Lazy Abstraction*

Thomas A. Henzinger Ranjit Jhala Rupak Majumdar
EECS Department, University of California
Berkeley, CA 94720-1770, U.S.A.
{tah,jhala,rupak}@eecs.berkeley.edu

Grégoire Sutre
LaBRI, Université de Bordeaux 1
33405 Talence Cedex, France
sutre@labri.u-bordeaux.fr

ABSTRACT

One approach to model checking software is based on the *abstract-check-refine* paradigm: build an abstract model, then check the desired property, and if the check fails, refine the model and start over. We introduce the concept of *lazy abstraction* to integrate and optimize the three phases of the abstract-check-refine loop. Lazy abstraction continuously builds and refines a single abstract model on demand, driven by the model checker, so that different parts of the model may exhibit different degrees of precision, namely just enough to verify the desired property. We present an algorithm for model checking safety properties using lazy abstraction and describe an implementation of the algorithm applied to C programs. We also provide sufficient conditions for the termination of the method.

1. INTRODUCTION

While model checking [11] has made significant inroads in hardware verification, a renewed focus on model checking for software has emerged only in the past couple of years. We believe that this renewed attention has been helped significantly by two related trends: first, modern model checking is increasingly viewed more broadly than state enumeration or BDD crunching, namely, as computation with predicates that represent state sets (keywords "predicate abstraction" [20], "symbolic transition systems" [22], "constraint-based model checking" [14]); second, the performance of decision procedures and theorem provers for relevant predicate theories (e.g., booleans, Presburger, arrays) and their combinations has been steadily improving [15, 25, 27].

*This work was supported in part by the NSF ITR grant CCR-0085949, the NSF Theory grant CCR-9988172, the DARPA PCES grant F33615-00-C-1693, the MARCO GSRC grant 98-DT-660, and the SRC contract 99-TJ-683.003.

POPL '02, January 16-18, 2002, Portland, Oregon, USA.

One traditional flow for model checking a piece of code proceeds through the following loop [5, 10, 28]:

Step 1 ("abstraction") A finite set of predicates is chosen, and an abstract model of the given program is built automatically as a finite or push-down automaton whose states represent truth assignments for the chosen predicates.

Step 2 ("verification") The abstract model is checked automatically for the desired property. If the abstract model is error-free, then so is the original program (**return** "program correct"); otherwise, an abstract counterexample is produced automatically which demonstrates how the model violates the property.

Step 3 ("counterexample-driven refinement") It is checked automatically if the abstract counterexample corresponds to a concrete counterexample in the original program. If so, then a program error has been found (**return** "program incorrect"); otherwise, the chosen set of predicates does not contain enough information for proving program correctness and new predicates must be added. The selection of such predicates is automated, or at least guided, by the failure to concretize the abstract counterexample [10].

Goto Step 1.

The problem with this approach is of course that both Step 1 and Step 2 are computationally hard problems, and without additional optimizations, the method does not scale to large systems. We believe that in order to evaluate the full promise of this approach, the loop from abstraction to verification to refinement should be short-circuited. We show that all three steps can be integrated tightly through a concept we call "lazy abstraction," and that this integration can offer significant advantages in performance, by avoiding the repetition of work from one iteration of the loop to the next.

Intuitively, lazy abstraction proceeds as follows. In Step 3, call the abstract state in which the abstract counterexample fails to have a concrete counterpart, the *pivot state*. The pivot state suggests which predicates should be used to refine the abstract model. However, instead of building an entire new abstract model, we refine the current abstract model "from the pivot state on." Since the abstract model may contain loops, such *refinement on demand* may, of course, refine parts of the abstract model that have already been constructed, but it will do so only if necessary; that is, if the desired property can be verified without revisiting some parts of the abstract model, then our algorithm succeeds in doing so. The algorithm integrates all three steps by constructing and verifying and refining *on-the-fly* an abstract model of the program, until either the desired property is established or a concrete counterexample is found. Upon termination with the outcome "program correct," the proof is not an abstract model on a global set of predicates, but an abstract model whose predicates change from state to state.

Thus, lazy abstraction combats the computational difficulties of Steps 1 and 2 in the following way. Abstraction is done on-the-fly, and only up to the precision necessary to rule out spurious counterexamples. On-the-fly construction of an abstract transition system eliminates an often wasteful and expensive model-construction phase; model checking only the "current" portion of the abstract transition system saves the cost of unnecessary exploration in parts of the state space that are already known to be free of errors. It is easy to envision extreme cases in which our algorithm achieves large savings. Suppose, for example, that the program flow graph makes an initial choice between two unconnected subgraphs. Then, once one of the subgraphs has been verified with a given set of abstract predicates, it does not need to be revisited even if the other subgraph requires additional predicates. While the typical savings of lazy abstraction may be less extreme, they can be significantly more subtle, in ways that cannot be easily recreated by manual intervention.

Lazy abstraction adds demand-driven path sensitivity to traditional dataflow analysis of programs. It is *sound*: the counterexample refinement phase rules out false positives. In case an error is found, the model checker also provides a counterexample trace in the program showing how the property is violated. Automatic abstraction allows running the analysis *directly* on an implementation, rather than constructing an abstract model that may or may not be a correct abstraction of the system. By always maintaining the minimal necessary information to validate or invalidate the property, lazy abstraction scales to large systems without sacrificing precision; it eliminates unnecessary and expensive computations that take place in a naive implementation of the abstract-check-refine loop.

In Section 2, we illustrate the algorithm on a small example. While lazy abstraction is a generic technique that works on any modeling paradigm, we provide examples from the automatic verification of C programs. In Section 3, we present the formal framework of predicate abstraction using labeled transition systems and symbolic representations enriched by so-called *precision measures*, which order the precision with which an abstract state approximates a given set of concrete states. In Section 4, we present the algorithm of lazy abstraction for safety checking. The simplest and most important class of program properties, *safety* properties are correctness properties that can be specified by a set of error states, either of the program (such as deadlock states) or of a monitor automaton (which may, for example, report an error if a file is accessed without having been opened). Lazy abstraction for other properties is deferred to future work. In Section 5, we show how the general framework can be instantiated to automatically verify safety properties of C programs. In Section 6, we describe BLAST (the Berkeley Lazy Abstraction Software verification Toolkit), which implements the lazy abstraction algorithm for C programs, and we provide some initial experimental evidence that the algorithm does indeed scale to real systems code.

The final Section 7 presents two theoretical results related to lazy abstraction. The main question of interest is, of course, given a theory of predicates (such as Presburger arithmetic), a C program, and a correctness property, if there is a *finite* set of predicates that contains enough information for verifying the program (i.e., if there is a predicate abstraction that is finite-state and witnesses the correctness property). We show this question to be, not surprisingly, undecidable. It follows that lazy abstraction (or any method) must be a *semi-algorithm*, which may or may not terminate. However, we show that the lazy abstraction semi-algorithm terminates under a customary condition on the predicate theory (no infinite ascending chains of predicates) and an abstract condition on the program (finite trace equivalence), which has been established for many interesting classes of infinite-state systems (such as timed automata [2]).

2. A LOCKING EXAMPLE

```
 Example() {
 1:  if (*){
 7:     do {
             got_lock = 0;
 8:          if (*){
 9:              lock();
                 got_lock++;
             }
 10:         if (got_lock){
 11:             unlock();
             }
 12:    } while (*)
     }
 2:  do {
         lock();
         old = new;
 3:      if (*){
 4:          unlock();
             new++;
         }
 5:  } while (new != old);
 6:  unlock();
     return;
}
```

```
lock(){
   if (LOCK == 0){
      LOCK = 1;
   } else {
      ERROR
   }
}

unlock(){
   if (LOCK == 1){
      LOCK = 0;
   } else {
      ERROR
   }
}
```

Figure 1: C program

We begin by showing how lazy abstraction works on C programs. The algorithm works in two phases. The first is the forward-search phase, where we build a tree that represents a portion of the reachable, abstract state space of the program. Each edge of the tree is labeled by a program fragment, such as a sequence of assignments or an assume predicate. Each node of the tree is labeled by a finite set of predicates, which determines the precision of the abstraction, and a boolean combination of these predicates, which describes the state of the program assuming execution follows the sequence of instructions labeling the edges from the root of the tree to the node.

If we find that an error state is reachable in the tree, then we go to the second phase, which checks if the error is real or results from our abstraction being too coarse (*i.e.*, if

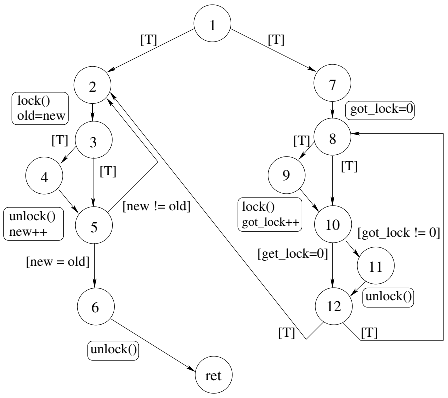


Figure 2: Control flow automaton

we lost too much information by restricting ourselves to a particular set of predicates). If the latter is true, we ask a theorem prover for additional predicates, such that by using the new predicates we can rule out that particular spurious counterexample (and maybe others as well). However, we add the new predicates only to those nodes in the search tree where they are required.

Consider the program given in Figure 1. The functions `lock` and `unlock` control some global variable called `LOCK`, which is 1 when the lock is held by the function `Example`, and 0 otherwise. We assume the precondition that when the function `Example` starts, it does not hold the lock. The property we wish to check is that (1) the function `Example` never calls `lock` when it holds the lock, and (2) it never calls `unlock` when it does not hold the lock. We have instrumented the functions `lock` and `unlock` so that checking this property amounts to checking that the $ERROR$ labels are never reached in the code.

2.1 Control flow automata for C programs

We work with the *control flow automaton* (CFA) for the various C functions of interest. This is essentially the control flow graph [1], with instructions labeling the edges rather than the vertices. Intuitively, the automaton comprises: (1) variables: local and global variables that the C function uses, (2) vertices: control locations of the C function, and (3) labeled directed edges: connecting vertices. An edge is labeled either with a *basic block* of instructions that are executed to move between the source and destination locations of the edge, or with an *assume predicate* corresponding to the branch condition that must be true for that edge to be taken.

Instead of a formal definition, we give in Figure 2 the CFA for the function `Example` shown in Figure 1. The labels in the C program correspond to the automaton vertices with the same label. The edges labeled with boxes are basic blocks; those labeled with [·] correspond to assume predicates. The `if (*)` represents branches that are taken due to predicates that are not modeled; we assume that either direction can be taken, hence both outgoing edges are labeled with `[T]`, which stands for the assume predicate *true*.

2.2 Verification

The model checking is done on the CFA shown in Figure 2. For simplicity, we assume that the call to `lock` and `unlock` are atomic operations: if `lock` is called properly (*i.e.*, with the lock not held), then it sets the value of `LOCK` to 1, otherwise it goes to $ERROR$; and similarly for `unlock`. From the specification we know it is important whether or not the lock is held, hence we start by considering the two predicates[1] $LOCK = 1$ and $LOCK = 0$. (This is not necessary: even if we start with the empty set of predicates, the algorithm discovers the above predicates via spurious counterexamples.)

Forward search

The first phase of the algorithm is the forward-search phase shown in Figure 3. The algorithm constructs in depth-first order a search tree whose nodes correspond to vertices of the CFA. The labels of the nodes are formulas, called *reachable regions*, which represent what is known about the state of the program with respect to the set of predicates being considered, after executing the instructions from the root of the tree to the given node. Each reachable region is obtained from the reachable region of the parent node in the tree and the instructions on the edge between the parent and the node, by a local computation. We require that the reachable regions be overapproximations of the set of states actually reachable by executing the path from the root of the tree. Furthermore, for each node we have a finite set of predicates we consider (in our case, $LOCK = 1$ and $LOCK = 0$), and we require that the reachable region be described as a boolean combination of these predicates.

We begin with the node that corresponds to location 1 in the CFA. The only information we have at this point is the assumption (or precondition) that the lock is not held: $LOCK = 0$. The edge from 1 to 2 is a branch that can always be taken (labeled by `[T]`), hence at 2 also we know $LOCK = 0$. To go from 2 to 3, we call `lock`, and set `old = new`. As our predicates contain no information about the variables `new` and `old`, all we can conclude is that at 4, $LOCK = 1$. Similarly going from 4 to 5 we model only what happens to `LOCK`, so due to the `unlock`, at 5 we know $LOCK = 0$. From 5 to 6 is a branch that can only be taken if `new == old`. We know nothing about `new` and `old`, hence they could be equal, and so we take the branch and again at 6, we have $LOCK = 0$, as nothing affects `LOCK` during that transition. At 6 we see that we call `unlock` with the lock not held (as $LOCK = 0$), and hence we reach an erroor node.

Backwards counterexample analysis

When we hit an error node in the search tree, we check if the path from the root to the error node is a genuine counterexample trace or results from the abstraction being too coarse. Figure 4 shows the result of this phase. In the figure, for each tree node, the formula in the curly braces, called the *bad region*, represents the set of states that can go from the corresponding control location in the CFA to an error location by executing the instructions labeling the tree edges on the path from that given node to the error node. In other words, the formula is the weakest precondition [16] of *true* with respect to the sequence of instructions labeling the path in the tree from the given node to the error node. It is easy to see that the bad region of the error node is *true*, and

[1] Predicates are written in *italics*, code in `typewriter` font.

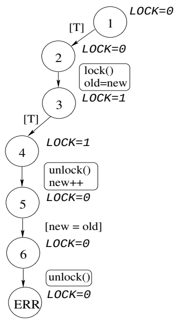


Figure 3: Forward search

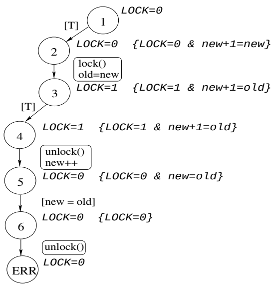


Figure 4: Backwards counterex. analysis

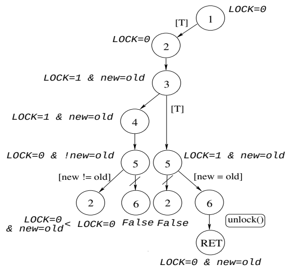


Figure 5: Search with new predicate

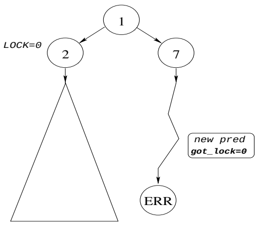


Figure 6: Searching right subtree

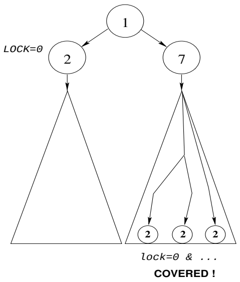


Figure 7: Leaves covered: search terminates

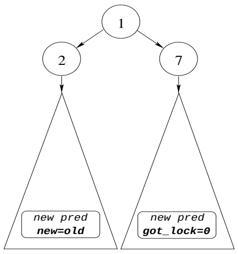


Figure 8: Different abstractions

for all other nodes it is the weakest precondition of the bad region of a child with respect to the instructions labeling the edge between the two. As we go backwards from the error node, we wish to find the first node in the tree where the intersection of the bad region (of the node) with the reachable region (from the forward-search phase) becomes empty. We may then conclude that it is *not* possible to reach the error via the given trace. That node becomes the *pivot node*, and we shall refine the abstraction from that node onwards.

Consider the bad regions labeling the nodes in Figure 4. The states that can call `unlock` from location 6 and thus end up hitting *ERROR* are exactly those where $LOCK = 0$, which therefore, is the bad region of 6. Similarly, the bad region of 5 is the weakest precondition of $LOCK = 0$ w.r.t. the assume predicate `[new=old]`, which is $LOCK = 0 \wedge new = old$. If the latter condition is not true, then the branch cannot be taken, and if the former is not true, then in the successor state does not have $LOCK = 0$. The bad region of 4 is the weakest precondition of $LOCK = 0 \wedge new = old$ w.r.t. `unlock();new++`, which is $LOCK = 1 \wedge new+1 = old$. The first conjunct comes from the fact that the lock must be held when we call `unlock`, so that thereafter the lock is not held (otherwise we would hit *ERROR*); the second from the fact that we know that after this $new = old$. The bad region of 3 is the same as that of 4, as the condition on the edge does not change the state. Finally the bad region of 2 is the weakest precondition of $LOCK = 1 \wedge new + 1 = old$ w.r.t. `lock();old=new`, which is $LOCK = 0 \wedge new + 1 = new$. It is easily seen that the first point where the intersection of the bad region with the reachable region becomes empty (*i.e.*, the conjuction of the two formulas is unsatisfiable) is at node 2. So we conclude that none of the states in the reachable region of 2, computed during the forward search, can actually hit *ERROR* following the sequence of instructions corresponding to the path we just followed backwards. In fact, the path from the node labeled 2 to the error node is the smallest infeasible suffix of the counterexample (which is the entire path from the root to the error node). Thus the node labeled 2 is the pivot node.

To do the emptiness check at each point we ask a theorem prover if the formula corresponding to the conjunction of the two regions is satisfiable. Instead of computing the classical weakest precondition, we compute weakest preconditions with explicit substitution, which gives us an equivalent formula for each of the bad regions; in particular, for 2 we get the bad region $(\exists old'.\ old' = new \wedge (\exists LOCK'.\ LOCK' = 0 \wedge (\exists new'.\ new' = new + 1 \wedge new' = old' \wedge LOCK' = 0)) \wedge LOCK = 1)$. By picking out the predicates that appear in the *proof of unsatisfiability* of this formula, we learn that the predicate $new = old$ is important (see Section 5 for details). The reason we hit an error node is that the abstraction is too coarse, and by keeping track of the new predicate, we can rule out this path. We add the new predicate $new = old$ to the subtree generated from the pivot node, and we will refine the abstraction from the pivot node onwards.

If backwards analysis had gone back all the way to the root without the theorem prover reporting unsatisfiability at any point, then we had found a real error (and counterexample).

Search with new predicates

We continue the forward-search phase, searching from the pivot node onwards. This time, we track also the predicate $new = old$, and the resulting search tree can be seen in Figure 5. Notice that we can stop the search at the leaf labeled 2, as the states satisfying the reachable region $LOCK = 0 \wedge new = old$ are a subset of those satisfying $LOCK = 0$, hence any error found from this point on would have been found by exploring from the ancestor labeled 2. We call such nodes, whose reachable regions are contained in the reachable regions of ancestor nodes in the tree, *covered.* They correspond to fixed points, and this is how loops are handled automatically. Whenever we see a covered node, we backtrack and search along some other branch in depth-first order.

Also, the reachable region of the leaf labeled 6 is empty, as that node could be reached only if at 5 `new` equalled `old`, but in node 5, we know that $new \neq old$ (as this time we are tracking the relationship of `new` and `old`). Thus we do not search from that node any more, but backtrack to the node labeled 3 and follow its other branch. Leaf 2 has the empty reachable region, as that branch is taken only when `new` is not equal to `old`. Moreover, the error node is not reachable from the sibling labeled 6, because we know from the reachable region that the lock is held when `unlock` is called. Thus we conclude that no error node is reachable in the entire left subtree.

In Figure 6 we search the right subtree of the root in a similar fashion. We discover a spurious counterexample that gives us the predicate $got\_lock = 0$. Considering this additional predicate is enough to rule out all error traces in the right subtree. Note that in the right subtree there are leaves with label 2, as control always flows to that point in the CFA. However they are all covered by the root node of the left subtree (Figure 7), as the reachable regions we compute at each of these leaf nodes is contained in the reachable region of of the root of the left subtree (at all those nodes, the lock is not held). Hence, we need not continue to search any further, and conclude that no error node is reachable.

Savings

We have achieved two savings. First, each part of the state space is refined only as much as required; in particular we have different abstractions for the two subtrees (see Figure 8). Second, we explore only the portion of the state space that is required in order to prove correctness, and do not throw away the work done earlier. For example, when we hit an error in the right subtree, we refine only that part of the search tree, keeping intact and using in the proof the left subtree, as we already know there is no error in that part of the state space. In the following sections, we make this intuitive algorithm precise.

3. ABSTRACT SYMBOLIC TRANSITION SYSTEMS

3.1 Symbolic abstraction structures

A *labeled transition system* (LTS) $\mathcal{S} = (S, \Sigma, \rightarrow)$ consists of a (possibly infinite) set S of *states*, a finite set Σ of *labels*, and a *labeled transition relation* $\rightarrow \subseteq S \times \Sigma \times S$. A transition $(s, l, s') \in \rightarrow$ is written $s \xrightarrow{l} s'$. The relation $\rightarrow$ is extended to Σ^* as follows: $s \xrightarrow{\epsilon} s'$ iff $s = s'$, and $s \xrightarrow{l\sigma} s'$ iff there exists a state s'' such that $s \xrightarrow{l} s''$ and $s'' \xrightarrow{\sigma} s'$. We write $s \xrightarrow{*} s'$ if there exists a finite sequence $\sigma \in \Sigma^*$ of labels such that $s \xrightarrow{\sigma} s'$. For a set $S_0 \subseteq S$ of states, we define the reachable set as $Reach_{\mathcal{S}}(S_0) = \{s \in S \mid \exists s_0 \in S_0.\, s_0 \xrightarrow{*} s\}$.

Symbolic algorithms on labeled transition systems manipulate regions, where each region represents a set of states. Following the framework of symbolic transition systems [18, 22], we define a (*symbolic*) *region structure* $(R, \bot, \sqcup, \sqcap, pre, post, [\![\cdot]\!])$ for the labeled transition system $\mathcal{S}$ to consist of a set R of *regions*, an element $\bot$ of R, two total functions $\sqcup, \sqcap$: $R \times R \rightarrow R$, two total functions $pre, post$: $R \times \Sigma \rightarrow R$, and a total *extension* function $[\![\cdot]\!]$: $R \rightarrow 2^S$, such that for all regions $r, r' \in R$ and every label $l \in \Sigma$, we have:

$$[\![\bot]\!] = \emptyset \tag{1}$$

$$[\![r \sqcup r']\!] = [\![r]\!] \cup [\![r']\!] \tag{2}$$

$$[\![r \sqcap r']\!] = [\![r]\!] \cap [\![r']\!] \tag{3}$$

$$[\![pre(r, l)]\!] = \{s' \in S \mid \exists\, s \in X.\, s' \xrightarrow{l} s\} \tag{4}$$

$$[\![post(r, l)]\!] = \{s' \in S \mid \exists\, s \in X.\, s \xrightarrow{l} s'\} \tag{5}$$

The intention is that the region r represents the set $[\![r]\!]$ of states. A region structure carries with it a natural preorder (i.e., a reflexive and transitive relation) $\sqsubseteq$ defined by $r \sqsubseteq r'$ if $[\![r]\!] \subseteq [\![r']\!]$, and a natural equivalence $\equiv$ defined by $r \equiv r'$ iff $[\![r]\!] = [\![r']\!]$. The region structure is *computable* if the functions $\sqcup$, $\sqcap$, pre, $post$, and $\sqsubseteq$ are computable.

EXAMPLE 1: Region structures for models of computation such as counter automata (resp. FIFO automata, timed automata) can be designed based on Presburger formulas [8] (resp. various classes of regular expressions [7, 9, 18], clock zones [2]). □

We do not require the preorder $\sqsubseteq$ to be a partial order (i.e., to be antisymmetric). Indeed, in predicate abstraction, we will design region structures with many distinct equivalent regions. Similarly, we do not require that the functions $\sqcup$ and $\sqcap$ be associative or commutative. This more general setting allows us to accommodate abstraction within the framework of region structures. However, for any finite set X of states, the operations $\sqcup X$ and $\sqcap X$ under any order of evaluation produce regions that are equivalent with respect to $\equiv$, and so there is no ambiguity.

We now introduce overapproximate versions $\widehat{post}$ and $\widehat{pre}$ of the exact successor and predecessor operators $post$ and pre on regions. A (*symbolic*) *abstraction structure* $\mathcal{A} = (\mathcal{R}, \widehat{post}, \widehat{pre}, \trianglelefteq)$ for a labeled transition system $\mathcal{S} = (S, \Sigma, \rightarrow)$ consists of a computable region structure $\mathcal{R} = (R, \bot, \sqcup, \sqcap, pre, post, [\![\cdot]\!])$ for $\mathcal{S}$ together with a *precision* preorder $\trianglelefteq \subseteq R \times R$, and two computable total functions $\widehat{post}, \widehat{pre}$: $R \times \Sigma \rightarrow R$ such that for all regions $r \in R$ and every label $l \in \Sigma$,

$$post(r, l) \sqsubseteq \widehat{post}(r, l), \tag{6}$$

$$pre(r, l) \sqsubseteq \widehat{pre}(r, l), \tag{7}$$

and $\widehat{post}$ and $\widehat{pre}$ are monotonic with respect to the preorder $(\trianglelefteq \cap \sqsubseteq)$; that is, if $r \trianglelefteq r'$ and $r \sqsubseteq r'$, then both $\widehat{post}(r, l) \trianglelefteq$

$\widehat{post}(r,l)$ and $\widehat{post}(r,l) \sqsubseteq \widehat{post}(r,l)$, and analogously for $\widehat{pre}$.

The novelty of this definition lies in the fact that regions carry precision information, which indicates how close the overapproximate operators $\widehat{post}$ and $\widehat{pre}$ are to the exact operators $post$ and pre. In particular, for two $\equiv$-equivalent regions r and r', if $r \trianglelefteq r'$, then $post(r,l) \equiv post(r',l) \sqsubseteq \widehat{post}(r,l) \sqsubseteq \widehat{post}(r',l)$; that is, $\widehat{post}(r,l)$ is a more precise overapproximation of the successor set than $\widehat{post}(r',l)$. The precision preorder permits us to perform both the concrete operations of a labeled transition system $\mathcal{S}$ and abstract interpretation of $\mathcal{S}$ within a single region structure. A region r is no longer interpreted as simply a description of the *concrete* state set $[\![r]\!]$ of $\mathcal{S}$, but as a description of an *abstract* state set, for some abstract state space, whose concretization is $[\![r]\!]$. If $r \trianglelefteq r'$, then the abstract state space in which r is interpreted is more precise than the abstract state space of r'.

The functions pre, $post$, $\widehat{pre}$, and $\widehat{post}$ are extended to $R \times \Sigma^*$ in the obvious way.

3.2 Predicate abstraction

Consider a labeled transition system $\mathcal{S} = (S, \Sigma, \rightarrow)$. A *predicate language* $\mathcal{L}$ for $\mathcal{S}$ is a set of *predicates* that are interpreted over the states in S (i.e., each predicate $p \in \mathcal{L}$ denotes a set $[\![p]\!] \subseteq S$ of states), such that the following two conditions are satisfied. (1) The boolean closure of $\mathcal{L}$ is a decidable theory (i.e., satisfiability is decidable). (2) The boolean closure of $\mathcal{L}$ is effectively closed under exact successor and predecessor operations in $\mathcal{S}$; that is, for every formula ψ in the boolean closure of $\mathcal{L}$ and every label $l \in \Sigma$, we can compute two boolean combinations φ_l^{pre} and φ_l^{post} of predicates from $\mathcal{L}$ such that

$$[\![\varphi_l^{pre}]\!] = \{s' \in S \mid \exists\, s \in [\![\psi]\!].\, s' \xrightarrow{l} s\}, \tag{8}$$

$$[\![\varphi_l^{post}]\!] = \{s' \in S \mid \exists\, s \in [\![\psi]\!].\, s \xrightarrow{l} s'\}. \tag{9}$$

Following the predicate-abstraction framework [20], we define an abstraction structure $\mathcal{A}_{\mathcal{L}} = (\mathcal{R}, \widehat{post}, \widehat{pre}, \trianglelefteq)$ for $\mathcal{S}$ and $\mathcal{L}$ as follows:

- Let $\mathcal{R} = (R, \bot, \sqcup, \sqcap, pre, post, [\![\cdot]\!])$. The regions in R are the pairs (φ, Γ), where $\Gamma \subseteq \mathcal{L}$ is a finite set of *support predicates*, and φ is a boolean formula over the predicates in Γ. The region $(false, \emptyset)$ represents $\bot$. The operators $\sqcup$ and $\sqcap$ are defined by $(\varphi,\Gamma)\sqcup(\varphi',\Gamma') = (\varphi\vee\varphi', \Gamma\cup\Gamma')$ and $(\varphi,\Gamma)\sqcap(\varphi',\Gamma') = (\varphi\wedge\varphi', \Gamma\cup\Gamma')$. Let $pre((\varphi,\Gamma),l) = (\varphi_l^{pre}, \Gamma')$, where Γ' is the least superset of Γ that contains all predicates in φ_l^{pre}. The region $post(\varphi, \Gamma)$ is defined similarly. Finally, $[\![(\varphi,\Gamma)]\!] = [\![\varphi]\!]$ straightforwardly interprets the boolean formula φ as a subset of S; that is, $[\![(\varphi,\Gamma)]\!]$ consists of all states $s \in S$ that satisfy the constraint φ.

- For a region (φ, Γ) and a label $l \in \Sigma$, we construct the overapproximation $\widehat{post}((\varphi,\Gamma),l)$ of $post((\varphi,\Gamma),l)$ as a boolean combination of the predicates in Γ. We fix a canonical sum-of-product form for formulas. We ask, for each disjunct ψ of the canonical sum-of-product form of φ, and each support predicate $p \in \Gamma$, if ψ implies $pre(p,l)$, and if ψ implies $pre(\neg p, l)$. This gives, for each disjunct ψ of φ, a conjunction ψ' of support predicates, where the predicate p occurs positively if $\psi \Rightarrow pre(p,l)$ is valid, occurs negatively if $\psi \Rightarrow pre(\neg p, l)$ is valid, and does not occur if neither validity holds. Let φ' denote the disjunction of all conjunctions ψ' so constructed. The region $\widehat{post}((\varphi,\Gamma),l)$ is the formula φ' together with the same set of support predicates, i.e., $\widehat{post}((\varphi,\Gamma),l) = (\varphi', \Gamma)$. The abstract predecessor $\widehat{pre}$ is computed similarly, using $post$ in place of pre.

- The precision preorder is the containment relation on support predicates: $(\varphi,\Gamma) \trianglelefteq (\varphi',\Gamma')$ iff $\Gamma \supseteq \Gamma'$.

Since $\mathcal{L}$ is a predicate language for $\mathcal{S}$, the preorder $\sqsubseteq$ on regions is computable. Moreover, from the definitions of $\widehat{post}$ and $\widehat{pre}$ it is clear that they can be computed effectively. Also, they are overapproximations of $post$ and pre, respectively, and monotonic with respect to the preorder $(\trianglelefteq \cap \sqsubseteq)$.

Intuitively, the support predicates determine the current abstract state space, and the formula over the support predicates represents an abstract set of states. The support predicates indicate which predicates are important, i.e., which predicates can be tracked by the abstract operations $\widehat{post}$ and $\widehat{pre}$. Note that the precision preorder $\trianglelefteq$ and properties of $\widehat{post}$ and $\widehat{pre}$ do not require all support predicates to be tracked. In particular, $\widehat{post}(r,l)$ is not the smallest set containing $post(r,l)$ expressible in terms of the support predicates —indeed, we lose information in constructing a *Cartesian abstraction* [4, 20]. Alternatively, we could construct the most precise overapproximation of $post$ using a recursive subdivision of the state space [12], splitting on the support predicates.

While the abstract operations $\widehat{post}$ and $\widehat{pre}$ do not change the support predicates, the concrete operations pre and $post$ may require the introduction of additional support predicates. These operations will be used to refine the current abstraction of the state space.

4. SYMBOLIC REACHABILITY WITH REFINEMENT

We present two algorithms for computing overapproximations of the reachable state space of a labeled transition system. The first algorithm is nondeterministic. The second algorithm, which is the lazy abstraction algorithm, resolves the nondeterminism in the first one to ensure that the overapproximation does not contain any error states, if in fact no error state is reachable in the system. These algorithms do not terminate in general (so we should call them "semi-algorithms," but we keep the term algorithm for simplicity). Sufficient conditions ensuring termination will be discussed in Section 7.1.

4.1 Reachability with refinement

The first algorithm we present is the SymbReachRefine algorithm (Algorithm 1). This is a standard symbolic forward-search algorithm, but with two extra features: (1) the ability to refine some path in the tree under construction (in order to compute a more precise abstraction of some part of the system), and (2) the ability to drop a subtree of the tree under construction (in order to recompute the subtree with more precision). Specifically, at each iteration of the algorithm, a nondeterministic choice is made: either some path in the tree is refined (lines 4–10), or some subtree is removed (lines 12–17), or the forward search goes on (lines 19–28). The `choose()` function used at lines 3 and 11 models the nondeterministic choice. The algorithm uses the abstract, overapproximate forward operator $\widehat{post}$; one can dually define a backward-search algorithm based on $\widehat{pre}$.

We use the following notation. By *reachability tree*, or simply *tree*, we mean a rooted directed tree with labels on both nodes and arcs. Each node is labeled by a region, the *reachable region*, which represents an overapproximation of the states that are reachable along the path from the root to the node. We write $\mathtt{n} : r$ for node $\mathtt{n}$ with reachable region r. Each arc is labeled by a label of the labeled transition system being analyzed. For an arc $\mathtt{n} : r \xrightarrow{l} \mathtt{n}' : r'$, we say that node $\mathtt{n}'$ is an *l-son* of node $\mathtt{n}$. A leaf is a node without l-sons, for any label l.

A classical symbolic forward-search algorithm computes a reachability tree starting from an initial region r_0. The algorithm maintains a worklist of nodes to be explored, and iteratively processes nodes until the worklist becomes empty. To process a node from the worklist, we remove it from the list and check whether the node's reachable region is covered by the already explored state space. If it is not covered, then for each label l, the l-sons are constructed using the $\widehat{post}$ operation and added to the worklist. If the node is covered, then the search is resumed by processing another node from the worklist; in this case, we need not compute the successors of the node, because they are processed elsewhere. When the algorithm stops, it has computed a region —the union of reachable regions for all nodes— which is an overapproximation of the states reachable from the initial region r_0.

Due to the extra features of our algorithm, we attach a marking to each node. At any time, a node of the tree computed by the SymbReachRefine algorithm is either

- *unmarked*, meaning that it has been discovered but not processed, or
- *marked and covered*, meaning that it has been processed and found to be covered (i.e., it did not add any new states to the union of reachable regions at the time of marking), so there is no need to compute its successors, or
- *marked and uncovered*, meaning that it has been processed and its reachable region is not contained in the union of reachable regions for the other nodes, so its successors must be computed.

Nodes may change type during the construction of the tree: unmarked nodes may become marked, and marked nodes may become unmarked. Every marked node carries a *time stamp* indicating the time when the node was marked last; these time stamps linearly order all marked nodes in the tree.

The goal of this generic symbolic reachability algorithm is to point out sufficient conditions for correctness. Indeed, once this algorithm is proved correct, any more detailed implementation (e.g., replacing the nondeterministic choose() function by some deterministic computation of the path to refine) must necessarily be correct. The correctness of the SymbReachRefine algorithm is expressed by the following theorem. Note that the correctness does not depend on the order in which the state space is explored (e.g., depth-first or breadth-first).

Algorithm 1 SymbReachRefine($\mathcal{A}, r_0$)

Require: a region structure $\mathcal{R} = (R, \bot, \sqcup, \sqcap, pre, post, [\![\cdot]\!])$, an abstraction structure $\mathcal{A} = (\mathcal{R}, \widehat{post}, \widehat{pre}, \trianglelefteq)$, and an initial region $r_0 \in R$.

1: create root $\mathtt{r}$ labeled with r_0
2: **while** there are unmarked nodes **do**
3: **if** choose() **then**
4: {some path is refined}
5: pick a path $\mathtt{n}{:}r \xrightarrow{\sigma} \mathtt{n}'{:}r'$ with $\sigma \in \Sigma^*$
6: let w be any region such that $w \trianglelefteq r$ and $w \equiv r$
7: **for each** node $\mathtt{m}{:}u$ along the path $\mathtt{n}{:}r \xrightarrow{\sigma} \mathtt{n}'{:}r'$ **do**
8: relabel $\mathtt{m}$ by $\widehat{post}(w, \sigma')$ where σ' is the prefix of σ such that $\mathtt{n}{:}r \xrightarrow{\sigma'} \mathtt{m}{:}u$
9: **for each** covered marked leaf $\mathtt{m}$ that was marked *after* $\mathtt{n}$ was marked **do**
10: unmark $\mathtt{m}$ {to guarantee correctness}
11: **else if** choose() **then**
12: {some subtree is removed}
13: pick a marked node $\mathtt{n}$
14: remove the subtrees starting at the sons of $\mathtt{n}$
15: **for each** covered marked leaf $\mathtt{m}$ that was marked *after* $\mathtt{n}$ was marked **do**
16: unmark $\mathtt{m}$ {to guarantee correctness}
17: unmark $\mathtt{n}$
18: **else**
19: {the reachability search goes on}
20: pick an unmarked node $\mathtt{n}{:}r$
21: **if** $r \sqsubseteq \bigsqcup\{u \mid \mathtt{m}{:}u$ is an uncovered marked node$\}$ **then**
22: mark $\mathtt{n}$ as *covered* $\{Reach_{\mathcal{S}}([\![r]\!])$ is processed elsewhere$\}$
23: **else**
24: **for each** label $l \in \Sigma$ **do**
25: $r' \leftarrow \widehat{post}(r, l)$
26: **if** $r' \not\sqsubseteq \bot$ **then**
27: construct a son $\mathtt{n}'{:}r'$ of $\mathtt{n}$ and label the arc $\mathtt{n} \xrightarrow{l} \mathtt{n}'$ $\{\mathtt{n}'$ is an l-son of $\mathtt{n}\}$
28: mark $\mathtt{n}$ as *uncovered*
29: **return** the region $\bigsqcup\{u \mid \mathtt{m}{:}u$ is an uncovered marked node$\}$

THEOREM 1. **[Correctness]** *Let $\mathcal{A}$ be an abstraction structure for a labeled transition system $\mathcal{S}$. For every initial region r_0, and every terminating execution of the algorithm* SymbReachRefine($\mathcal{A}, r_0$)*, we have*

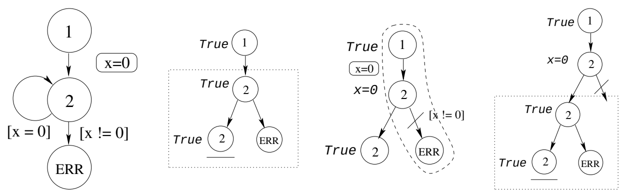
$$Reach_{\mathcal{S}}([\![r_0]\!]) \subseteq [\![\mathsf{SymbReachRefine}(\mathcal{A}, r_0)]\!].$$

4.2 Counterexample-driven refinement

In an actual implementation, the nondeterministic choice function for refinement is replaced by a function that depends on abstract counterexample traces. As long as the reachable regions have an empty intersection with a specified error region, the symbolic reachability algorithm with error-driven refinement behaves like a usual symbolic forward-search algorithm (i.e., the algorithm obtained by removing lines 3–18 from Algorithm 1). However, when we discover a node whose reachable region contains an error state, and the path of the (abstract) reachability tree from the root to the node is not a feasible path in the (concrete) labeled transition system, then we refine that path. This is made precise in the algorithm LazyAbstraction (Algorithm 2).

When the algorithm processes a node $\mathtt{n}$ whose reachable region has a nonempty intersection with the error region (line 4), it checks if this is an actual error. For each node along the path from the root to $\mathtt{n}$, let the *bad path* of the node be the path in the tree from the node to $\mathtt{n}$, and let the node's *bad region* be the predecessor of the error region with respect to the sequence of operations labeling its bad path. The algorithm checks for every ancestor node of $\mathtt{n}$ if the reachable region of the ancestor has a nonempty intersection with the ancestor's bad region. This happens in line 5, where we find the oldest ancestor with a nonempty intersection. If the oldest ancestor is the root of the tree, then we have found a genuine error trace and the algorithm stops (lines 6, 7). If not, then the error trace is spurious. The parent (node $\mathtt{n}''$ in line 9) of the oldest ancestor with a nonempty intersection is the first node (going up the tree) with an empty intersection, and is the *pivot node*. It is not possible to reach the error node via the path in the tree from the root to `n`, because the reachable region of the pivot node is an overapproximation of the set of states that the system can be in following the labels from the root to the pivot, and as the intersection of the pivot's reachable region with its bad region is empty, none of the states that are actually reached at the pivot can lead via the bad path into the error region. Hence, the abstraction must be refined to eliminate this trace (lines 8–18).

The algorithm refines the search from the pivot node on. If it is worth keeping the whole subtree of the pivot node (which is determined by the `keep_subtree` heuristic), then it refines the path from the pivot node to the currently processed node `n` (lines 13, 14); otherwise it removes the entire subtree rooted at the pivot node and unmarks the pivot node (line 16). Any node that was marked covered after the pivot node is now uncovered, and the search continues (line 17). The `keep_subtree` function determines whether the subtree rooted at the pivot node is discarded in the refinement process. It does not affect correctness, but may affect termination. For efficiency, we want to keep as much computation as possible (to avoid repeating the same work), and hence we would like to keep subtrees. But there may be coarse nodes in the subtrees that can cause the algorithm to go into infinite "refinement loops." The use of a `keep_subtree` function allows us to experiment with different strategies.



(a) (b) (c) (d)

Figure 9: Refinement loops

EXAMPLE 2: Consider the CFA given in Figure 9(a). The result of the first forward-search phase is given in (b). The second node labeled 2 is covered by its parent. Along the other branch the search hits an error node. Figure (c) shows how the error path is refined, by adding the predicate $x = 0$: at the parent 2 node, the reachable region is now $x = 0$, so the branch to the error node is ruled out. The child 2 node is now no longer covered, so it is unmarked and the search resumes from that node. In (d) we see that the search results in exactly the same subtree we had in (b), hence, the refine-search process will repeat forever. If, instead, we delete the entire subtree below the pivot node (i.e., the node labeled 1), unmark the pivot node, and start over searching with the new predicate $x = 0$ from the pivot node onwards, then the algorithm terminates. □

Note that we explore the state space forward, and analyse counterexamples backward. Hence we use the abstract, overapproximate $\widehat{post}$ operator and the concrete, exact *pre* operator. One can dually define an algorithm that explores backward (using $\widehat{pre}$), and analyses counterexamples forward (using *post*).

The refinement step uses a *focus operator*, denoted Φ. Intuitively, a focus operator is needed when we have a region r that cannot reach an error region $\mathcal{E}$ in the labeled transition system along some path labeled σ, but r is not precise enough, and we hit the error region in the abstract forward search. We want $\Phi(r, \sigma, \mathcal{E})$ to return a region equivalent to r, but precise enough to rule out the the spurious error trace σ. Formally, a focus operator Φ for an abstraction structure $\mathcal{A}$ with region set R is a function Φ: $R \times \Sigma^* \times R \to R$ such that for all regions $r, \mathcal{E} \in R$ and all $\sigma \in \Sigma^*$,

– $\Phi(r, \sigma, \mathcal{E}) \equiv r$ and $\Phi(r, \sigma, \mathcal{E}) \trianglelefteq r$, and

– if $pre(\mathcal{E}, \sigma) \sqcap r \equiv \bot$, then $\widehat{post}(\Phi(r, \sigma, \mathcal{E}), \sigma) \sqcap \mathcal{E} \equiv \bot$.

The second condition is not necessary for correctness of the algorithm LazyAbstraction, but it will allow us to obtain a termination criterion in Section 7.1. In predicate abstraction, a focus operator typically adds support predicates in order to avoid spurious error traces, but leaves the boolean formula characterizing the reachable region unchanged. As the support predicates are added locally, as a result, at any time there may be regions in the reachability tree with different support predicates. In this way support predicates can be used to locally refine the state space during search.

The correctness of the LazyAbstraction algorithm, which again does not depend on the order in which the state space is explored, is expressed by the following theorem.

THEOREM 2. **[Correctness]** *Let $\mathcal{A}$ be an abstraction structure for a labeled transition system $\mathcal{S}$, and let Φ be a focus operator for $\mathcal{A}$. For every initial region r_0, error region $\mathcal{E}$, and every terminating execution of* LazyAbstraction$(\mathcal{A}, \Phi, r_0, \mathcal{E})$*, we have:*

(i) If LazyAbstraction$(\mathcal{A}, \Phi, r_0, \mathcal{E})$ *returns an error trace σ, then there exist two states $s \in [\![r_0]\!]$ and $s' \in [\![\mathcal{E}]\!]$ such that $s \xrightarrow{\sigma} s'$.*

(ii) Otherwise, LazyAbstraction$(\mathcal{A}, \Phi, r_0, \mathcal{E})$ *returns a region r that satisfies both $Reach_{\mathcal{S}}([\![r_0]\!]) \subseteq [\![r]\!]$ and $[\![r]\!] \cap [\![\mathcal{E}]\!] = \emptyset$.*

We mention two optimizations of Algorithm 2. First, if nodes are processed in depth-first search order (i.e., we always pick an unmarked node among those of higher depth on line 3), then for every ancestor `n` of the latest marked node, the marked nodes in the subtree of `n` are precisely the nodes that were processed *after* `n` was marked. Hence the **for**-loop of lines 17–18 can be limited to the leaves in the subtree of `n`. Second, in order to implement efficiently the covering test of line 19, we use a variable c to collect the union of reachable regions of uncovered marked nodes as we go along. We can then replace the covering test of line 19 by the test $r \sqsubseteq c$, and the return statement of line 27 by **return** c. To update c, we add the statement $c \leftarrow c \sqcup r$ after line 26, and we recompute c after line 18.

Algorithm 2 $\mathsf{LazyAbstraction}(\mathcal{A}, \Phi, r_0, \mathcal{E})$

Require: a region structure $\mathcal{R} = (R, \perp, \sqcup, \sqcap, pre, post, [\![\cdot]\!])$, an abstraction structure $\mathcal{A} = (\mathcal{R}, \widehat{post}, \widehat{pre}, \trianglelefteq)$, a focus operator Φ for $\mathcal{A}$, an initial region $r_0 \in R$, and an error region $\mathcal{E} \in R$.

create root $\mathtt{r}$ labeled with r_0
while there are unmarked nodes **do**
  pick an unmarked node $\mathtt{n}{:}r$
  if $r \sqcap \mathcal{E} \not\sqsubseteq \perp$ **then**
    let $\mathtt{n}' : r' \xrightarrow{\sigma} \mathtt{n} : r$ be the oldest ancestor of $\mathtt{n}$ with $pre(\mathcal{E}, \sigma) \sqcap r' \not\equiv \perp$
    if $\mathtt{n}'$ is the root **then**
      return the "error trace" σ
    else
      let $\mathtt{n}''{:}r'' \xrightarrow{l} \mathtt{n}'{:}r'$ be the father of $\mathtt{n}'$ $\{\mathtt{n}''{:}r''$ satisfies $pre(\mathcal{E}, l\sigma) \sqcap r'' \equiv \perp\}$
      let τ denote the time stamp of $\mathtt{n}''$
      relabel $\mathtt{n}''$ by $w'' = \Phi(r'', l\sigma, \mathcal{E})$
      if `keep_subtree`$(\mathtt{n}'')$ **then**
        for each node $\mathtt{m}{:}u$ along the path $\mathtt{n}'{:}r' \xrightarrow{\sigma} \mathtt{n}{:}r$ **do**
          re-label $\mathtt{m}$ by $\widehat{post}(w'', l\sigma')$ where σ' is the prefix of σ such that $\mathtt{n}'{:}r' \xrightarrow{\sigma'} \mathtt{m}{:}u$
      else
        remove the subtrees starting at the sons of $\mathtt{n}''$ and unmark $\mathtt{n}''$
      for each covered marked leaf $\mathtt{m}$ that was marked *after* τ **do**
        unmark $\mathtt{m}$ {to guarantee correctness}
  else if $r \sqsubseteq \bigsqcup\{u \mid \mathtt{m}:u$ is an uncovered marked node$\}$ **then**
    mark $\mathtt{n}$ as *covered* $\{Reach_{\mathcal{S}}([\![r]\!])$ is processed elsewhere$\}$
  else
    for each label $l \in \Sigma$ **do**
      $r' \leftarrow \widehat{post}(r, l)$
      if $r' \not\sqsubseteq \perp$ **then**
        construct a son $\mathtt{n}'{:}r'$ of $\mathtt{n}$ and label the arc $\mathtt{n} \xrightarrow{l} \mathtt{n}'$ $\{\mathtt{n}'$ is an l-son of $\mathtt{n}\}$
    mark $\mathtt{n}$ as *uncovered*
return the region $\bigsqcup\{u \mid \mathtt{m}{:}u$ is an uncovered marked node$\}$

5. LAZY ABSTRACTION FOR C

We now consider how the algorithm of the previous section can be applied to the verification of C programs. In order to do this, we must define (1) a labeled transition system for a given C program, (2) a region structure and an abstraction structure for the labeled transition system, and (3) a focus operator.

5.1 From C to labeled transition systems

In the sequel, we assume for convenience and without loss of generality that the expressions of C programs (and hence the edge labels of CFAs) satisfy the following conditions: (1) all expressions are free of side-effects and of short-circuit evaluation, and do not contain multiple dereferences of a pointer (e.g., `**p`); (2) a function call occurs only at the top-most level of an expression (for example, "`z=x+f(y);`" is replaced by "`t=f(y); z=x+t;`").

Consider a C program that consists of a set F of functions over a set X of variables. For each function $f \in F$, let (V_f, E_f) be the vertices and edges of the CFA for f (recall Section 2.1), and let $V = \bigcup_{f \in F} V_f$ and $E = \bigcup_{f \in F} E_f$. We define a labeled transition system $(S, \Sigma, \rightarrow)$ as follows. The states in S are the triples (pc, v, cs), where $pc \in V$ is the "program counter," v is a valuation of the set X of program variables, and $cs \in V^*$ is the function call stack. The set Σ of labels is E. Then, a transition $(pc, v, cs) \xrightarrow{e} (pc', v', cs')$ is one of the following:

- If e is an assume predicate, then e is true at the valuation v, and pc' is the e-successor of pc and $v' = v$ and $cs' = cs$.
- If e is a basic block corresponding to a function call, then pc' is the start location of the function being called, $v' = v$, and $cs' = pc.cs$.
- If e is a basic block corresponding to a function return, and $cs = pc''.cs''$, then pc' is the (unique) successor of pc'' and $v' = v$ and $cs' = cs''$.
- For all other basic blocks e, we have that pc' is the e-successor of pc, and v' results from v by executing e, and $cs' = cs$.

5.2 A symbolic abstraction structure for C

We use the predicate-abstraction framework described in Section 3.2. Our predicate language $\mathcal{L}$ contains the quantifier-free formulas of the theory of equality with uninterpreted functions and of the theory of integers with addition. The combination of these theories (together with others such as the theory of arrays) is decidable, and efficient decision procedures are available [26, 6]. This choice of predicate language means that we can analyze exactly only C programs whose basic data types are integers and pointers, with the operations integer addition, arithmetic comparison, and pointer equality. In the sequel, we assume that the given C program has been modified so that any C data type or operation that we cannot model in our predicate language (e.g., integer multiplication) has been replaced by an uninterpreted function. This modification is conservative: a path in the original program is still a path in the modified program (but there may be more paths in the modified program, for instance, paths that depend on a particular property of the multiplication operator). For the device driver code we analyzed (see Section 6.2), the properties of interest can all be proved in this way. For other programs or properties, of course, one may have to use richer predicate languages.

We define the set D of *data regions* to be the set of pairs (φ, Γ), where φ is a boolean formula over predicates from Γ, and Γ is a finite subset of $\mathcal{L}$. Data regions model the data component (*i.e.*, the valuations of the variables) of states; the other components (*i.e.*, the program conter and the function call stack) are modeled explicitly. Specifically, an *atomic region* is a triple consisting of a control location, a data region, and a call stack. For every atomic region $(pc, (\varphi, \Gamma), cs)$, we have $[\![(pc, (\varphi, \Gamma), cs)]\!] = \{(pc, v, cs) \mid v$ satisfies $\varphi\}$. We write $A = V \times D \times V^*$ for the set of atomic regions. A *region* is a finite set of atomic regions.

5.2.1 *Concrete pre and post*

For algorithm $\mathsf{LazyAbstraction}$ we need only *pre*, hence we omit a detailed discussion of the computation of *post*. First we define pre_D: $D \times \Sigma \rightarrow D$ on data regions. This operator can be extended in a straightforward manner to obtain an operator pre_A: $A \times \Sigma \rightarrow A$ on atomic regions. Finally, given a region $r \subseteq A$ and a label $l \in \Sigma$, define $pre(r, l) = \{pre_A(a, l) \mid a \in r\}$.

We now describe the computation of pre_D. For a statement s and a formula φ, let $wp(\varphi, s)$ denote the *weakest liberal precondition* [16, 21] of φ with respect to s; that is,

$wp(\varphi, s)$ is the weakest formula whose truth before s entails the truth of φ after s terminates, if it terminates. For example, for the assignment "`x = e`," where x is a scalar variable and e is an expression of the appropriate type, we have $wp(\varphi, \texttt{x = e}) = \varphi[e/x]$, where $\varphi[e/x]$ denotes φ with all occurrences of x replaced by e. In the presence of pointers and aliasing, of course, a syntactic substitution is no longer accurate: for example, $wp(*p = 1, \texttt{x = 0})$ is not $*p = 1$, because if `x` and `*p` are aliased, then $*p = 1$ does not hold after the assignment. Thus, we use Morris' general axiom of assignment [24]. An *address* is either a variable, a structure-field access from a address, or a dereference of a address. Consider the computation of $wp(\varphi, \texttt{x=e})$, where x is an address, and let y be an address mentioned in the formula φ. Then there are two cases to consider: either x and y are aliases, and hence the assignment of e to x causes the value of y to become e; or they are not aliases, and the assignment to x leaves y unchanged. Define

$$\varphi[x, e, y] = (\ \&x = \&y\ \wedge\ \varphi[e/y]) \vee (\ \&x \neq \&y\ \wedge\ \varphi).$$

In general, let $y_1, y_2, \ldots, y_n$ be the addresses mentioned in φ. Then $wp(\varphi, \texttt{x=e})$ is $\varphi[x, e, y_1][x, e, y_2] \ldots [x, e, y_n]$. In the example above, we have $wp(*p = 1, \texttt{x = 0}) = (\ \&x = p \wedge 0 = 1) \vee (\ \&x \neq p \wedge *p = 1)$. If k addresses occur in the formula φ, then the weakest precondition has 2^k syntactic disjuncts, each considering a possible alias scenario of the k addresses with x. However, one can use a pointer analysis to improve the precision of the weakest-precondition computation [3]: if the pointer analysis says that x and y cannot be aliased at the program point before $\texttt{x} = \texttt{e}$, then we can prune the disjuncts that represent a scenario where x is aliased to y, and we can partially evaluate the disjuncts that represent a scenario where x is not aliased to y.

For an assume predicate, the weakest precondition $wp(\varphi, \texttt{assume p})$ is $p \wedge \varphi$. Finally, the weakest precondition of a sequence of statements is the composition of the weakest preconditions: $wp(\varphi, s_1; s_2) = wp(wp(\varphi, s_2), s_1)$. From this, we get the weakest precondition of a basic block (*i.e.*, a finite sequence of statements) by induction. Finally, from the wp operator we construct the operator pre_D on data regions by adding all predicates that occur in the weakest precondition as support predicates.

EXAMPLE 3: We use the *pre* operator for backwards counterexample analysis. Consider the nodes labeled 4 and 5 in Figure 4. The bad region (in the curly braces) for node 4 is the *pre* of the bad region of 6 with respect to `unlock(); new++` labeling the edge between the nodes 4 and 5 in the CFA. The wp operator returns:

$$wp(LOCK = 0 \wedge new = old, \texttt{unlock();new++}) = 0 = 0 \wedge new + 1 = old$$

(by inlining the code for `unlock`, *i.e.*, treating it as simply $LOCK = \texttt{0}$), which is the bad region of node 4 (ignoring the support predicates). □

5.2.2 Abstract pre and post

Given the concrete pre_D operations, we construct the abstract $\widehat{post}_D$ operation for data regions by Cartesian abstraction, as in Section 3.2. We extend this to $\widehat{post}$ on regions as in the concrete case. Note that by considering the statements of a basic block together, we gain precision in the analysis: the abstract $\widehat{post}$ approximates the effect of an entire basic block rather than abstracting (and losing information) the effect of each statement within a basic block. The abstract $\widehat{pre}$ operator can be computed dually, but is not needed for algorithm LazyAbstraction.

EXAMPLE 4: Recall the parent-child pair of nodes with labels 4 and 5 in the search tree of Figure 5. At 4 we have the reachable region $LOCK = 1 \wedge new = old$. The label of the node 5 is $\widehat{post}(LOCK = 1 \wedge new = old, \texttt{unlock();new++})$, with respect to the three support predicates $LOCK = 0$, $LOCK = 1$, and $new = old$. The weakest preconditions of each of the support predicates (and their negations) are $0 = 0$ $(0 \neq 0)$, $0 = 1$ $(0 \neq 1)$, $new = new + 1$ $(new \neq new + 1)$. As $0 = 0$, $0 \neq 1$, and $new \neq new + 1$ are trivially implied by the region $LOCK = 1 \wedge new = old$, we see that

$$\widehat{post}(LOCK = 1 \wedge new = old, \texttt{unlock();new++}) = LOCK = 0 \wedge LOCK \neq 1 \wedge new \neq old,$$

which is the reachable region of node 5 (in the figure the middle conjunct is dropped for clarity). □

We found in our implementation that our method outperforms the recursive subdivision method to construct the most precise overapproximation of *post* [12] significantly. In our experiments (see Section 6), we could prove all the properties using the Cartesian $\widehat{post}$ in much less running time. Since the abstract $\widehat{post}$ of a region is computed very frequently in the lazy-abstraction algorithm, any speedup in its computation results in a significant overall speedup. The Cartesian abstraction computed above takes time linear in the number of support predicates, as opposed to exponential in the number of support predicates for the most precise computation.

5.3 A focus operator for C

The focus operator adds to the set of supporting predicates of a region enough predicates to show infeasibility of an abstract error path. The focus operator is called with a region r, a sequence σ of labels, and an error region $\mathcal{E}$ such that $pre(\mathcal{E}, \sigma) \sqcap r \equiv \perp$. We use a proof-generating theorem prover to produce a minimal set Π of predicates that suffices for proving this equivalence: the predicates in Π are the ones that appear as atomic predicates in the proof of unsatisfiability of $pre(\mathcal{E}, \sigma) \sqcap r$ constructed by the theorem prover. However, to maintain the syntactic form of the predicates obtained along the path, all substitutions in weakest preconditions must be maintained *explicitly*. Thus, to compute $wp(\varphi, \texttt{x} = \texttt{e})$, instead of returning $\varphi[e/x]$, we introduce a fresh primed variable x' and return $x' = e \wedge \varphi[x'/x]$ (note that the variable x' acts as a Skolem constant). Finally, for each predicate $p \in \Pi$, we replace all primed instances of variable in p with the corresponding unprimed versions.

EXAMPLE 5: At the end of the forward-search phase (Figure 4) of the example in Section 2 we find that at node 2 the bad region intersected with the reachable region is empty. Thus, we call the focus operator $\Phi(r, \sigma, \mathcal{E})$ with, considering only the data region,

$$r = (LOCK = 0,\ \{LOCK = 0, LOCK = 1\}),$$
$$\sigma = \texttt{lock}(); \texttt{old} = \texttt{new} \cdot [\texttt{T}] \cdot \texttt{unlock}(); \texttt{new++} \cdot [\texttt{new} = \texttt{old}] \cdot \texttt{unlock}(),$$
$$\mathcal{E} = (True,\ \{\})$$

The focus operator first computes the following as the weakest precondition (with explicit substitutions of $\mathcal{E}$ w.r.t. σ):

$$old' = new \wedge LOCK' = 0$$
$$\wedge\ new' = new + 1 \wedge new' = old' \wedge LOCK' = 0$$

and conjoins it with r and submits the result to the proof-generating theorem prover. The prover says that the proof of unsatisfiability of the conjunction involves the predicates $new' = new + 1$, $new' = old'$, and $old' = new$, from which it is clear, by simply dropping the primes, that $new = old$ is a useful predicate. Hence the data region returned by focus is $((LOCK = 0,\ \{LOCK = 0, LOCK = 1, new = old\})$. $\square$

6. EXPERIMENTAL RESULTS

6.1 The BLAST toolkit

We have implemented a tool that applies lazy abstraction to model check safety properties of C programs. We handle all syntactic constructs of the C language, including pointers, structures, and procedures (leaving the constructs not in the predicate language uninterpreted). However, we do not model pointer arithmetic precisely, because we assume a *logical* model of memory; thus, we model the expression $p + i$, where p is a pointer and i is an integer, as yielding a pointer value that points to the object pointed to by p. Currently we handle procedure calls using an explicit stack and do not handle recursive functions, but the systems code we have analyzed is not recursive.

Our tool is written in Objective Caml, and consists of two main parts: (1) a functor implementing the LazyAbstraction algorithm, which takes a symbolic abstraction structure together with a focus operator as input, and (2) the symbolic abstraction structure and focus operator for C. The latter is made up of two parts: (a) the C front end, for which we use the C-Breeze C Compiler Infrastructure [23], which converts a program to its CFA, and (b) a module that contains the data structures for C regions as well as the functions pre, $\widehat{post}$, and focus. The boolean formulas over predicates that represent data regions are stored as BDDs [29] to get a canonical sum-of-product form for formulas. The BDD representation also allows easy boolean manipulation and inclusion checking. For the implication checks while computing $\widehat{post}$ and for the emptiness checks in the counterexample analysis we use the theorem prover Simplify [15]. For focus we use the proof-generating theorem prover Vampyre [6].

In order to be practical, the tool uses several optimizations to the general procedure described in Section 5. The cost is dominated by the cost of theorem proving, so we extensively optimize calls to the theorem prover. Theorem prover calls are cached, and for each query, several syntactic matching rules are checked first to cheaply solve the easy cases. In the computation of $\widehat{post}$, we check if a predicate p is affected by a statement s, and invoke theorem prover calls only on the subset of predicates for which $wp(p, s) \neq p$.

6.2 Driver verification in BLAST

We have run the implementation to check simple safety properties of some Linux and Microsoft Windows NT device drivers. The results are tabulated in Table 6. The properties we check are instrumented into the code by hand (by modifying certain library functions). This essentially involves constructing a monitor for the property of interest, and updating the monitor state whenever there is an interesting state change. For example, to check for correct locking behavior, we instrument calls to `lock()` and `unlock()` to call the monitor function, which updates some internal state, and goes to an error label if the internal state reaches an error configuration. We make an optimistic assumption about unknown library functions: we assume they do not affect the values of the tracked predicates. Finally, we check the code using a model of the kernel that exercises the driver. The model first calls the driver initialization routine, then calls the driver functions (read, write, etc.) in a loop, and finally unloads the driver.

The program `driver.c` is the driver code from [5]; the program `funlock.c` is the example from Section 2. The file `read.c` is a (simplified) serial driver and `floppy.c` is a floppy driver from the Microsoft Windows DDK. Finally, `qpmouse.c` and `ll_rw_block.c` are Linux device drivers (from the 2.4.9 kernel). We check locking disciplines in `floppy.c` and `ll_rw_block.c`. We check for null pointer dereferences in `qpmouse.c`. In `read.c` we check the property discussed in [5], namely, that the driver dispatch routine correctly handles both immediate and asynchronous services. In most cases, correctness cannot be proven using a data-independent analysis [19], and requires the automatic discovery of relevant predicates. Moreover, correctness spans several functions, so an interprocedural analysis is required.

In `ll_rw_block.c`, a spinlock is acquired in function `__make_request`, and passed on to function `add_request`. Under normal circumstances, `add_request` returns with the lock held, and `__make_request` unlocks it. However, in case there is a system bug, the driver invokes the macro `BUG()`, and the lock is unlocked. The first run of the tool did not model the behavior of `BUG()` (which causes the system to crash), and found an error trace that involved a call to `add_request` from `__make_request` with the lock held, a system bug, an unlock and return, and a subsequent unlock in `__make_request`. We then modified the specification to check for the locking discipline only when no system bugs occur. The property could now be proved.

In Table 6, LOC refers to lines of code. The total number of predicates is the total number required in the run; the active column gives the total number of support predicates active at any particular node in the reachability tree. There are many redundant theorem prover calls (the fraction of cached calls is very high). In all cases, the tool can prove the property quite fast.[2] Moreover, in several examples, the benefits of local predicates can be seen as the number of active predicates is less than the total number of predicates. This is especially true for `read.c`, because the property being checked has two disjoint branches, which require different sets of predicates to be verified. While this is not a complete experimental validation of the method, the initial results are encouraging. We are currently investigating the limits of the tool by running it on larger examples and checking for more complicated properties.

7. THEORETICAL ISSUES

In this section we consider two theoretical issues regarding lazy abstraction. First, we provide sufficient conditions for

[2] All times are on a 800MHz Pentium III with 256M RAM, and do not include parsing time.

Name	LOC	Predicates		Thm Prover Calls		Running
		Total	Active	Total	Cached	Time (s)
`driver.c`	95	3	3	260	165	0.08
`funlock.c`	40	4	3	340	182	0.14
`read.c`	370	28	18	5643	2862	4.42
`floppy.c`	6473	5	5	4137	3759	2.05
`qpmouse.c`	400	3	3	3117	2925	0.74
`ll_rw_block.c`	1281	9	7	10143	9483	5.82

Table 1: Experimental results with BLAST

the termination of the algorithm LazyAbstraction. Second, we show that it is undecidable to check if there is a finite predicate abstraction that is sufficient to prove a given safety property.

7.1 Termination

Let $\mathcal{S} = (S, \Sigma, \rightarrow)$ be a labeled transition system. For a state $s \in S$ and a sequence $\sigma \in \Sigma^*$ of labels, we write $s \xrightarrow{\sigma}$ if there is a state $s' \in S$ such that $s \xrightarrow{\sigma} s'$. Two states $s_1, s_2 \in S$ are *trace-equivalent* if for every $\sigma \in \Sigma^*$, we have $s_1 \xrightarrow{\sigma}$ iff $s_2 \xrightarrow{\sigma}$. The labeled transition system $\mathcal{S}$ has a *finite trace equivalence* if the trace-equivalence relation on S has a finite index.

Let $\mathcal{A}$ be an abstraction structure for $\mathcal{S}$ with region set R and extension function $[\![\cdot]\!]$. The abstraction structure $\mathcal{A}$ satisfies the *ascending-chain condition* if there does not exist an infinite strictly increasing sequence $r_0 \sqsubset r_1 \sqsubset \cdots \sqsubset r_k \sqsubset \cdots$ of regions in R, where $r \sqsubset r'$ if $[\![r]\!] \subset [\![r']\!]$.

In the following theorem we make two assumptions. First, in order to relate trace equivalence with the reachability of error states, we assume without loss of generality that error states have no outgoing transitions; that is, for every state $s \in [\![\mathcal{E}]\!]$, there is no label $l \in \Sigma$ such that $s \xrightarrow{l}$. Second, we assume that the `keep_subtree` function used by algorithm LazyAbstraction on line 11 always returns *false*, to avoid infinite loops as in Example 2.

THEOREM 3. [**Termination**] *Let $\mathcal{A}$ be an abstraction structure for a labeled transition system $\mathcal{S}$, and let Φ be a focus operator for $\mathcal{A}$. If*

(i) $\mathcal{S}$ has a finite trace equivalence, and

(ii) $\mathcal{A}$ satisfies the ascending chain condition,

then for every initial region r_0 and error region $\mathcal{E}$, the execution of LazyAbstraction$(\mathcal{A}, \Phi, r_0, \mathcal{E})$ *(Algorithm 2) terminates.*

In the proof, we use finite trace equivalence to show that a node in the reachability tree cannot be refined infinitely often, and then derive (by way of contradiction) an infinite ascending chain of regions for any nonterminating run. Unfortunately, the regions obtained from predicate abstraction with respect to an infinite predicate language usually do not satisfy the ascending chain condition. However, for a given labeled transition system with a finite trace equivalence, we may be able to choose a predicate language with a *finite* set of predicates, such as predicates that define (unions of) trace-equivalence classes. For example, this is the case for timed automata [2]. As the boolean combinations of a finite set of predicates trivially satisfy the ascending chain condition, the theorem guarantees termination.

7.2 Finite predicate abstraction is undecidable

The lazy-abstraction algorithm with predicate abstraction does not necessarily terminate on labeled transition systems with infinite state spaces. Indeed, we show that the problem whether there is a finite set of support predicates that witnesses a given safety property is undecidable. Let $\mathcal{L}$ be a predicate language for the labeled transition system $\mathcal{S} = (S, \Sigma, \rightarrow)$. Let Γ be a finite set of predicates from $\mathcal{L}$, and define the induced equivalence $\cong_\Gamma$ on S as $s_1 \cong_\Gamma s_2$ iff for all predicates $p \in \Gamma$, we have $s_1 \in [\![p]\!]$ iff $s_2 \in [\![p]\!]$; denote by $[s]_{\cong_\Gamma}$ the equivalence class of state s. The *quotient* $\mathcal{S}_{\cong_\Gamma}$ is the labeled transition system $(S/_{\cong_\Gamma}, \Sigma, \mapsto)$, where $S/_{\cong_\Gamma}$ is the (finite) set of equivalence classes of $\cong_\Gamma$, and for all $l \in \Sigma$, we have $s_{\cong} \xmapsto{l} s'_{\cong}$ iff there exist two states $s \in s_{\cong}$ and $s' \in s'_{\cong}$ with $s \xrightarrow{l} s'$. Note that every path in labeled transition system has a counterpart in the quotient, but not necessarily vice versa.

For a predicate language $\mathcal{L}$ for 2-counter machines, the $\mathcal{L}$-*finite abstraction problem* $\mathcal{L}$-FINABS is defined as follows:

- **Input** A 2-counter machine M, an initial state m_0 and a final state m_f of M, both definable in $\mathcal{L}$.

- **Output** "Yes" if either m_f is reachable in M from m_0, or there is a finite set Γ of predicates from $\mathcal{L}$ such that $[m_f]_{\cong_\Gamma}$ is not reachable in the quotient $M_{\cong_\Gamma}$ from $[m_0]_{\cong_\Gamma}$.

Notice that the problem is not trivial as the set of states reachable from m_0 (or the set of states that can reach m_f) may not be expressible as a boolean formula over predicates in $\mathcal{L}$.

Let Presburger-FINABS be the finite abstraction problem where $\mathcal{L}$ contains the control locations as propositions, and the quantifier-free formulas of Presburger arithmetic for constraining the counter values. We show undecidability by reduction from the halting problem for 2-counter machines. In particular, given M, we construct a 2-counter machine M' with initial state m'_0 and halting state m'_f such that M halts iff $\langle M', m'_0, m'_f \rangle$ is in Presburger-FINABS (we construct M' such that the reachable states of M' cannot be defined by a Presburger formula).

THEOREM 4. *Presburger-FINABS is complete for Σ^0_1 sets.*

More generally, let $\mathcal{L}$ be any predicate language for 2-counter machines. Then $\mathcal{L}$-FINABS is complete for Σ^0_1 sets.

Related work

Our work is related to counterexample-driven abstraction refinement [5, 10, 13, 28]. As in [3, 12, 20], we automatically construct a predicate abstraction by using an automatic theorem prover to answer satisfiability queries. However, all previous counterexample-driven refinement methods do not reuse the work done in one pass in the next pass: after every pass, the abstraction is constructed from scratch, and the new system is model checked. The results from model checking the previous passes are not reused, and a large part of the symbolic state space may be traversed repeatedly, even though a coarser abstraction is sufficient to prove the property of interest for that region. Lazy abstraction takes advantage of previous runs by abstracting locally.

Dataflow and type-based analyses have been used to check safety properties of systems code (e.g., [17, 19, 30]). These analyses typically ignore data dependence and may generate false positives owing to infeasible paths. Our work can be seen as an extension to such analyses by introducing path sensitivity to the analysis. Moreover, counterexample-driven refinement avoids an explosion of spurious error traces.

Acknowledgments

We thank Wes Weimer and Jeff Foster for many useful discussions.

8. REFERENCES

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] R. Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[3] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI 01: Programming Language Design and Implementation*, pp. 203–213. ACM, 2001.

[4] T. Ball, A. Podelski, and S.K. Rajamani. Boolean and Cartesian abstractions for model checking C programs. In *TACAS 01: Tools and Algorithms for Construction and Analysis of Systems*, LNCS 2031, pp. 268–283. Springer, 2001.

[5] T. Ball and S.K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN Workshop*, LNCS 2057, pp. 103–122. Springer, 2001.

[6] D. Blei *et al.* Vampyre: A proof generating theorem prover. http://www.eecs.berkeley.edu/ rupak/Vampyre.

[7] B. Boigelot and P. Godefroid. Symbolic verification of communication protocols with infinite state spaces using QDDs. In *CAV 96: Computer Aided Verification*, LNCS 1102, pp. 1–12. Springer, 1996.

[8] B. Boigelot and P. Wolper. Symbolic verification with periodic sets. In *CAV 94: Computer Aided Verification*, LNCS 818, pp. 55–67. Springer, 1994.

[9] A. Bouajjani and P. Habermehl. Symbolic reachability analysis of FIFO-channel systems with nonregular sets of configurations. *Theoretical Computer Science*, 221:211–250, 1999.

[10] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV 00: Computer Aided Verification*, LNCS 1855, pp. 154–169. Springer, 2000.

[11] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, 1999.

[12] S. Das, D.L. Dill, and S. Park. Experience with predicate abstraction. In *CAV 99: Computer Aided Verification*, LNCS 1633, pp. 160–171. Springer, 1999.

[13] S. Das and D.L. Dill. Successive approximation of abstract transition relations. In *LICS 01: Logic in Computer Science*, pp. 51–60. IEEE, 2001.

[14] G. Delzanno and A. Podelski. Model checking in CLP. In *TACAS 99: Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 1579, pp. 223–239. Springer, 1999.

[15] D. Detlefs, G. Nelson, and J. Saxe. Simplify theorem prover. http://research.compaq.com/SRC/esc/Simplify.html.

[16] E.W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[17] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI 00: Operating System Design and Implementation.* Usenix Association, 2000.

[18] A. Finkel, S. P. Iyer, and G. Sutre. Well-abstracted transition systems. In *CONCUR 00: Concurrency Theory*, LNCS 1877, pp. 566–580. Springer, 2000.

[19] J.S. Foster, T. Terauchi, and A. Aiken. *Flow-sensitive Type Qualifiers.* Technical Report CSD-01-1162, University of California, Berkeley, 2001.

[20] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV 97: Computer Aided Verification*, LNCS 1254, pp. 72–83. Springer, 1997.

[21] D. Gries. *The Science of Programming.* Springer, 1981.

[22] T. A. Henzinger and R. Majumdar. A classification of symbolic transition systems. In *STACS 00: Theoretical Aspects of Computer Science*, LNCS 1770, pp. 13–34. Springer, 2000.

[23] C. Lin *et al.* C-breeze: C compiler infrastructure. http://www.cs.utexas.edu/users/c-breeze.

[24] J.M. Morris. A general axiom of assignment. In *Theoretical Foundations of Programming Methodology*, International Summer School, pp. 25–34. Reidel, 1982.

[25] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *DAC 01: Design Automation Conference*, pp. 530–535, 2001.

[26] G. Nelson. *Techniques for Program Verification.* Technical Report CSL81-10, Xerox Palo Alto Research Center, 1981.

[27] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: Combining specification, proof checking, and model checking. In *CAV 96: Computer Aided Verification*, LNCS 1102, pp. 411–414. Springer, 1996.

[28] H. Saidi. Model checking guided abstraction and analysis. In *SAS 00: Static Analysis Symposium*, LNCS 1824, pp. 377–396. Springer, 2000.

[29] F. Somenzi. Colorado university decision diagram package. http://vlsi.colorado.edu/pub, 1998.

[30] N. Sterling. Warlock: A static data race analysis tool. In *USENIX Winter 1993 Technical Conference*, pp. 97–106, 1993.