallem: 2000, 'Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions'. In: *Symposium on Operating Systems Design and Implementation*.
34. Graf, S. and H. Saidi: 1997, 'Construction of Abstract State Graphs with PVS'. In: O. Grumberg (ed.): *Computer Aided Verification*, Vol. 1254. pp. 72–83.
35. Havelund, K. and T. Pressburger: 2000, 'Model Checking JAVA Programs using JAVA PathFinder'. *International Journal on Software Tools for Technology Transfer* **2**(4), 366–381.
36. Henzinger, T. A., R. Jhala, R. Majumdar, and S. Qadeer: 2003, 'Thread-Modular Abstraction Refinement'. In: *Proceedings of CAV (to appear)*.
37. Henzinger, T. A., R. Jhala, R. Majumdar, and G. Sutre: 2002, 'Lazy abstraction'. In: *Proceedings of POPL*. pp. 58–70.
38. Henzinger, T. A., S. Qadeer, and S. K. Rajamani: 2000, 'Decomposing Refinement Proofs using Assume-guarantee Reasoning'. In: *Proceedings of ICCAD*. pp. 245–252.
39. Hoare, C. A. R.: 1969, 'An axiomatic basis for computer programming'. *Communications of the ACM* **12**(10), 576–580.
40. Hoare, C. A. R.: 1985, *Communicating Sequential Processes*. Prentice Hall.
41. Kurshan, R. P.: 1989, 'Analysis of Discrete Event Coordination'. In: *Proceedings REX Workshop 89*, Vol. 430. pp. 414–453.
42. Kurshan, R. P.: 1994, *Computer-aided verification of coordinating processes: the automata-theoretic approach*. Princeton University Press.
43. Lakhnech, Y., S. Bensalem, S. Berezin, and S. Owre: 2001, 'Incremental Verification by Abstraction'. In: *Proceedings of TACAS*, Vol. 2031. pp. 98–112.

44. McMillan, K. L.: 1997, 'A Compositional Rule for Hardware Design Refinement'. In: *Proceedings of CAV*, Vol. 1254. pp. 24–35.
45. Milner, R.: 1989, *Communication and Concurrency.* London: Prentice-Hall International.
46. Namjoshi, K. S. and R. P. Kurshan: 2000, 'Syntactic Program Transformations for Automatic Abstraction'. In: *Proceedings of CAV*, Vol. 1855. pp. 435–449.
47. Naumovich, G., L. A. Clarke, L. J. Osterweil, and M. B. Dwyer: 1997, 'Verification of concurrent software with FLAVERS'. In: *Proceedings of ICSE*. pp. 594–595.
48. Nelson, G.: 1980, 'Techniques for Program Verification'. Ph.D. thesis, Stanford University.
49. Paige, R. and R. E. Tarjan: 1987, 'Three Partition Refinement Algorithms'. *SIAM Journal of Computing* **16**(6), 973–989.
50. Păsăreanu, C. S., M. B. Dwyer, and W. Visser: 2001, 'Finding Feasible Counter-examples when Model Checking Abstracted Java Programs'. In: *Proceedings of TACAS*, Vol. 2031. pp. 284–298.
51. Roscoe, A. W.: 1997, *The Theory and Practice of Concurrency.* London: Prentice-Hall International.
52. Stoller, S. D.: 2002, 'Model-checking multi-threaded distributed Java programs'. *International Journal on Software Tools for Technology Transfer* **4**(1), 71–91.