Lazy Annotation for Program Testing and Verification

K. L. McMillan

Cadence Berkeley Labs

Abstract. We describe an interpolant-based approach to test generation and model checking for sequential programs. The method generates Floyd/Hoare style annotations of the program on demand, as a result of failure to achieve goals, in a manner analogous to conflict clause learning in a DPLL style SAT solver.

1 Introduction

The DPLL approach to Boolean satisfiability combines search and deduction in a mutually reinforcing way. It focuses deduction where the search becomes blocked, deducing facts that guide the search away from the failure. Here, we consider an approach to program testing and verification inspired by DPLL. As in DART [3], we use symbolic execution to search the space of execution paths. When the search fails to reach a goal, we deduce a program annotation that will prevent us from being blocked in the same way in the future. Since annotations are deduced only in response to search failures, we will call this method *lazy annotation*.

The algorithm proceeds roughly as follows. We designate a set of program locations as goals to be reached. In the following examples, the goals are calls to a function `error`. The vertices (locations) and edges (statements) of the program's control flow graph will be labeled with formulas. A label represents a condition under which no goal can be reached. Initially, there are no labels (no annotation being equivalent to false). We execute the program *symbolically* along some chosen path. Each input to the program is represented by a symbolic value p_i. In the symbolic state, each program variable is evaluated as a symbolic expression over these parameters. A *constraint* is also maintained, which accumulates the conjunction of the branch guards along the chosen path, as a function of the parameters.

We say that our state is *blocked* if the current vertex label is implied, meaning we cannot reach a goal from this state. When we are blocked, we will backtrack along the edge we just executed, annotating it with a new label that blocks that edge. When choosing a branch to execute, we are guided by these edge labels. A blocked edge cannot lead to a goal, so we always continue along an unblocked edge if there is one. When all outgoing edges are blocked in the current state, we label the current location with the conjunction of the conditions that block the outgoing edges, thus blocking the current state and causing us to backtrack.

As an example, consider the fragment `simple` in Figure 1. Suppose on entering this code at l_1, our symbolic state is $x = p_0$ with constraint T (true). Branching from l_1 to l_2, we obtain the constraint $p_0 = 0$. At l_2 we have the choice to branch to l_3 or l_6. Since neither edge is labeled, we choose arbitrarily l_6. Now there is only one choice and we branch to l_7, obtaining the unsatisfiable constraint $p_0 = 0 \wedge p_0 < 0$. At this point we are blocked, since F (false) holds in the current state and absence of annotation is equivalent to F.

We therefore backtrack, annotating edge $l_6 \rightarrow l_7$. In general, if the current annotation is ϕ, we can annotate the incoming edge with the weakest precondition of ϕ along that edge. In this case, the current annotation is $\phi = F$ and the weakest precondition is $x \geq 0$. After backtracking, we return to l_6. There is only outgoing edge, labeled $x \geq 0$. Therefore, we can label $l_6 : x \geq 0$. We are now blocked (since $x = 0$ in our current state) so we backtrack to l_2, labeling edge $l_2 \rightarrow l_6 : x \geq 0$ (again using the wp). Notice that each time we label an edge, that edge becomes blocked in the current symbolic state (and possibly other states).

Since the edge $l_2 \rightarrow l_3$, is still unblocked, we follow it. Our annotation has forced the search in a different direction. Moving on to l_4, we have $y = p_1$ (a new input), then at l_5 we have the constraint $p_1 \geq 0$ (corresponding to the guard $y \geq 0$). Finally we arrive at l_6 in the state $x = p_0 + p_1$ with constraints $p_0 = 0$ and $p_1 \geq 0$. Since this implies the previous annotation $l_6 : x \geq 0$, we are blocked. The fact we previously learned tells us there is no path to the goal from our current state. Backtracking to l_5 and taking the weakest precondition of $x \geq 0$ then gives us $l_5 : x + y \geq 0$. When we backtrack to l_4, however, we observe a slight problem. The weakest (liberal) precondition of $x + y \geq 0$ with respect to the assertion $y \geq 0$ is $y < 0 \vee x + y \geq 0$. However, the variable y is irrelevant here, and we could just as well block the state with $x \geq 0$ (also a precondition, not the weakest). This latter fact can be computed as an *interpolant* as we will show later. The advantage of the interpolant becomes clear in the next step when the weakest precondition would yield $l_3 : \forall y.\ (y < 0 \vee x + y \geq 0)$. We can simplify this to $x \geq 0$, but this requires quantifier elimination, which can be very expensive. By computing preconditions with interpolants, we avoid the need for quantifier elimination.

Now we backtrack to l_2, labeling the edge $(l_2 \rightarrow l_3)$ with $x \geq 0$. At this point, both edges from l_2 are blocked, so we label it with the conjunction of the blocking labels along these edges, yielding $l_2 : x \geq 0$. Finally, we label $l_1 : T$, proving that the goal `error` cannot be reached from l_1.

Lazy annotation has the advantage that, like DPLL, it can recover from irrelevant or too-specific deductions. Consider, for example, fragment `diamond`, and suppose we first execute the path through l_3 and l_7. After backtracking, we label $l_6 \rightarrow l_7 : p$. Then executing edge $l_6 \rightarrow l_9$ we are blocked, because $a = 1$ in the current state. Thus, rather than exploring this path and discovering it is also safe because p holds, we label $l_6 \rightarrow l_9$ with the irrelevant condition a, and thus we label $l_6 : p \wedge a$. However, when we return to l_6 via l_5, note what happens. The edge $l_6 \rightarrow l_7$ is blocked by label p so we follow $l_6 \rightarrow l_9$, ultimately labeling it p

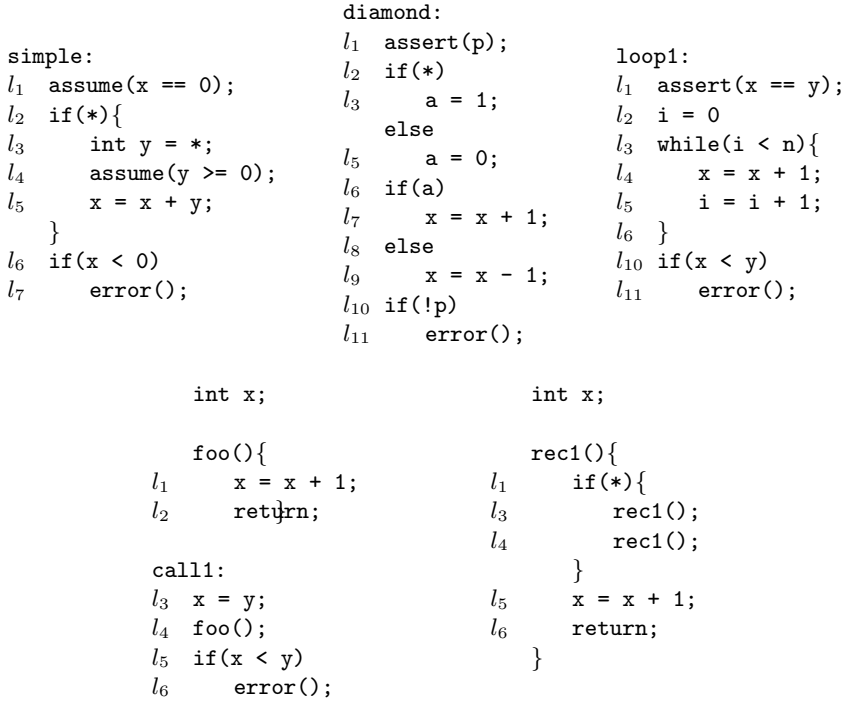


Fig. 1. Example program fragments

as well. Thus, we label $l_6 : p$. Location l_6 is now labeled by the *disjunction* of p and $p \wedge a$. The stronger condition $p \wedge a$ is effectively subsumed, and we discard it.

Note how this differs from partition refinement approaches such as Synergy [5] and its descendants [4]. In Synergy, after executing the path through l_7, location l_9 is on the frontier. This causes l_6 to be partitioned by the irrelevant predicate a. There is no way to recover from this irrelevant refinement. With sequences of such diamonds, we can construct reasonable scenarios in which this leads Synergy to an exponential explosion of partitions, while lazy annotation is polynomial.

Unbounded loops Consider the simple unbounded loop in fragment `loop1`. To force this loop to terminate, we will use a simple trick: we instrument the program with a variable τ that decreases in every iteration, and we annotate all locations with $\tau < 0$, blocking any path in which τ becomes negative. We pick an arbitrary initial value of τ, say, zero. This forces the loop to be executed at most once. Say we execute first the path that skips the loop. Using interpolants, we label $l_3 \rightarrow l_{10} : x \geq y$ and backtrack into the loop. When the loop completes, we decrement τ and are thus blocked by $l_3 : \tau < 0$. We thus backtrack through the loop. Using interpolants, we obtain $l_3 \rightarrow l_4 : \tau < 1$. Thus, we label $l_3 : x \geq$

$y \wedge \tau < 1$. Backtracking further, we label $l_1 : \tau < 1$ and terminate. We have not proved unreachability of `error`, since our annotation depends on the bogus label $\tau < 0$. However, our annotation can be strengthened by induction. We simply plug $\tau = 0$ into our labels and see if they are inductive. We obtain $l_3 : x \geq y$ which is in fact inductive, so we keep it. We drop any non-inductive labels iteratively, obtaining the greatest inductive subset as a fixed point. Effectively we have used bounded model checking as a heuristic for constructing an inductive invariant. Note that using weakest precondition, we would have obtained $l_3 : x \geq 0 \vee i < n$ which is not inductive. To handle unbounded loops, we need some form of generalization, here provided by interpolation. If strengthening by induction fails to prove unreachability, we can increase the initial τ value and try again.

Procedure summaries To handle programs with procedures modularly, we label them with negative summaries. This is a formula that uses primed variables to represent the exit state of the procedure, and is true when that exit state is *not* reachable. For example, if on exit the value of x must be greater than its current value, the negative summary would be $x' \leq x$. We can use lazy annotation to compute a summary for a procedure from a given initial state, with a desired post-condition ψ. This reusable summary can replace a call to the procedure in various contexts. Consider fragment `call1` in the figure. Here, we start at l_3 in state $x = p_0, y = p_1$. When we reach the call to `foo` at l_4, we recursively call lazy annotation to compute a summary of `foo` that proves the current post-condition of the call, which is $l_5 : F$. Obviously, this cannot be proved, and we obtain a counterexample, which is a path to the return state $x = p_1 + 1, y = p_1$. Continuing from l_5, we eventually label $l_5 : y \leq x$. Now when we backtrack to l_4, we recursively try to compute a summary of `foo` that proves the post-condition $y \leq x$. This time we succeed, computing the negative summary $l_1 : x' < x$ as an interpolant (see Section 2.3). Using this summary for $l_4 \rightarrow l_5$, we can label $l_4 : x \geq y$ and terminate. Moreover this same summary at l_1 may be useful in other contexts, allowing us to return immediately from `foo`.

The advantage of using interpolants to compute summaries is that we can obtain more general summaries. A method such as Smash [4] that uses weakest precondition to compute a summary with the post-condition $x \geq y$ would yield a summary such as $x \geq y \overset{\text{foo}}{\Rightarrow} x \geq y$ containing the irrelevant variable y. To be able to reuse this summary in another context, we need to be able to universally quantify over y, which again involves us in quantifier elimination. Using interpolants, this complication is avoided.

Recursion Finally, consider the recursive function `rec1` in the figure and suppose we want to compute a summary for initial state $x = 0$ and post-condition $x \geq 0$. To force termination, we decrement τ on recursive calls, and initialize τ to 1. Now suppose we first take the path through l_3. Because τ is decremented, the recursive calls at l_3 and l_4 must take the non-recursive path, yielding an exit state $x = 2$. This satisfies the post-condition $x \geq 0$, giving a negative

summary $l_5 : x' < x$. Backtracking to l_4, we again call recursively with a post-condition equivalent to $x \geq 0$ (for details, see Section 2.3) which yields a summary $l_1 : x' < x \wedge \tau < 1$. The same summary is reused in backtracking to l_3, which eventually gives us $l_1 : x' < x \wedge \tau < 2$. Setting $\tau = 0$, we find that $l_1 : x' < x$ is inductive and terminate.

Related work Lazy annotation is similar to lazy abstraction with interpolants [8] in that it computes an inductive invariant using only interpolation. Because it explores only feasible paths, however, it is useful for testing, and can efficiently handle bounded loops. Moreover, it handles procedures modularly. It is also similar in some respects to Synergy [5] and related methods [4] that use partition refinement. However, unlike these methods, it can recover from too-specific refinements. In fact, we can think of the annotation as partitioning each location into exactly two state sets. Lazy annotation can also compute more general summaries using interpolants.

Relative to predicate abstraction approaches [1, 9], lazy annotation has the advantage that it avoids the expensive predicate image computation. However it is in another sense orthogonal to these methods, as predicate abstraction can be used to inductively strengthen the annotations obtained by lazy annotation, possibly speeding convergence for unbounded loops, while avoiding the many iterations produced by counter-example guided abstraction refinement. In fact, any backward abstract interpretation can be used for this purpose.

2 Lazy annotation

Throughout this paper, we will use standard first-order logic (FOL) and the notation $\mathcal{L}(\Sigma)$ to denote the set of well-formed formulas (*wff's*) of FOL over a vocabulary Σ of uninterpreted symbols (the formulas may also include various interpreted symbols, such as $=$ and $+$). For a given formula ϕ, $\mathcal{L}(\phi)$ will denote the *wff's* over the uninterpreted vocabulary of ϕ. We will write $\phi[\sigma]$ to indicate that structure σ is a model of formula ϕ. To every uninterpreted symbol s, we associate a unique symbol s' (that is, s with one prime added). For any formula or term ϕ or vocabulary S, we will write ϕ' or S' for the result of adding one prime to all the non-logical symbols in ϕ or S.

Given a pair of FOL formulas (A, B), such that $A \wedge B$ is inconsistent, an *interpolant* for (A, B) is a formula $\hat{A}$ such that A implies $\hat{A}$, $\hat{A}$ implies $\neg B$, and $\hat{A} \in \mathcal{L}(A) \cap \mathcal{L}(B)$. The Craig interpolation lemma [2] states that interpolants always exist for inconsistent formulas in FOL. A variety of techniques exist for deriving an interpolant for (A, B) from refutation of $A \wedge B$ in a suitable proof system [7]. This allows us to generate interpolants using a theorem prover or proof-generating decision procedure.

Modeling programs We assume a vocabulary S of variables representing the program's data state, a domain D of data values, and a collection of program

actions $\mathcal{A}$ provided by the programming language. A *program* is a finite, rooted, labeled graph (Λ, l_0, Δ) where Λ is a finite set of program locations, $l_0 \in \Lambda$ is a distinguished initial location and $\Delta \subseteq \Lambda \times \mathcal{A} \times \Lambda$ is a set of transitions labeled by actions. Let $\mathrm{Out}(l)$ denote the set of outgoing edges from location l.

A *program path* of length k is an alternating sequence of the form $l_0 a_0 l_1 a_1 \ldots l_k$, where each triple (l_i, a_i, l_{i+1}) is in Δ. A *data state* in $\mathcal{D}$ is a map $S \to D$. We fix an initial data state d_0. The semantics of an action $a \in \mathcal{A}$, denoted $\mathrm{Sem}(a)$, is subset of $\mathcal{D} \times \mathcal{D}$. A *program run* of length k is a pair (π, σ), where π is a program path, and $\sigma = d_0 \cdots d_k$ is a sequence of data states such that for all $0 \leq i < k$, $(d_i, d_{i+1}) \in \mathrm{Sem}(a_i)$. A *state* is a pair $(l, d) \in \Lambda \times \mathcal{D}$. The reachable states are all the pairs (l_k, d_k) for some run of length k.

A *state formula* is a formula in $\mathcal{L}(S)$. A *transition formula* is a formula in $\mathcal{L}(S \cup S')$. For action a and formulas ϕ, ψ (that may contain non-program variables) the *Hoare triple* $\{\phi\}a\{\psi\}$ is valid when for every data state d_1, and interpretation $\mathcal{I}$ of the non-program variables, such that $\phi[d_1 \cup \mathcal{I}]$ and every d_2 such that $(d_1, d_2) \in \mathrm{Sem}(a)$, we have $\psi[d_2 \cup \mathcal{I}]$. We assume that $\mathrm{Sem}(a)$ can be expressed as a transition formula, which by abuse of notation, we will write $\mathrm{Sem}(a)$. Since actions and transition formulas are interchangeable, we will also write $\{\phi\}t\{\psi\}$ where t is an arbitrary transition formula.

It is useful to define a *relational join* operator for relations expressed as formulas. Let ϕ and ψ be formulas, and f be an indexed set of variables with a unique variable f_v associated to each $v \in S$. Then $\phi \times_f \psi$ is the formula $\phi\langle f_v / v' \rangle \wedge \psi\langle f_v / v \rangle$. If ϕ and ψ are transition formulas, we can think of this formula as representing a succession of two transitions, the first satisfying ϕ and the second satisfying ψ, with f representing the intermediate state. If we omit the subscript f, then the intention is that f is some set of variables not previously used. One important fact we will use is that $\{\phi\}t\{\psi\}$ is valid exactly when $\phi \wedge (t \times \neg\psi)$ is unsatisfiable.

Symbolic Interpreters A *symbolic data state* represents a set of data states parametrically. The symbolic data states $\mathcal{S}$ consist of the triples (P, C, E), where P is a set of parameters (variables not in S), $C \in \mathcal{L}(P)$ is a constraint over the parameters, and E is a map from the program variables S to functions over P. We assume the these functions are expressible as first order terms over P. Thus, a symbolic state s can be characterized by the predicate $\chi(s) = C \wedge \bigwedge_{v \in S} v = E(v)$. A symbolic data state s represents a set of data states $\gamma(s)$ defined as follows:

$$\gamma(s) = \{d \in \mathcal{D} \mid d \models \exists P.\ \chi(s)\}$$

This is the set of data states produced by the map E for some valuation of the parameters satisfying the constraint C. We assume a defined initial symbolic data state s_o such that $\gamma(s_0) = \{d_o\}$. A (full) *symbolic state* is a pair $(l, s) \in \Lambda \times \mathcal{S}$.

A *symbolic interpreter* SI maps $\mathcal{A}$ to $\mathcal{S} \times \mathcal{S}$. We require that $\mathrm{SI}(a)$ is total for all actions a. Intuitively, a symbolic interpreter takes a symbolic state and an action, and returns a non-empty set of symbolic states representing the effect

of executing a. Symbolic interpreter SI is *sound* when, for all symbolic states s and actions a,

$$\cup\gamma(\mathrm{SI}(a)(s)) = \mathrm{Sem}(a)(\gamma(s))$$

That is, the symbolic successors of s must together represent exactly the successors of the concrete states represented by s. Note that the set of states represented can be empty, since the constraint in a symbolic state can be F.

Note that $\mathrm{SI}(a)$ may be a function (i.e., deterministic). In this case, non-determinism in a is modeled by the introduction of parameters. On the other hand, we may decide based on heuristic considerations to replace parameters with concrete values, introducing non-determinism in $\mathrm{SI}(a)$. Injecting concrete values in this way is analogous to decision making in DPLL. Soundness is not sacrificed as long as SI is sound. As in DART, however, it is also possible to substitute concrete values in an unsound way for operations which cannot be modeled symbolically [3].

2.1 Intraprocedural algorithm

We first consider the case without procedure calls. We define a set of goals $G_0 \subset \Lambda$ that we wish to reach. For each goal, we wish to find a concrete run reaching the goal, or a proof that it is unreachable. The state of the algorithm is a triple (Q, A, G), where Q is a query set, A is a program annotation, and $G \subseteq \Lambda$ is the set of remaining goals. A *query* is a pair (s, ψ), where s is a symbolic state called the initial state, and ψ is a formula called the post-condition of the query. In the intraprocedural setting, the post-condition serves no purpose. It will be used later when computing procedure summaries.

An *annotation* is a set of pairs in $(\Lambda \cup \Delta) \times \mathcal{L}(S)$. We will notate these pairs in the form $l : \phi$ or $e : \phi$, where l is a location, e an edge and ϕ a formula called the *label*. The intended semantics is that no path beginning with location l or edge e can reach any remaining goal *if ϕ is initially true*. We will write $A(l)$ for $\vee\{\phi \mid l : \phi \in A\}$.

For an edge $e = (l_1, a, l_2)$, we say that a label $e : \phi$ is *justified* in A when $\{\phi\}\ a\ \{A(l_2)\}$, that is, when it implies the annotation of l_2 after executing a. We notate this condition $\mathcal{J}(e : \phi, A)$. For a location l, we will say that a label $l : \phi$ is justified in A when, for all edges $e \in \mathrm{Out}(l)$, there exists $e : \psi \in A$ such that $\phi \Rightarrow \psi$. An annotation is justified when all its elements are justified. A justified annotation is inductive. If it is also initially true, then it is an inductive invariant. Our algorithm maintains the invariant that A is always justified.

We will say that a query $q = ((l, s), \psi)$ is *blocked* by formula ϕ, when $s \models \phi$ and write $\mathcal{B}(q, \phi)$. With respect to q, the edge e is blocked when $\mathcal{B}(q, A(e))$, and the location l is blocked when $\mathcal{B}(q, A(l))$.

The algorithm IntraLA proceeds according to the transition rules defined in Figure 2. The initialization rule Init sets the algorithm state to $Q = \{((l_0, s_0), \psi_0)\}$, $A = A_0 = \emptyset$, $G = G_0$. That is, we are at the program's initial state, with no

$$\textsc{Init}\frac{}{\{((l_0, s_0), \psi_0\},\ A_0,\ G_0}$$

$$\textsc{Decide}\frac{Q,\ A,\ G}{Q + ((l_2, s_2), \psi),\ A,\ G} \quad \begin{array}{l} q = ((l_1, s_1), \psi) \in Q \\ e = (l_1, a, l_2) \in \Delta \\ \neg\mathcal{B}(q, A(e)) \\ s_2 \in \mathrm{SI}(a)(s_1) \\ \neg\mathcal{B}(((l_2, s_2), \psi), A(l_2)) \end{array}$$

$$\textsc{Conjoin}\frac{Q, A, G}{Q - q,\ A + l : \phi,\ G - l} \quad \begin{array}{l} q = ((l, s), \psi) \in Q \\ \neg\mathcal{B}(q, A(l)) \\ \forall e \in \mathrm{Out}(l), e : \phi_e \in A \wedge \mathcal{B}(q, \phi_e) \\ \phi = \wedge\{\phi_e \mid e \in \mathrm{Out}(l)\} \end{array}$$

$$\textsc{Learn}\frac{Q, A, G}{Q,\ A + e : \phi,\ G} \quad \begin{array}{l} q = ((l_1, s_1), \psi) \in Q \\ e = (l_1, a, l_2) \in \Delta \\ \mathcal{B}(q, \phi) \\ \mathcal{J}(e : \phi, A) \end{array}$$

Fig. 2. Algorithm IntraLA

locations labeled, and all goals yet to be reached. The decision rule Decide generates a new query from an existing one by symbolically executing one program action. It may choose any edge that is not blocked, and any symbolic successor state generated by the action a. If the newly generated query is itself not blocked, it is added to the query set.

If all of the outgoing edges of a query are blocked, the Conjoin rule is used to block the query by labeling its location with the conjunction of the labels that block the outgoing edges. At this point, we know that the symbolic state is not empty (since otherwise the query would already be blocked). Thus, if the location is a goal, we have reached the goal, and we remove it from the set of remaining goals. The blocked query is discarded.

The remaining case is that some outgoing edge $e = (l_1, a, l_2)$ is not blocked, but every possible symbolic step along that edge leads to a blocked state. In this case, the Learn rule infers a new label ϕ that blocks the edge. The new edge label can be any formula ϕ that both blocks the current query and is justified. Such a formula can be obtained as an interpolant for (A, B), where $A = \chi(s_1)$ and $B = \mathrm{Sem}(a) \wedge \neg A(l_2)'$. Thus we can derive ϕ, if it exists, using an interpolating theorem prover [7].

The algorithm maintains the invariant that no queries are blocked, and, for every $l \in G$, $A(l) = F$. It terminates when no rules can be applied, which implies the query set is empty.

Theorem 1. *When algorithm* IntraLA *terminates, all the locations in* $G_0 \setminus G$ *are reachable and all the locations in* G *are unreachable.*

Proof sketch. All the rules preserve the invariant that A is justified (therefore inductive), that all the locations in G are unlabeled (meaning their annotation is equivalent to F) and that no queries are blocked. Now suppose the algorithm is in a state where no rules can be applied and consider some $q \in Q$. Since Decide does not apply, all possible successor queries are blocked. Thus, since Learn does not apply, all outgoing edges are blocked. Thus, since Conjoin does not apply, q is blocked, a contradiction. Since Q is empty, it follows that the initial query is blocked, meaning that $d_0 \models A(l_0)$. Therefore, A is an inductive invariant. Since all remaining goals are annotated F, it follows that they are unreachable. Moreover, since goals are only removed from G when reached, all locations in $G_0 \setminus G$ are reachable. □

Of course, we can also terminate the algorithm immediately if the set of remaining goals becomes empty.

2.2 Handling unbounded executions

The approach described above has one clear drawback: if the program has any loop that can execute unboundedly, then the algorithm will not terminate. That is, for any learning to occur, we must first reach a blocked state. However, if there is an unbounded loop, we can keep extending the run infinitely without reaching a blocked state.

To deal with this situation, we use the following generic approach. We introduce an auxiliary variable τ to the program. This variable must be non-increasing and infinitely often decreasing according to some pre-order that is well-founded over a domain characterized by some predicate W. For example, τ could be an integer that is decremented by every program action and the domain predicate could be $\tau >= 0$. We can think of τ as the program's "time to live". Alternatively, it would be sufficient to decrement τ on each back-edge of the program graph, so that τ has to be decremented at least once on each cycle. Or, τ could be a vector with one element for each SCC of the graph.

Now fix an initial value τ_0 of τ, and label every location l with the predicate $\neg W$. With this construction, every infinite run is eventually blocked. Thus, algorithm IntraLA is guaranteed to terminate (at least if the symbolic interpreter SI is finitely non-deterministic). When termination occurs, the annotation A is a proof that the remaining goals cannot be reached for the particular initial value τ_0. This is, in effect, a form of bounded model checking.

To obtain an unbounded proof, we can use the heuristic that bounded proofs may contain the ingredients of unbounded proofs. To do this, we will eliminate the dependence on τ in the annotation A, resulting in an unbounded annotation A_U. This can be done, for example by substituting some fixed value $\bot$ for τ, typically the bottom value of the pre-order.

The unbounded annotation A_U is not necessarily justified. However, we can make it justified by iteratively dropping labels that are not justified until a fixed

Algorithm StrengthenIntraLA
Input: a query q and goal set G_0
Output: set of unreachable goals

$Q \leftarrow \{q\}$, $A \leftarrow \emptyset$, $G \leftarrow G_0$
while T **do**
    Run IntraLA on (q, A, G) to termination
    $A_U \leftarrow \{l : \phi\langle\bot/\tau\rangle \mid l : \phi \in A\}$
    while exists $l : \phi \in A_U$ s. t. $\neg\mathcal{J}(l : \phi, A)$ **do**
        $A_U \leftarrow A_U - l : \phi$
    done
    if $\mathcal{B}(q, A_U(l_0))$ return G
    $A \leftarrow A \cup A_U$
    increase τ_0
done

Fig. 3. Algorithm with inductive strengthening

point is reached. The result is the greatest inductive subset of A_U. This set of unbounded facts can then be used to strengthen A. If the resulting annotation is true in the initial state for all values of τ, then we have proved unreachability of the remaining goals. Otherwise, we increase τ_0 and repeat algorithm IntraLA. This overall procedure is depicted in Figure 3.

Because the problem of determining the reachable goals is undecidable, we do not expect this algorithm to terminate in all cases. The hope is that the computed interpolants will converge to inductive assertions after a small number of iterations of the loops, and thus τ_0 will not become large. An alternative inductive strengthening approach would be to apply predicate abstraction using the atomic predicates occurring in A_U. Though the cost would be higher, the chance of convergence might be better.

We must also take care to handle loops with large fixed bounds efficiently. That is, suppose we have a loop that iterates N times where N is a large fixed number. If we increment τ_0 by one, then we may increment τ_0 N times before exiting the loop, resulting in $O(N^2)$ decision steps. One simple way to prevent this would be to double τ_0 instead of incrementing it, which would give $O(N \log N)$ steps. Alternatively, if we can determine statically that the loop is bounded, we can simply remove the decrements of τ from the loop, without causing nontermination of IntraLA.

2.3 Interprocedural algorithm

To handle procedures in a modular fashion, we will annotate the program with *negative summaries*. This is a transition formula ϕ with the intended meaning that if $\phi[d_0, d_1]$ holds, then entry to the procedure in data state d_0 may *not* result in exit in state d_1. Negative summaries are used because they are inductive in the normal, forward sense.

To detect goals reached within procedures, we designate a special variable f. On reaching a goal, a procedure *aborts*, that is, it exits immediately with f true. On normal exit, f is false. Given a negative summary ϕ, we can think of $\phi\langle T/f'\rangle$ as a pre-condition under which the procedure guarantees not to reach a goal and abort.

Modeling programs with procedures To model data in programs with procedures we designate a set of global variables G and local variables L. For $i = 0, 1, \ldots$ the *frame* L_i consists of a variable v_i for each $v \in L$. A frame represents the local state of one procedure instance, with L_0 representing the current procedure instance.

To model procedure calls, we introduce special actions $\text{call}(l)$, where l is a location of a procedure to be called. We introduce a set of procedure call edges Θ that are distinct from the ordinary edges Δ. We also designate a set $\Omega \subset \Lambda$ of *exit states*. Due to space considerations, we give the semantics of calls only informally. The effect of an edge $(l_1, \text{call}(l_2), l_3)$ is to transfer control to l_2, execute until reaching an exit state, then return by transferring control to l_3. The effect of a call action on the data state is to "push" one frame on the "stack". We define an operation push on data states that shifts the local variables up by one frame, so that $\text{push}(d)(v_{i+1}) = d(v_i)$. On return, one stack frame is "popped". We define $\text{pop}(d)$ so that $\text{pop}(d)(v_i) = d(v_{i+1})$. We also define corresponding renaming operators on formulas, so that $\text{push}(\phi) = \phi\langle L_{i+1}/L_i\rangle$ and $\text{pop}(\phi) = \phi\langle L_{i-1}/L_i\rangle$.

Computing summaries We will say a query $q = ((l, s), \psi)$ is *blocked* by a negative summary ϕ, that is, $\mathcal{B}(q, \phi)$, when every exit allowed by ϕ satisfies the post-condition ψ. Because summaries are negative, this is equivalent to saying that $\{\chi(s)\}\ \neg\phi\ \{\psi\}$ or that the formula $\chi(s) \wedge (\neg\phi \times \neg\psi)$ is unsatisfiable.

Justification of ordinary edges and locations remains as before. Now, however, we say that ϕ is justified at an *exit state* l when $\phi = \neg I_S \vee f'$, where I_S is the identity relation over the data variables S. In other words, exiting from a procedure leaves the data unchanged, and does *not* abort. We will consider justification of call edges shortly.

Using algorithm IntraLA, we can define a function Sum (see Figure 4) that constructs a negative summary for a procedure. It takes a query $((l_1, d_1), \psi)$ and returns either a reachable exit state d_2 that does *not* satisfy the post-condition ψ (*i.e.* a counterexample to the query) or it labels l_1 with a summary that ensures that no such counterexample exists.

Using procedure summaries We use procedure summaries to justify the annotation of call edges. Intuitively, a summary ϕ is justified at a call edge $e = (l_1, \text{call}(l_2), l_3)$ if it is justified by considering the summary of the called procedure l_2 as an action. There are two subtleties involved in this, however. The first is that we must account for the shift in stack frames between the calling and called contexts. We can do this by applying the push operator to the formulas in the calling context. The second is that the calling context must abort if the called context aborts. We can effect this by weakening the summary at the return location. We define $\mathcal{W}(\phi) = \phi \wedge \neg(f \wedge f')$. In effect, this removes transitions from

Function $\textsc{Sum}(q_0, A, G)$

Apply IntraLA from initial state $(\{q_0\}, A, G)$ until
1) there exists $((l, d), \psi) \in Q$, where $l \in \Omega$:
in which case, return (d, A, G), *or*
2) Q is empty:
in which case, return (ϵ, A, G).

Fig. 4. Algorithm for procedure summary construction

the negative summary at the return site, allowing the calling context to abort when the called function does. Using this notion, the justification condition for procedure calls is defined as follows:

$$\mathcal{J}((l_1, \text{call}(l_2), l_3) : \phi, A) \quad \text{iff} \quad \{\text{push}(\phi)\} \;\; \neg A(l_2) \;\; \{\text{push}(\mathcal{W}(A(l_3))\}$$

We can then prove the following lemma:

Lemma 1. *If negative summary annotation A is justified, and if $d_0 \models A(l_0)\langle T/f'\rangle$ and if $A(l) = F$ for some location l, then location l is unreachable.*

Proof sketch. By induction on the length of runs, we show that if the initial state (l_0, d_0) of a run satisfies $A(l_0)\langle T/f'\rangle$ then every state (l_i, d_i) satisfies $A(l_i)\langle T/f'\rangle$. This property is preserved by call actions because we have weakened the summary at the return site, so that the caller aborts when the callee aborts. As a result, no reachable state can be labeled F. □

Now we are ready to define versions of the Decide and Learn rules for call edges. These are shown in Figure 5. In both cases, we have an outgoing call edge e from the current query. We compute a post-condition ψ_2 for the called function based on the current summary of the return site and the post-condition of the calling context. Notice we use the weakened summary to allow the called function to abort. Also notice that we apply push to the current state when entering the called function. If Sum returns an exit state, then the return site is not blocked, and DecideC generates a new query after the call (note pop is applied to this state). On the other hand, if Sum returns ϵ, then we are blocked. Thus LearnC can annotate the edge with a blocking formula ϕ.

A suitable condition ϕ that is both blocking and justified can be obtained as an interpolant for (A, B), where $A = \chi(s_1) \wedge \neg\psi_1'$ and $B = \neg\text{pop}(A(l_2)) \times \neg\mathcal{W}(A(l_3))$. That is, the transition formulas implied by A are exactly the negative summaries blocking the query q (note s_1 is the symbolic state of q and ψ_1 is the post-condition of q). The transition formulas inconsistent with B are exactly those justified at the call edge. Moreover, an interpolant for (A, B) must be a transition formula, since only variables in $S \cup S'$ can be common to A and B. Thus, any interpolant for (A, B) satisfies the conditions for the annotation ϕ of the call edge.

We will call the algorithm IntraLA with the addition of these two rules InterLA. The initial annotation A_0 consists of the labels $r : \neg I_S \vee f'$, for

$$\textsc{DecideC}\frac{Q,\ A,\ G}{Q + ((l_3, \mathrm{pop}(s_3)), \psi_1),\ A_3,\ G_3} \quad \begin{array}{l}(s_3, A_3, G_3) = \textsc{Sum}(q_2, A, G)\\ \neg\mathcal{B}(q, A(e))\end{array}$$

$$\textsc{LearnC}\frac{Q, A, G}{Q,\ A + e : \phi,\ G} \quad \begin{array}{l}(\epsilon, A_3, G_3) = \textsc{Sum}(q_2, A, G)\\ \mathcal{B}(q, \phi)\\ \mathcal{J}(e : \phi, A_3)\end{array}$$

$$\text{where} \quad \begin{cases} q = ((l_1, s_1), \psi_1) \in Q \\ e = (l_1, \mathrm{call}(l_2), l_3) \in \Theta \\ \psi_2 = \neg\mathcal{W}(A(l_3)) \times \psi_1 \\ q_2 = ((l_2, \mathrm{push}(s_1), \psi_2) \end{cases}$$

Fig. 5. Rules for InterLA (in addition to IntraLA)

all the exit locations $r \in \Omega$. The initial post-condition $\psi_0 = \neg f$. Essentially, this constructs a summary that proves that the program does not abort. This procedure is recursive. When it encounters a call edge, it calls Sum with a suitable query. This in turn runs InterLA on the called procedure. The recursive call can result in the addition of labels, and the elimination of reachable goals.

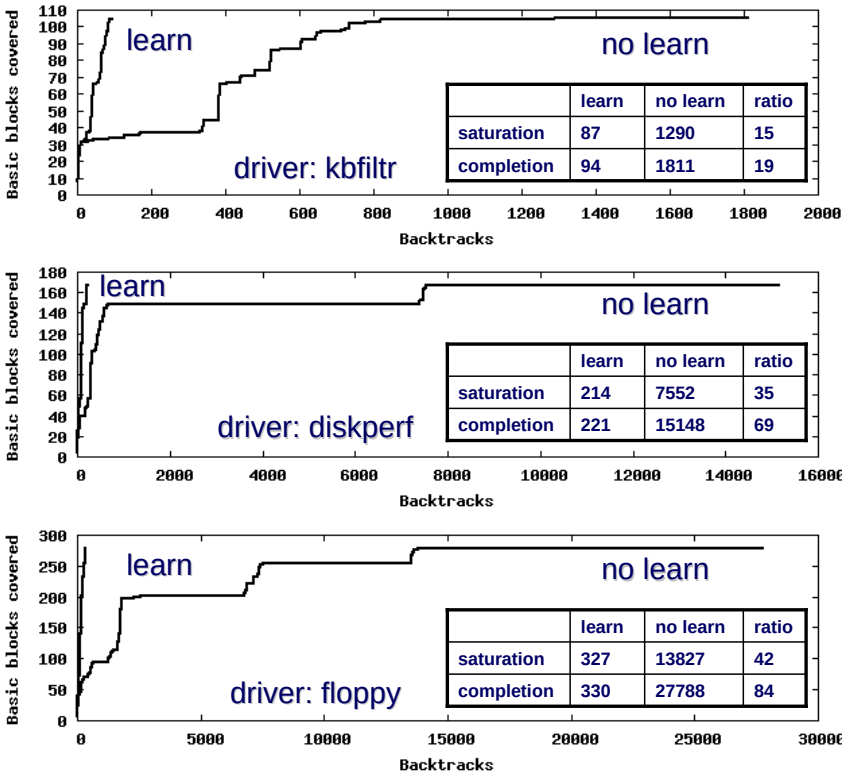
Theorem 2. *When algorithm* InterLA *terminates, all the locations in* $G_0 \setminus G$ *are reachable and all the locations in* G *are unreachable.*

Proof sketch. As in the intraprocedural case, all the rules preserve the invariant that A is justified and that the locations in G are unlabeled. When the initial query $((l_0, s_0), \neg f)$ is blocked, we know that $d_0 \models A(l_0)\langle T/f'\rangle$ therefore by the lemma, all unlabeled locations are unreachable. □

3 Implementation and Experiments

Algorithm InterLA has been implemented in a tool we will call Impact II. This tool uses the LLVM compiler infrastructure [6] to generate CFG's from C programs, with basic blocks corresponding to edges of the graph. The tool uses the FOCI interpolating prover [7] both for checking satisfiability of formulas and computing interpolants. The C heap is modeled using the theory of arrays. Static analysis is used to partition loads and stores into alias classes, with each alias class modeled by an array. External functions are modeled as having no side effects and returning a non-deterministic value, which can be considered an input to the program.

When a goal location is reached, Impact II extracts a satisfying assignment to the symbolic constraint C of the symbolic state as a test case. By initially marking every location as a goal, we can generate a set of tests that provides 100% coverage of reachable locations, and prove that the uncovered locations



	learn	no learn	ratio
saturation	87	1290	15
completion	94	1811	19

	learn	no learn	ratio
saturation	214	7552	35
completion	221	15148	69

	learn	no learn	ratio
saturation	327	13827	42
completion	330	27788	84

Fig. 6. Comparison of test generation with and without learning

are unreachable (since the compiler inlines small functions, some goals may be duplicated).

Experiments were conducted to test the hypothesis that this approach can produce a greater diversity of program behavior more quickly than methods such as DART that enumerate all execution paths. We used as benchmarks several device driver examples previously used as software model checking benchmarks [8]. We compare InterLA against an implementation of DART using the same symbolic interpreter and prover. This allows us to gauge the effect of learned annotations in guiding the search. Without learning, we simply enumerate all possible control paths. DART terminates in these tests because all the loops are bounded and there is no recursion.

Figure 6 plots the number of coverage goals reached as a function of the number of times the symbolic interpreter backtracked to an alternative program branch (which is also the number of test sequences generated). Three examples [1] are shown, with the number of reachable basic blocks ranging from 104 to 279. Each plot shows a line for InterLA ("learn") and a line for DART

[1] Source code available at `http://www.kenmcmil.com/benchmarks.html`.

("no learn"). The tables compare the number of backtracks needed for saturation (all reachable locations reached) and completion. Without learning, there are long plateaus during which many paths are explored but no new locations are reached. Learning clearly acts to push the search away from these regions, allowing the search to make steady progress. This effect is more pronounced in the larger program, with learning reducing backtracks to completion by a factor 84.

4 Conclusion

Lazy annotation allows us to deduce program annotations in response to search failure, much in the way that a DPLL SAT solver learns conflict clauses. As we have seen, this allows us to prune the search in test generation to achieve a given coverage goal at a greatly reduced cost. The method also has some potential advantages with respect to existing software model checking techniques. Since it is based entirely on interpolants, it avoids the expense of quantifier elimination or predicate image computations. The annotation approach avoids irreversible partitioning of the abstract state space, and potentially allows more general procedure summaries. Compared to lazy abstraction with interpolants, the method allows procedure summarization (thus may be more effective for deeply nested procedures) and handles bounded loops more effectively. In the other hand, it may be that the lazy abstraction approach of exploring infeasible program paths produces interpolants that are more relevant to the property being checked. Moreover, the question of how to obtain convergence in practice for unbounded loops needs further study.

References

1. T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *POPL*, pages 1–3, 2002.
2. W. Craig. Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. *J. Symbolic Logic*, 22(3):269–285, 1957.
3. P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *PLDI*, pages 213–223, 2005.
4. P. Godefroid, A. V. Nori, S. K. Rajamani, and S. D. Tetali. Compositional may-must program analysis: Unleashing the power of alternation. In *POPL*, pages 43–56, 2010.
5. B. Gulavani, T. A. Henzinger, Y. Kannan, A. Nori, and S. K. Rajamani. Synergy: A new algorithm for property checking. In *FSE*, pages 117–27, 2006.
6. C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88, 2004.
7. K. L. McMillan. An interpolating theorem prover. *Theor. Comput. Sci.*, 345(1):101–121, 2005.
8. K. L. McMillan. Lazy abstraction with interpolants. In *CAV*, pages 123–36, 2006.
9. R. Majumdar T. A. Henzinger, R. Jhala and G. Sutre. Lazy abstraction. In *POPL*, pages 58–70, 2002.