Temporal-Safety Proofs for Systems Code*

Thomas A. Henzinger[1] Ranjit Jhala[1] Rupak Majumdar[1]
George C. Necula[1] Grégoire Sutre[2] Westley Weimer[1]

[1] EECS Department, University of California, Berkeley
[2] LaBRI, Université de Bordeaux, France

Abstract. We present a methodology and tool for verifying and certifying systems code. The verification is based on the *lazy-abstraction* paradigm for intertwining the following three logical steps: construct a predicate abstraction from the code, model check the abstraction, and automatically refine the abstraction based on counterexample analysis. The certification is based on the *proof-carrying code* paradigm. Lazy abstraction enables the automatic construction of small proof certificates. The methodology is implemented in BLAST, the Berkeley Lazy Abstraction Software verification Tool. We describe our experience applying BLAST to Linux and Windows device drivers. Given the C code for a driver and for a temporal-safety monitor, BLAST automatically generates an easily checkable correctness certificate if the driver satisfies the specification, and an error trace otherwise.

1 Introduction

An important goal of software engineering is to facilitate the construction of *correct* and *trusted* software. This is especially important for low-level systems code, which usually cannot be shielded from causing mischief by runtime protection mechanisms. Correctness requires technologies for the *verification* of software, which enable engineers to produce programs with few or no bugs. Trust requires technologies for the *certification* of software, which assure users that the programs meet their specifications, e.g., that the code will not crash, or leak vital secrets. Both verification and certification are most effective when performed for actual code, not for separately constructed abstract models.

For verification, *model-checking* based approaches have the advantages of being, unlike most theorem-proving based approaches, fully automatic and, unlike most program-analysis based approaches, capable of checking path-sensitive properties. The main obstacle to model checking is, of course, scalability. Recently, *abstraction-refinement* based techniques have been developed for (mostly) automatically constructing and model checking abstract models derived directly from code [3, 7, 25, 16]. However, the main problem faced by such techniques is

* This work was supported in part by the NSF ITR grants CCR-0085949,CCR-0081588, the NSF Career grant CCR-9875171, the DARPA PCES grant F33615-00-C-1693, the MARCO GSRC grant 98-DT-660, the SRC contract 99-TJ-683, a Microsoft fellowship, and gifts from AT&T Research and Microsoft Research.

still scalability: for large software systems and complicated specifications, the abstraction process can take too much time and space, and the resulting model may again be too large to be model checked. The technique of *lazy abstraction* [15] is an attempt to make counterexample-guided abstraction refinement for model checking [6, 3] scalable by localizing the abstraction process and avoiding unnecessary work. Lazy abstraction builds an abstract model on-the-fly, during model checking, and on demand, so that each predicate is used only in abstracting those portions of the state space where it is needed to rule out spurious counterexamples. This is unlike traditional predicate-abstraction based model checking [13, 8, 1], which constructs a uniform predicate abstraction from a given system and a given set of predicates. The result is a nonuniform abstract model, which provides for every portion of the state space just as much detail as is necessary to prove the specification. Also, lazy abstraction short-circuits the traditional abstract-verify-refine loop [3], and avoids the repetition of work in successive abstraction phases and in successive model-checking phases.

For certification, *proof-carrying code* (PCC) [18] has been proposed as a mechanism for witnessing the correct behavior of untrusted code. Here, the code producer sends to the consumer the code annotated with loop invariants and function pre- and postconditions, as well as a proof of correctness of a verification condition, whose validity guarantees the correctness of the code with respect to the specification. From the code and the annotations, the consumer can build the verification condition and check the supplied proof for correctness. The checking of the proof is much simpler than its construction. In particular, by encoding the proof, proof checking becomes a type-checking problem. Proof-carrying code has the advantages of avoiding trusted third parties, and of being tamper-proof, because tampering with either the proof or the code will result in an invalid proof. The main problem faced by PCC is that a user may have to supply annotations such as loop invariants. In [18] it is shown how loop invariants can be inferred automatically for proofs of type and memory safety, but the problem of inferring invariants for behavioral properties, such as temporal safety, remains largely open [11].

We show that lazy abstraction can be used naturally and efficiently to construct small correctness proofs for temporal-safety properties in a PCC based framework. The proof generation is intertwined with the model-checking process: the data structures produced by lazy abstraction automatically supply the annotations required for proof construction, and provide a decomposition of the proof which leads to a small correctness certificate. In particular, using abstraction predicates only where necessary keeps the proof small, and using the model checker to guide the proof generation eliminates the need for backtracking, e.g., in the proof of disjunctions. Our strategy to generate proofs from model-checking runs is different from [17, 22]. We exploit the structure of sequential code so that the proof is an invariant for every control location, along with local checks for every edge of the control-flow graph that the invariants are sound. Both [17, 22] work at the transition-system level. On the other hand, they generate proofs for properties more general than safety.

```
1:    do {
          lock();
          old = new;
2:        if (*) {
3:            unlock();
              new++;
          }
4:    } while (new != old);
5:    unlock();
      exit;
```

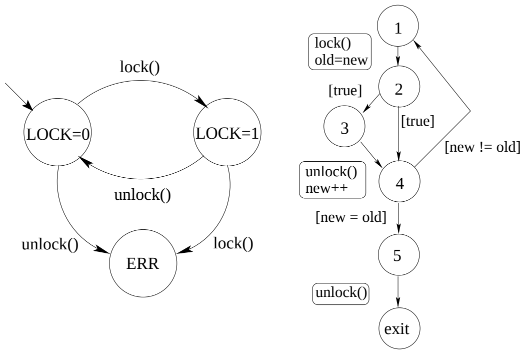

Fig. 1. (a) The program `Example`. (b) Locking specification. (c) CFA.

We have implemented proof generation in the tool BLAST [15] for model checking C programs. BLAST has been inspired by the Microsoft SLAM project [3], and attempts to improve on the abstract-verify-refine methodology by the use of lazy abstraction for model construction, and the use of theorem proving for predicate discovery. We focus here on the automatic verification and certification of device drivers. Device drivers are written at a fairly low level, but must meet high-level specifications, such as locking disciplines, which are difficult to verify without path-sensitive analysis. They are critical for the correct functioning of modern computer systems, but are written by untrusted third-party vendors. Some studies show that device drivers typically contain 7 times as many bugs as the rest of the OS code [5]. Using BLAST, we have run 10 examples of Linux and Windows device drivers, of up to 60K lines of C code. We have been able to discover several errors, and construct, fully automatically, small proofs of correctness, each less than 150K. This demonstrates that lazy-abstraction based verification and proof generation scales well to large software systems.

2 An Example

We consider a small example to give an overview of lazy abstraction and proof generation. Consider the program in Figure 1(a). A temporal-safety specification is a monitor automaton with error locations. Consider the locking discipline specified by the automaton of Figure 1(b). The monitor uses a global variable $LOCK$, which is 1 when the lock is held, and 0 otherwise. An error occurs if the function `lock` is called with the lock held ($LOCK = 1$) or `unlock` is called without the lock held ($LOCK = 0$). The program and specification are input to BLAST as C code, which are then combined to get a single sequential program with a special error location that is reachable iff the specification is not met. Thus we assume that the program has a special error label, and safety checking is checking whether the error label is reachable.

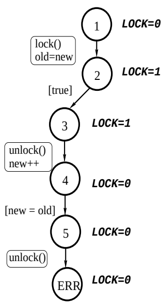

Fig. 2. Forward search.

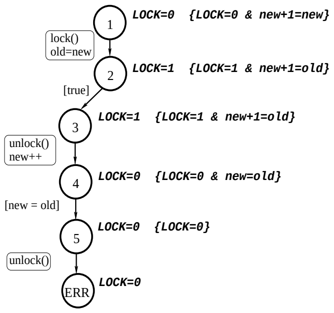

Fig. 3. Backward counterexample analysis.

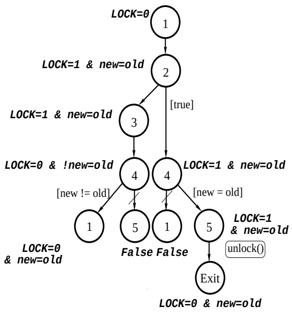

Fig. 4. Search with new predicate.

We represent programs as *control flow automata* (CFA). A CFA is a directed graph with vertices corresponding to control points of the program (begins and ends of basic blocks), and edges corresponding to program operations. An edge is labeled either by a *basic block* of instructions that are executed along that edge, or by an *assume predicate* that represents the condition that must hold for the edge to be taken. Figure 1(c) shows the CFA for the program `Example`. The program labels correspond to CFA vertices with the same label. The edges labeled with boxes represent basic blocks; those labeled with $[\cdot]$ represent assume predicates. The condition of the `if (*)` statement is not modeled. We assume that either branch can be taken, hence both outgoing edges are labeled with `[true]`, which stands for the assume predicate `true`.

2.1 Verification

The lazy abstraction algorithm comprises two phases. In the forward-search phase, we build a *reachability tree*, which represents a portion of the reachable, abstract state space of the program. Each node of the tree is labeled by a vertex of the CFA and a boolean formula, called the *reachable region*, constructed as a combination of a finite set of *abstraction predicates*. Each edge of the tree is labeled by a basic block or an assume predicate. Each path in the tree corresponds to a path in the CFA. The reachable region of a node describes the reachable states of the program in terms of the abstraction predicates, 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 node 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 we lost too much information by restricting ourselves to a particular set of abstraction predicates). In the latter case, we ask a theorem prover to suggest additional abstraction predicates which rule out that particular spurious counterexample. By iterating the two phases of forward search and backwards counterexample analysis, different portions of the reachability tree will use different sets of abstraction predicates.

We now describe the lazy-abstraction algorithm on the program `Example`. From the specification we know that it is important whether or not the lock is held, hence we start by considering the two abstraction predicates[1] $LOCK = 1$ and $LOCK = 0$[2].

Forward search. Consider Figure 1. We construct a reachability tree in depth-first order. The root is labeled with the entry vertex of the program (location 1). The reachable region of the root is a boolean expression over the abstraction predicates which includes the precondition of the program ($LOCK = 0$, as initially the lock is not held). The reachable region of each node in the tree is obtained from the reachable region of the parent and the instructions labeling the edge between the parent and the node, by an overapproximate successor computation with respect to the set of abstraction predicates. This computation involves calls to a theorem prover and is also used in the generation of correctness certificates. In the example, the search finds a path to an error node, namely, $(1, 2, 3, 4, 5)$.

Backwards counterexample analysis. We check if the path from the root to the error node is a genuine counterexample or results from the abstraction being too coarse. To do this, we symbolically simulate the error trace backwards in the original program. As we go backwards from the error node, we try to find the first node in the reachability tree where the abstract trace fails to have a concrete counterpart. If we find such a *pivot node*, then we conclude that the counterexample is spurious and refine the abstraction from the pivot node on. If the analysis goes back to the root without finding a pivot node, then we have found a real error in the program.

Figure 3 shows the result of this phase. In the figure, for each node, the formula in the curly braces, called the *bad region*, represents the set of states that can go from the corresponding control location to an error by executing the sequence of instructions labeling the path from the node to the error node. Formally, the bad region of a node is the intersection of the reachable region of the node with the weakest precondition of *true* with respect to the sequence of instructions labeling the path in the reachability tree from the node to the error node. It is computed inductively, starting backwards from the error node, which has the bad region *true*. Note that unlike the search phase, the counterexample analysis is precise: we track all predicates obtained along a path. In Figure 3, we find that the bad region at location 1 is *false*, which implies that the counterexample is spurious. In [15] we show how the *proof* that the bad region at a node is empty (i.e., unsatisfiable) can be used to extract additional abstraction predicates which rule out the spurious counterexample. In the example, we find that $new = old$ is such an important predicate.

Search with new predicates. We again start the forward-search phase, starting from the pivot node, but this time we track the predicate $new = old$ in

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

[2] This is not necessary; we can also start with the empty set of abstraction predicates.

addition to $LOCK = 0$ and $LOCK = 1$. The resulting search tree is shown in Figure 4. Notice that we can stop the search at the leaf labeled 1: as the states satisfying the reachable region $LOCK = 0 \wedge new = old$ are a subset of those satisfying the reachable region $LOCK = 0$ of root, the subtree constructed from the leaf would be included in the subtree of the root. In the new reachability tree, no error node is reachable. Hence we conclude that the program `Example` satisfies the locking specification.

2.2 Certification

To certify that a program satisfies its specification, we use a standard *temporal-safety rule* from deductive verification: given a transition system, if we can find a set I of states such that (1) I contains all initial states, (2) I contains no error states, and (3) I is closed under successor states, then the system cannot reach an error state from an initial state. If (1)–(3) are satisfied, then I is called an *invariant* set. In our setting, the temporal-safety rule reduces to supplying for each vertex q of the CFA an invariant formula $I(q)$ such that

1. $(LOCK = 0) \Rightarrow I(1)$;
2. $I(\mathtt{ERR}) = \mathit{false}$;
3. for each pair of CFA vertices q and q' with an edge labeled `op` between them, $sp(I(q), \mathtt{op}) \Rightarrow I(q')$, where sp is the strongest-postcondition operator [10].

Thus, to provide a proof of correctness, it suffices to supply a location invariant $I(q)$ for each vertex q of the CFA, and proofs that the supplied formulas meet the above three requirements.

The location invariants can be mined from the reachability tree. In particular, the invariant for q is the disjunction of all reachable regions that label the nodes in the tree which correspond to q. For example, $I(4)$ is $(LOCK = 0 \wedge \neg new = old) \vee (LOCK = 1 \wedge new = old)$. It is easy to check that $(LOCK = 0) \Rightarrow I(1)$, since the root of the reachability tree is labeled by the precondition of the program $(LOCK = 0)$. Also, as there is no node labeled `ERR` in the tree, we get the second requirement by definition. The interesting part is checking that the third requirement, that for each edge $q \xrightarrow{\mathtt{op}} q'$ of the CFA, $sp(I(q), \mathtt{op}) \Rightarrow I(q')$. Consider the edge $4 \xrightarrow{[\mathtt{new!=old}]} 1$. We need to show that

$$sp((LOCK = 0 \wedge \neg new = old) \vee (LOCK = 1 \wedge new = old), [\mathtt{new!} = \mathtt{old}]) \Rightarrow (LOCK = 0).$$

By distributing sp over $\vee$, we are left with the proof obligation $((LOCK = 0) \vee \mathit{false}) \Rightarrow (LOCK = 0)$. To prove this, notice that the disjuncts on the left can be broken down into subformulas obtained from the reachable regions of individual nodes. Hence we can show the implication by matching each subformula with the appropriate successor on the right. So we get the two obligations $(LOCK = 0) \Rightarrow (LOCK = 0)$ and $\mathit{false} \Rightarrow (LOCK = 0)$. Exactly these obligations were generated in the forward-search phase when computing abstract successors. Each obligation is discharged, and the whole proof assembled, using a proof-generating theorem prover.

3 From Lazy Abstraction to Verification Conditions

3.1 Control flow automata

Syntax. A *control flow automaton* C is a tuple $\langle Q, q_0, X, \mathtt{Op}, \rightarrow\rangle$, where Q is a finite set of control locations, q_0 is the initial control location, X is a finite set of typed variables, $\mathtt{Op}$ is a set of operations, and $\rightarrow \subseteq (Q \times \mathtt{Op} \times Q)$ is a finite set of edges labeled with operations. An edge $(q, \mathtt{op}, q')$ is also denoted $q \xrightarrow{\mathtt{op}} q'$. The set $\mathtt{Op}$ of operations contains (1) *basic blocks* of instructions, i.e., finite sequences of assignments `lval = exp`, where `lval` is an lvalue from X (i.e., a variable, structure field, or pointer dereference), and `exp` is an arithmetic expression over X; and (2) *assume predicates* `assume(p)`, where `p` is a boolean expression over X (arithmetic comparison or pointer equality), representing a condition that must be true for the edge to be taken. For ease of exposition we describe our method only for CFAs without function calls; the method can be extended to handle function calls in a standard way (and function calls are handled by the BLAST implementation). A program written in an imperative language such as C can be transformed to a CFA [21].

Semantics. The set $\mathcal{V}_X$ of *(data) valuations* over the variables X contains the type-preserving functions from X to values. A *state* is a pair in $Q \times \mathcal{V}_X$. A *region* is a set of states; let $\mathcal{R}$ be the set of regions. We use first-order formulas over some fixed set of relation and function symbols to represent regions. The semantics of operations is given in terms of the strongest-postcondition operator [10]: $sp(r, \mathtt{op})$ of a formula r with respect to an operation $\mathtt{op}$ is the strongest formula whose truth holds after $\mathtt{op}$ terminates when executed in a valuation that satisfies r. For a formula $r \in \mathcal{R}$ and operation $\mathtt{op} \in \mathtt{Op}$, the formula $sp(r, \mathtt{op}) \in \mathcal{R}$ is syntactically computable. However, we leave certain relation and function symbols (e.g., multiplication) uninterpreted, and the set of states denoted by $sp(r, \mathtt{op})$ may be overapproximate. A location $q \in Q$ is *reachable from a precondition* $Pre \in \mathcal{R}$ if there is a path $q_0 \xrightarrow{\mathtt{op}_1} q_1 \xrightarrow{\mathtt{op}_2} \cdots \xrightarrow{\mathtt{op}_n} q_n$ in the CFA and a sequence of formulas r_i, for $i \in \{0, \ldots, n\}$, such that $q_n = q$, $r_0 = Pre$, $r_n \not\Leftrightarrow \mathit{false}$, and $sp(r_i, \mathtt{op}_{i+1}) = r_{i+1}$ for all $0 \le i < n$. The witnessing path is called a *feasible* path from (q_0, Pre) to q.

3.2 Verification conditions

Let $C = \langle Q, q_0, X, \mathtt{Op}, \rightarrow\rangle$ be a CFA with a precondition $Pre \in \mathcal{R}$ and a special *error location* $q_{\mathcal{E}} \in Q$. We consider the specification that there is no feasible path in C from (q_0, Pre) to the error location $q_{\mathcal{E}}$. Such a path is called an *error trace*. Note that every temporal-safety property can be checked in this way, using a product with a suitable monitor automaton. A *verification condition* (VC) [10] for a program and a specification is a first-order formula r such that the validity of r ensures that the program adheres to the specification. In order to produce the VC we require that every location q of the CFA is annotated with a formula, the *invariant region* $I(q)$. Given the invariants $I\colon Q \to \mathcal{R}$, the verification condition

$\mathcal{VC}(C, Pre, q_{\mathcal{E}}, I)$ that asserts the correctness of the CFA is

$$(Pre \Rightarrow I(q_0)) \ \wedge \ (I(q_{\mathcal{E}}) = \mathit{false}) \ \wedge \bigwedge_{q \xrightarrow{\mathtt{op}} q'} \left(sp(I(q), \mathtt{op}) \Rightarrow I(q') \right),$$

which contains one conjunct for each edge of C. In other words, the VC states that the invariant of each location is an inductive overapproximation of the states that can be reached at that location. Formally, if $\mathcal{VC}(C, Pre, q_{\mathcal{E}}, I)$ is valid, then the error location $q_{\mathcal{E}}$ is not reachable in C from Pre [10].

3.3 Invariant generation via lazy abstraction

The invariants required for the VC are got automatically from the data structure built by the lazy-abstraction algorithm [15]. We assume that both the code producer and consumer are working at the CFA level. Consider a CFA $C = \langle Q, q_0, X, \mathtt{Op}, \rightarrow \rangle$. Let T be a rooted tree, where each node is labeled by a pair $(q, r) \in Q \times \mathcal{R}$, and each edge is labeled by an operation $\mathtt{op} \in \mathtt{Op}$. We write $\mathtt{n}\!:\!(q, r)$ if node $\mathtt{n}$ is labeled by control location q and region r. If there is an edge from $\mathtt{n}\!:\!(q, r)$ to $\mathtt{n}'\!:\!(q', r')$ labeled by $\mathtt{op}$, then node $\mathtt{n}'$ is a $(\mathtt{op}, q')$-*son* of node $\mathtt{n}$. We write Leaves_T for the set of leaf nodes, and Int_T for the set of internal nodes. The tree T is a *reachability tree* for the CFA C if (1) each internal node $\mathtt{n}\!:\!(q, r)$ has a $(\mathtt{op}, q')$-son $\mathtt{n}' : (q', r')$ for each edge $q \xrightarrow{\mathtt{op}} q'$ of C; (2) if $\mathtt{n}' : (q', r')$ is a $(\mathtt{op}, q')$-son of $\mathtt{n}\!:\!(q, r)$, then $sp(r, \mathtt{op}) \Rightarrow r'$; and (3) for each leaf node $\mathtt{n}\!:\!(q, r)$, there are internal nodes $\mathtt{n}_1\!:\!(q, r_1), \ldots, \mathtt{n}_k\!:\!(q, r_k)$ such that $r \Rightarrow (r_1 \vee \ldots \vee r_k)$. The reachability tree is *safe* w.r.t. to precondition $Pre \in \mathcal{R}$ and error location $q_{\mathcal{E}} \in Q$ if (1) the root has the form $\mathtt{n}\!:\!(q_0, Pre)$ and (2) for all nodes of the form $\mathtt{n}\!:\!(q_{\mathcal{E}}, r)$, we have $r \Leftrightarrow \mathit{false}$.

Theorem 1. [15] *Let C be a CFA with precondition Pre and error location $q_{\mathcal{E}}$. If the lazy-abstraction algorithm $\mathsf{LA}(C, Pre, q_{\mathcal{E}})$ terminates, then it returns either an error trace, or a safe reachability tree.*

The reachability problem for CFAs is undecidable, so LA may not terminate on all inputs. However, it terminates on all our examples (and [15] contains some termination criteria). A safe reachability tree witnesses the correctness of the CFA, and the reachable regions that label its nodes provide invariants. In particular, if T is a safe reachability tree for $(C, Pre, q_{\mathcal{E}})$, then the *invariant region* $I(q)$ of each control location q of C is defined to be the union of all reachable regions of *internal* nodes of T labeled by q, that is, $I(q) = \bigvee_{\mathtt{n}:(q,r) \in \mathit{Int}_T} r$. From these invariants, we will generate a proof of the verification condition $\mathcal{VC}(C, Pre, q_{\mathcal{E}}, I)$. In fact, we modify the lazy-abstraction algorithm to guide, during the construction of a safe reachability tree, the generation of the correctness proof.

4 Proof Generation

Representing proofs. We encode the proof of the verification condition in LF [14], so that proof checking reduces to a linear-time type-checking problem.

The logic we encode in LF is first-order logic with equality and special relation and function symbols for arithmetic and memory operations. The encoding is standard [14, 18], and is omitted. The inference rules of the proof system include the standard introduction and elimination rules for the boolean connectives used in natural deduction with hypothetical judgments [23], together with special rules for equality, arithmetic, and memory operations. In BLAST, proofs are represented in binary form using Implicit LF [20]. We use the proof encoding and checking mechanism of an existing PCC implementation to convert proofs from a textual representation to binary, and to check proofs.

Generating proofs. Given a safe reachability tree T for a CFA C with precondition Pre and error location $q_{\mathcal{E}}$, we must prove the three conjuncts of the verification condition $\mathcal{VC}(C, \mathit{Pre}, q_{\mathcal{E}}, I)$, namely, that (1) the precondition implies the invariant of the initial location, (2) the invariant of the error location is false, and (3) the invariants are closed under postconditions. We prove each conjunct separately. The first conjunct of the VC is $\mathit{Pre} \Rightarrow I(q_0)$. Since the root of T is labeled with the control location q_0 and the reachable region Pre, the precondition Pre is a disjunct of the invariant $I(q_0)$. Hence, the first conjunct follows from simple propositional reasoning. The second conjunct is $I(q_{\mathcal{E}}) = \mathit{false}$. This is again true by the construction of T and the invariants. For the third conjunct, it suffices to show a proof obligation for each edge of the CFA. We use distributivity of postconditions and implication over disjunction to break the obligation for a CFA edge into individual obligations for the edges of the safe reachability tree that correspond to the CFA edge. Then we discharge the smaller proof obligations by relating them to the construction of the reachable regions during the forward-search phase of the lazy-abstraction algorithm.

Consider the edge $q \xrightarrow{\mathtt{op}} q'$ of C, and the corresponding proof obligation $sp(I(q), \mathtt{op}) \Rightarrow I(q')$. Recall that $I(q)$ is the union of all reachable regions of nodes of T labeled by q. Since sp distributes over disjunction, it suffices to prove $\left(\bigvee_{\mathtt{n}:(q,r_n)\in \mathit{Int}_T} sp(r_n, \mathtt{op})\right) \Rightarrow \left(\bigvee_{\mathtt{m}:(q',r_m)\in \mathit{Int}_T} r_m\right)$, or equivalently, to prove $\bigwedge_{\mathtt{n}:(q,r_n)\in \mathit{Int}_T} \left(sp(r_n, \mathtt{op}) \Rightarrow \bigvee_{\mathtt{m}:(q',r_m)\in \mathit{Int}_T} r_m\right)$. Hence, it suffices to prove one obligation for each internal node labeled by q. For every internal node $\mathtt{n}:(q, r_n)$ of T, there is a unique $(\mathtt{op}, q')$-son $\mathtt{m}:(q', r_m)$ of $\mathtt{n}$. This observation is essential for guiding the proof generation. We break the proof of $sp(r_n, \mathtt{op}) \Rightarrow I(q')$ into two cases, corresponding to whether $\mathtt{m}$ is an internal node or a leaf.

If $\mathtt{m}$ is an internal node of T, then it suffices to prove $sp(r_n, \mathtt{op}) \Rightarrow r_m$. We generate a proof for this by considering the computation that put the edge from $\mathtt{n}$ to $\mathtt{m}$ into the safe reachability tree. Assume that $r_n = \bigvee R_i$, where each disjunct R_i is a conjunction of literals, and each literal is either an abstraction predicate or its negation. Then $r_m = \bigvee R'_i$, where for each i, the disjunct R'_i is computed as an overapproximate (abstract) successor region of R_i as follows [15]: the literal p (resp., $\neg p$) appears in R'_i iff $sp(R_i, \mathtt{op}) \Rightarrow p$ (resp., $sp(R_i, \mathtt{op}) \Rightarrow \neg p$) is valid. Distributing sp over disjunction, it suffices to prove $sp(R_i, \mathtt{op}) \Rightarrow r_m$, which further reduces to proving $sp(R_i, \mathtt{op}) \Rightarrow R'_i$ for each i. This is proved by putting together exactly the proofs used in the abstract successor computation.

If $\mathtt{m}$ is a leaf of T, then we break the proof into three parts. First, we generate a proof for $sp(r_n, \mathtt{op}) \Rightarrow r_m$ as above. Second, we check why the node $\mathtt{m}$ is a leaf of the safe reachability tree. There must be a set $S = \{\mathtt{k} \mid \mathtt{k}\!:\!(q', r_k)\}$ of nodes of T such that $r_m \Rightarrow \bigvee_{\mathtt{k} \in S} r_k$; this set can be obtained from the lazy-abstraction algorithm. We extract the proof of the above implication. Third, we notice that $\bigvee_{\mathtt{k} \in S} r_k \Rightarrow I(q')$, by the definition of $I(q')$. These three proofs are combined into a proof of $sp(r_n, \mathtt{op}) \Rightarrow I(q')$.

Our proof generation is optimized in two ways. First, we use the intermediate steps of the model checker to break a proof that invariants are closed under post-conditions into simpler proofs about the disjuncts that make up the invariants. Moreover, these proofs are available from the forward-search phase of the model checker. Second, we reduce the size of the proof by using a coarse, nonuniform abstraction sufficient for proving correctness, as provided by the lazy-abstraction algorithm. This eliminates predicates that are not essential to correctness and submits fewer obligations to the proof generator than would a VC obtained by direct symbolic simulation of all paths.

5 Experiments

5.1 The Blast toolkit

We have implemented a tool that applies lazy abstraction and proof generation for temporal-safety properties of C programs. The input to Blast is a C program and a safety monitor written in C; these are compiled into a CFA with a special error state. The lazy-abstraction algorithm runs on the CFA and returns either a genuine error trace or a proof of correctness (or fails to terminate). Our handling of C features follows that of [1]. We handle all syntactic constructs of C, including pointers, structures, and procedures (leaving the constructs not in the predicate language uninterpreted). However, we model integer arithmetic as infinite-precision arithmetic (no wraparound), and we assume a *logical* model of the memory. In particular, 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. Hence, we do not model pointer arithmetic precisely.

Blast makes use of several existing tools. We use the CIL compiler infrastructure [21] to construct CFAs from C programs. We use the CUDD package [24] to represent regions as BDDs over sets of abstraction predicates. Finally, we use the theorem prover Simplify [9] for abstract-successor computations and inclusion checks, and the (slower) proof-generating theorem prover Vampyre [4] where proofs are required. The cost of verification and certification is dominated by the cost of theorem proving, so we incorporate automatic lemma extraction by caching theorem prover calls. Our experiments show that many atomic proof obligations that arise during the entire process are identical, and so the size of the proof in dag representation is considerably smaller than a derivation tree.

5.2 Checking device drivers with Blast

The Linux kernel has two primitive locking functions, `spin_lock` and `spin_unlock`, which are used extensively by device drivers to ensure mutual exclusion. We checked the locking specification from Figure 1(b) on the Linux drivers listed in the top part of Table 1. We modeled the behavior of the kernel using a nondeterministic `main` function which calls all possible device functions registered with the kernel. Column 2 shows the code size. Column 3 gives the total number of abstraction predicates; the number of active predicates is the maximum number of predicates considered at any one point in the model-checking process. The times are for a 700 MHz Pentium III processor with 256M RAM, and for verification only; in all cases parsing time is less than a second. Column 5 shows the part of verification time spent in counterexample analysis. We found several interprocedural bugs in the locking behavior. For example, we found paths in `aha152x.c` where a lock can be acquired twice; see also [12]. On the other hand, some correctness proofs use nontrivial data dependencies involving conditional locking (e.g., `tlan.c`), or unlocking based on some status code (e.g., `ide.c`), and come up as false positives in [12].

We also ran Blast on an I/O request packet (IRP) completion specification for several device drivers included in the Microsoft Windows NT DDK[3]; this is shown in the bottom part of Table 1. The Windows OS uses an IRP data structure to communicate with a kernel-mode device driver. The IRP completion specification gives correct ways for Windows device drivers to handle IRPs by specifying a sequence of functions to be called in a certain order, and specific return codes. The entire specification is captured by a safety-monitor automaton with 22 states. To check this specification against some drivers from the DDK, we wrote a model of the rest of the kernel as nondeterministic functions modeling the interface, and we wrote a driver `main` function that calls the dispatch routines nondeterministically. We found several bugs in the drivers involving incorrect status codes. These bugs were independently confirmed by Slam [2].

5.3 Experiences with Blast

We conjecture that the following are the main reasons for the efficiency of the lazy-abstraction algorithm.

1. *Sparse reach set.* While the state space of programs is huge, the set of states that are *reachable* is often very small. Traditional abstraction schemes compute the abstract transition relation for the entire state space. Lazy abstraction, on the other hand, considers only the reachable part of the state space. We believe that this is the single most important advantage of lazy abstraction.
2. *Local structure.* Instead of building a uniform predicate abstraction, the lazy-abstraction algorithm exploits the control-flow structure of the program by using predicates only where required. This ensures that abstract successors

[3] Available from http://www.microsoft.com/ddk.

Program	Postprocessed LOC	Predicates		BLAST Time (sec)	Ctrex analysis (sec)	Proof Size (bytes)
		Total	Active			
qpmouse.c	23539	2	2	0.50	0.00	175
ide.c	18131	5	5	4.59	0.01	253
aha152x.c	17736	2	2	20.93	0.00	
tlan.c	16506	5	4	428.63	403.33	405
cdaudio.c	17798	85	45	1398.62	540.96	156787
floppy.c	17386	62	37	2086.35	1565.34	
[fixed]		93	44	395.97	17.46	60129
kbfiltr.c	12131	54	40	64.16	5.89	
		48	35	256.92	165.25	
[fixed]		37	34	10.00	0.38	7619
mouclass.c	17372	57	46	54.46	3.34	
parport.c	61781	193	50	1980.09	519.69	102967

Table 1. Verification times with BLAST. A blank proof size indicates a bug was found.

are computed only to the necessary precision, and recomputed only in parts of the state space where the abstraction changes. Large parts of the state space that are known to be error free are not searched again. This also helps in producing compact proofs.

3. *Cartesian post.* Our experiments show that we can replace the most precise but computationally expensive abstract-postcondition computation [8] by an imprecise one [1, 15], and still prove all properties of interest. This speeds up the computation significantly.[4]

While our running times and proof sizes are encouraging, we feel there is a lot of room for improvement. We are transitioning to an oracle-based representation of proofs [19], which we expect, based on previous experience, to further reduce the size of the proofs by an order of magnitude. The times taken by the counterexample analysis often dominates the verification time. We believe that a faster theorem prover and optimization techniques from program analysis will be useful in speeding up the process.

Acknowledgments. We thank Tom Ball and Sriram Rajamani for many fruitful discussions and for the IRP completion specification. We thank Microsoft Research for hosting R. Majumdar and W. Weimer as summer interns, which gave them the opportunity to learn about SLAM.

References

1. T. Ball, R. Majumdar, T. Millstein, and S.K. Rajamani. Automatic predicate abstraction of C programs. *Conf. Programming Language Design and Implementation*, pp. 203–213. ACM, 2001.

[4] For the Windows examples, [8] failed to finish even after 24 hours.

2. T. Ball and S.K. Rajamani. Personal communication.
3. T. Ball and S.K. Rajamani. The SLAM project: debugging system software via static analysis. *Symp. Principles of Programming Languages*, pp. 1–3. ACM, 2002.
4. D. Blei, C. Harrelson, R. Jhala, R. Majumdar, G.C. Necula, S.P. Rahul, W. Weimer, and D. Weitz. *Vampyre: A Proof-generating Theorem Prover.* http://www.eecs.berkeley.edu/˜rupak/Vampyre.
5. A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating system bugs. *Symp. Operating System Principles*, pp. 78–81. ACM, 2001.
6. E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. *Computer-Aided Verification*, LNCS 1855, pp. 154–169. Springer-Verlag, 2000.
7. J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera: extracting finite-state models from Java source code. *Int. Conf. Software Engineering*, pp. 439–448. ACM, 2000.
8. S. Das, D. L. Dill, and S. Park. Experience with predicate abstraction. *Computer-Aided Verification*, LNCS 1633, pp. 160–171. Springer-Verlag, 1999.
9. D. Detlefs, G. Nelson, and J. Saxe. *The Simplify Theorem Prover.* http://research.compaq.com/SRC/esc/Simplify.html.
10. E. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.
11. M.D. Ernst. *Dynamically Discovering Likely Program Invariants.* Ph.D. Thesis. University of Washington, Seattle, 2000.
12. J.S. Foster, T. Terauchi, and A. Aiken. Flow sensitive type qualifiers. *Conf. Programming Languages Design and Implementation* (to appear), ACM, 2002.
13. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. *Computer-Aided Verification*, LNCS 1254, pp. 72–83. Springer-Verlag, 1997.
14. R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the ACM*, 40:143–184, 1993.
15. T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. *Symp. Principles of Programming Languages*, pp. 58–70. ACM, 2002.
16. G. Holzmann. Logic verification of ANSI-C code with Spin. *SPIN Workshop*, LNCS 1885, pp. 131–147. Springer-Verlag, 2000.
17. K. Namjoshi. Certifying model checkers. *Computer-Aided Verification*, LNCS 2102, pp. 2–13. Springer-Verlag, 2001.
18. G.C. Necula. Proof carrying code. *Symp. Principles of Programming Languages*, pp. 106–119. ACM, 1997.
19. G. Necula and S.P. Rahul. Oracle-based checking of untrusted software. *Symp. Principles of Programming Languages*, pp. 142–154. ACM, 2001.
20. G.C. Necula and P. Lee. Efficient representation and validation of proofs. *Symp. Logic in Computer Science*, pp. 93–104. IEEE Computer Society, 1998.
21. G.C. Necula, S. McPeak, S.P. Rahul, and W. Weimer. CIL: intermediate language and tools for analysis and transformation of C programs. *Compiler Construction*, LNCS 2304, pp. 213–228. Springer-Verlag, 2002.
22. D. Peled and L. Zuck. From model checking to a temporal proof. *SPIN Workshop*, LNCS 2057, pp. 1–14. Springer-Verlag, 2001.
23. F. Pfenning. *Computation and Deduction.* Lecture notes, CMU, 1997.
24. F. Somenzi. *Colorado University Decision Diagram Package.* http://vlsi.colorado.edu/pub.
25. W. Visser, K. Havelund, G. Brat, and S. Park. Model Checking Programs. *Conf. Automated Software Engineering*, pp. 3-12. IEEE, 2000.