Testing, Abstraction, Theorem Proving: Better Together!

Greta Yorsh*
Tel Aviv University
Israel
gretay@post.tau.ac.il

Thomas Ball
Microsoft Research
Redmond, US
tball@microsoft.com

Mooly Sagiv
Tel Aviv University
Israel
msagiv@post.tau.ac.il

ABSTRACT

We present a method for static program analysis that leverages tests and concrete program executions. State abstractions generalize the set of program states obtained from concrete executions. A theorem prover then checks that the generalized set of concrete states covers all potential executions and satisfies additional safety properties. Our method finds the same potential errors as the most-precise abstract interpreter for a given abstraction and is potentially more efficient. Additionally, it provides a new way to tune the performance of the analysis by alternating between concrete execution and theorem proving. We have implemented our technique in a prototype for checking properties of C# programs.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing; D.2.4 [**Software Engineering**]: Software/Program Verification—*abstract interpretation*

General Terms

reliability, verification

Keywords

program analysis, testing, abstraction, theorem prover, abstract interpretation, software fault injection, fabricated states, adequacy criteria, coverage, state-based coverage

1. INTRODUCTION

Recently, there has been much interest in combining dynamic and static methods for analyzing programs [28, 16, 8, 29]. Dynamic analysis (or testing) is based on concrete program executions and *underapproximates* the set of program behaviors. That is, if B_P denotes the set of all behaviors of a program P then dynamic analysis explores a finite subset of B_P. Static analysis is based on the abstract interpretation [6] of program behavior and typically *overapproximates* the set of program behaviors. That is, static analysis has the effect of analyzing a superset of B_P, which may include *infeasible* behaviors that cannot be exhibited by the program.

The pros and cons of the two techniques are clear. If dynamic analysis detects an error then the error is real. However, dynamic analysis cannot provide a proof of the absence of errors. On the other hand, if static analysis does not find an error (of a particular kind) in the superset of B_P then B_P clearly cannot contain an error (of that same kind). However, if static analysis detects an error, it may be a false error as the behavior that induces the error may lie outside B_P.

We show how to perform static analysis using a novel combination of dynamic analysis, abstraction, and an automated theorem prover. Our technique is oriented towards finding a proof rather than detecting real errors. As a result, it has the pros and cons of a static analysis, but leverages dynamic analysis as its execution vehicle.

Our method uses state abstractions to generalize the set of program states gathered by monitoring concrete executions of a program P. An automated theorem prover is used to check that the generalized set of concrete states covers all potential executions of P (essentially the set B_P) and satisfies additional safety properties. If this check succeeds, we have a proof that all executions of the program satisfy the given properties.

However, if this check fails, our technique creates a *fabricated* concrete state from which we continue concrete program execution. We use a model generator (a theorem prover that can produce concrete counterexamples) to create a fabricated state so as to increase the coverage. Under some standard assumptions (detailed later) our method is guaranteed to converge and obtain the same result as a standard abstract interpretation of the program P. In particular, our method produces the same amount of false alarms as a standard abstract interpretation (over the same abstract domain). It is noteworthy that we can make this guarantee even if we prematurely halt concrete execution in order to perform the coverage check. In this way, we can control the amount of time spent executing the program vs. the amount of time spent calling the theorem prover.

We make the following contributions:

- We explain the result of abstract interpretation in terms of concrete executions and abstraction. This sheds some light on the trade-offs that arise when combining dynamic and static analyses.
- The result of our method is sound and as precise as the result of the most-precise abstract interpreter (over the same abstract domain).
- We implemented our method in two platforms: the TVLA system for generating shape analyses [24] and the XRT sys-

*This research was supported by The Israel Science Foundation (grant No 304/03).

ISSTA'06, July 17–20, 2006, Portland, Maine, USA.

tem for generating unit tests [18]. The XRT implementation supports all C# features including pointers and procedures. It employs the Simplify theorem prover [10] and demonstrates the feasibility of our approach.

- Our method provides an effective test for abstraction-based adequacy of a test set, as defined by [12, 1].
- We show that our method can find safety proofs with much simpler abstractions than those used by [29], which uses a combination of concrete execution, abstraction and theorem proving to find bisimilar abstractions of programs.

Section 2 gives a high-level overview of our method and illustrates it using a simple example. Section 3 formalizes our method using the framework of abstract intepretation and describe a symbolic algorithm for it. Section 4 discusses some of the practical issues that arise when implementing the algorithm and describes our implementation on top of the XRT infrastructure. Section 5 presents a set of examples illustrating the benefits of our approach. Section 6 compares our approach to related work and Section 7 concludes the paper.

2. OVERVIEW

It is well known that the problem of proving safety properties can be reduced to proving that a program point is unreachable, which is undecidable in general. Fortunately, these properties often can be proved using abstraction to overapproximate the reachable concrete states of a program.

Abstraction and abstract interpretation [6] are key tools for automatically proving properties of systems, both for hardware [5, 9] and software systems [27]. An abstraction function α maps concrete program states to abstract states. An abstract state is reachable if it is the abstraction of some reachable concrete state. Identifying exactly the reachable abstract states is undecidable in general. Abstract interpretation provides a way to compute a *superset* of all reachable abstract states. Thus, the result of abstract interpretation can be used to check safety properties: if safety properties hold on (a superset of) all reachable abstract states, then these safety properties also hold on all reachable concrete states.

2.1 Analysis Method

We propose a new method for computing a superset of all reachable abstract states. In contrast to abstract interpretation, which "executes" the program on abstract states, our method executes the program on concrete states, and then performs abstraction, as shown in Fig. 1. Our method has five steps:

Execute. Given a program P and a set of test inputs T, execute the program and collect the concrete program states C_T obtained during execution.

Abstract. We say that an abstract state is *covered* by a set of tests T if it is the abstraction of a concrete state reachable from some state in T. Our method uses the abstraction function α to obtain the set of abstract states that correspond to C_T: $A_T = \alpha(C_T)$, i.e., A_T is the set of abstract states covered by T.

Check Adequacy. We define an *abstraction-based adequacy criteria* for a set of tests. A set of tests T is adequate under a given abstraction if all reachable abstract states are covered by T. Note that if a set of tests T is adequate then safety properties can be (conservatively) checked on A_T—the abstract states covered by T.

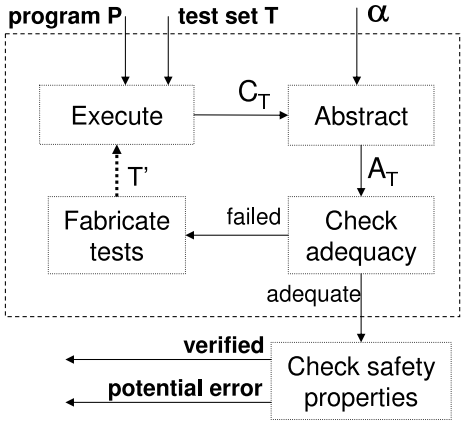

Figure 1: Overview of the method. C_T denotes the set of concrete states reachable from the states in T. A_T is the set of abstract states covered by T, i.e., $A_T = \alpha(C_T)$.

Our method checks a condition that implies adequacy. We check that A_T is invariant under the program P: if a concrete state is represented by A_T then its successor states also are represented by A_T. Formally, we check that for all concrete states c and c' such that $\alpha(\{c\}) \subseteq A_T$ and c' is a successor state of c in the program P, $\alpha(\{c'\}) \subseteq A_T$. This condition is expressed as a logical formula using strongest postconditions [11], such that if the formula is valid then T is adequate. The validity of the formula is checked using a theorem prover.

Fabricate Tests. It is a fundamental (but often ignored) fact that abstract interpretation proves properties about unreachable as well as reachable abstract states. Testing can only cover reachable abstract states. To make abstraction-based testing as powerful as abstract interpretation, our method creates *fabricated states*, intermediate program states that are not necessarily reachable states of the program.

More precisely, if the test set T is not adequate then there are concrete states c and c' such that $\alpha(\{c\}) \subseteq A_T$ and c' is a successor state of c in P, but $\alpha(\{c'\}) \not\subseteq A_T$. Our method finds such a state c' using a model generator, i.e., a theorem prover that produces a concrete counter-example for invalid formulas. Then, our method augments T with c' and repeats the process, as shown in Fig. 1. This guarantees that the coverage increases in the next iteration, and that the process terminates with an adequate set T and the corresponding A_T (for finite-height abstractions or abstraction with widening).

Check Safety Properties. We check, using a theorem prover, whether the covered abstract states A_T satisfy the safety properties. This safety check can be performed either as part of an adequacy check, or after an adequate test set is obtained. If the safety check succeeds for an adequate test set, we have proven safety properties of the program based on the information obtained from executing the test set. If the safety check fails, we report a *potential error*, which may indicate a real error in the program or a false alarm, due to the abstraction.

Our analysis (and abstract interpretation) does not distinguish between a false error and a real error. It is possible to combine our method (and abstract interpretation) with an analysis for classifying potential errors into real errors and false alarms. In this paper, we

assume that the abstraction α is given. It is possible to combine our method with abstraction refinement to find a suitable abstraction.

2.2 Example

We now illustrate our basic method using a simple example. Consider the following C procedure `foo` which contains a null pointer dereference error at line G:

```
foo(int x, int y) {
     int *px = NULL;
  A: x = x + 1;
  B: if (x<4)
  C:   px = &x;
  D: if (px==&y)
  E:   x = x+1;
  F: if (x<5)
  G:   *px = *px+1;
  H: return;
}
```

The state space of this procedure is described by a quadruple (pc, x, y, px), where pc is the value of the program counter ranging over the labels A–H, x and y are the integer values of variables x and y, and px is the value of px of type integer pointer. To simplify the exposition, we assume that `foo` can be called with any integer values x and y, which defines the set of all possible initial states. When `foo` is called with $x = 3$, the statement at label G will execute with $px = NULL$, causing a null pointer dereference. Note that the conditional at label D always evaluates false, so the assignment statement at label E is dead code.

We use the following abstraction function:

$$\alpha(C) = \{(pc, x < 5, px = NULL) \mid (pc, x, y, px) \in C\}$$

Here, a concrete program state is mapped to a triple of values for the following expressions: the program counter pc (ranging over eight labels), the predicate $(x < 5)$ and the predicate $(px = NULL)$. A predicate evaluates to true (t) or false (f). For a singleton set C, we abuse the notations slightly by writing $\alpha(pc, x, y, px)$ instead of $\alpha(\{(pc, x, y, px)\})$.

The abstract state space is finite, consists of $8 \times 2 \times 2 = 32$ possible triples. The reachable abstract states are shown in Fig. 2(a). Our method does not construct these abstract states beforehand. Instead, we execute the procedure on a set of tests and compute, using α, the abstraction of the concrete states encountered during test executions.

Now consider executing `foo(2,0)`, `foo(6,0)`, `foo(11,0)`, that define the test set $T = \{(A, 2, 0, NULL), (A, 6, 0, NULL), (A, 11, 0, NULL)\}$. The test set T does not uncover the null-pointer dereference in the program. The abstract states covered by the execution of T, denoted by A_T, are marked in Fig. 2(a) with bold-circles. Note that the error abstract state (G, t, t) is not in A_T.

2.2.1 Finding a Bug

Next, we check whether the test set T is adequate under α. That is, is the set of covered abstract states A_T an invariant? This check fails, because there is a concrete state b such that $\alpha(\{b\}) = (B, t, t)$, a covered abstract state, from which in one step of the program it is possible to reach a concrete state d such that $\alpha(\{d\}) = (D, t, t)$, an uncovered abstract state. Using a model generator, our method fabricates such a pair of concrete states, say $b = (B, 4, 0, NULL)$ and $d = (D, 4, 0, NULL)$. Execution from state b leads to a null pointer dereference error at label G, as shown in Fig. 2(a). Thus, our analysis reports a potential error. In this case the program contains a real error, because the state b is a reachable concrete state (reachable from the initial state $(A, 3, 0, NULL)$).

2.2.2 Finding a Proof

Now, let us consider what our technique does on a version of the above procedure obtained by modifying the conditional at label B from $(x < 4)$ to $(x < 5)$, which eliminates the null pointer dereference. Let us call the new version `fixed_foo`. The abstract state space of `fixed_foo`, obtained using the abstraction function α as before, is shown in Fig. 2(b).

Again, the set of covered abstract states is not an invariant. In particular, there is a concrete state d' such that $\alpha(\{d'\}) = (D, t, f)$ from which in one step of the program it is possible to reach a concrete state e' such that $\alpha(\{e'\}) = (E, t, f)$, an uncovered abstract state. A pair that satisfies this constraint is $d' = (D, 4, 0, \&y)$ and $e' = (E, 4, 0, \&y)$. Note that neither of these states is a reachable concrete state, as the address of the variable y is never assigned to the variable px in the program, and thus the label E is not reachable. However, in the abstract state space, the label E is reachable whenever the predicate $(px = NULL)$ is false at label D.

Concrete execution from the state d' covers the additional abstract states: $(E, t, f), (F, f, f)$, and (H, f, f). At this point, our method shows that the set of covered abstract states is an invariant. This implies that the abstract states represent all reachable concrete states of `fixed_foo`. Since these abstract states do not contain the error state (G, t, t), we have found a proof that there is no null pointer dereference at label G.

Interestingly, (H, f, f) represents some reachable concrete states, i.e., there exists a test input (not in our test set) that covers (H, f, f). However, finding such an input is a non-trivial task, because to cover (H, f, f) `fixed_foo` must be called with precisely $x = 3$ as an argument (and any value of y). Random or symbolic path-exploration techniques can be used to address the problem, but we avoid this problem using fabrication. The abstract state (H, f, f) is covered by a test execution that starts from the fabricated state d'. It shows that we can benefit from fabricated states to discover abstract states that are reachable via rare executions.

Moreover, execution of d' covers the abstract state (F, f, f), which does not represent any reachable concrete state (although the statement at label F is reachable). Fortunately, all executions from this abstract state are safe, because px is not $NULL$ in this abstract state. That is, the current abstraction shows the absence of errors in all feasible executions as well as in some infeasible executions. This shows the strength of our method: we obtain a proof using a much coarser abstraction than the existing methods [29, 1] that are based only on feasible executions. In this example, these methods would unnecessarily refine the abstraction, even though the current abstraction is sufficient to prove the absence of null pointer dereferences (and our method obtains a proof).

2.2.3 Finding A False Error

Now, let us show an abstraction function that is not precise enough to prove the correctness of the (corrected) program `fixed_foo` above. Suppose our abstraction function is:

$$\alpha'(C) = \{(pc, x < 10, px = NULL) \mid (pc, x, y, px) \in C\}$$

In this case, the abstraction function cannot precisely track the relationship between the value of x and the predicate $(px = NULL)$ in the program. Fig. 2(c) shows the reachable abstract states for this new abstraction function.

Using a theorem prover and a model generator, we find a concrete state f' such that $\alpha'(\{f'\}) = (F, t, t)$ from which in one step of the program it is possible to reach a concrete state g such that $\alpha'(\{g'\}) = (G, t, t)$, an uncovered abstract state. A pair that satisfies this constraint is $f' = (F, 4, 0, NULL)$ and $g' = (G, 4, 0, NULL)$. Note that f' is not a reachable concrete state.

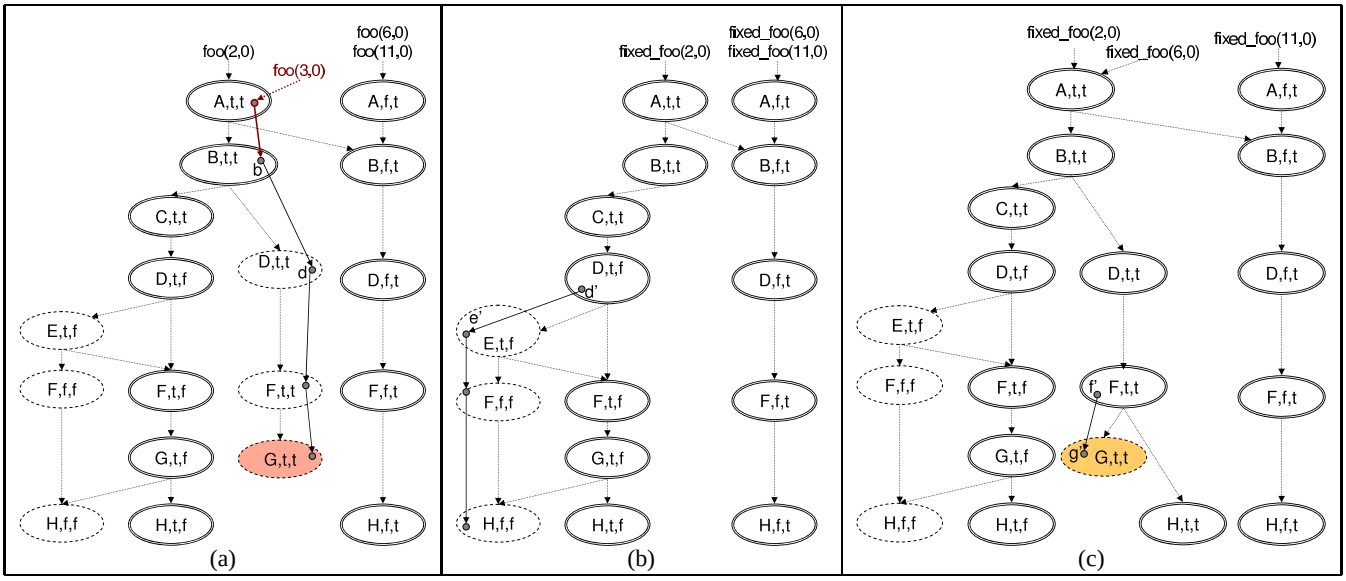

Figure 2: Reachable abstract states for (a) finding a null pointer dereference in `foo` using $\alpha(pc, x, y, px) = (pc, x < 5, px = NULL)$; (b) finding a proof for `fixed_foo` using $\alpha(pc, x, y, px) = (pc, x < 5, px = NULL)$; (c) finding a false error in `fixed_foo` using $\alpha'(pc, x, y, px) = (pc, x < 10, px = NULL)$. Abstract states covered by the set of tests $T = \{(A, 2, 0, NULL), (A, 6, 0, NULL), (A, 11, 0, NULL)\}$ are marked with bold-circles.

Running the program from state f' will cause a null pointer dereference to occur at label G. At this point, our analysis reports a potential error. This false error also would be reported by an abstract interpreter using the abstraction function α'.

Our analysis does not distinguish between a false error such as this one, and a real error such as the one in Section 2.2.1. Both are reported as *potential* errors. In particular, our method may not discover the test input $(A, 3, 0, NULL)$ mentioned in Section 2.2.1.

3. FORMAL DESCRIPTION

This section formalizes our method and compares it to the traditional static analysis by abstract interpretation. Section 3.1 quickly reviews relevant terminology about abstract interpretation. Section 3.2 presents an idealized version of our method and discusses its basic properties. It does not provide an effective algorithm, as it uses an incomputable operation. Section 3.3 discusses how to realize the method as a symbolic algorithm which employs a theorem prover and a model generator.

3.1 Abstraction and Concretization

Let $\mathcal{C}$ be a set of potential concrete states of program P. Let $\mathcal{A}$ be a set of potential abstract values.

In abstract interpretation, we usually assume that $\mathcal{A}$ forms a lattice, with partial order $\sqsubseteq$, meet $\sqcap$ and join $\sqcup$ operations (see Appendix A). In the previous section, we used a powerset lattice, in which an element (abstract value) is a set of abstract states, ordered by set inclusion $\subseteq$, meet is set-intersection $\cap$, and join is set-union $\cup$.

An **abstraction function** $\alpha\colon 2^{\mathcal{C}} \to \mathcal{A}$ yields an abstract value that represents a set of concrete states. A **concretization function** $\gamma\colon \mathcal{A} \to 2^{\mathcal{C}}$ yields a set of concrete states that an abstract value represents. The partial order on $\mathcal{A}$ satisfies for all $a, a' \in A$,

$$a \sqsubseteq a' \iff \gamma(a) \subseteq \gamma(a'). \tag{1}$$

The concretization and the abstraction functions form a **Galois connection** between $2^{\mathcal{C}}$ and $\mathcal{A}$, i.e., for all $a \in \mathcal{A}$ and $X \subseteq \mathcal{C}$:

$$\alpha(X) \sqsubseteq a \iff X \subseteq \gamma(a) \tag{2}$$

This implies that $X \subseteq \gamma(\alpha(X))$ if and only if $\alpha(\gamma(a)) \sqsubseteq a$. That is, abstraction followed by concretization potentially yields more states, and concretization followed by abstraction potentially yields a more precise abstract value.

Given an abstraction function α, it is easy to define the corresponding concretization function: $\gamma(a) = \{c \mid \alpha(\{c\}) \sqsubseteq a\}$. For example, using the abstraction function α' from Section 2.2.3, $\gamma(\{(G, f, t)\}) = \{(G, i, j, NULL) \mid i \geq 10\}$.

To simplify the presentation, we assume that an abstract value in $\mathcal{A}$ collectively describes states at all program points, rather than having a separate abstract value for each program counter. This can be achieved by encoding the program counter in the representation of a concrete state $\mathcal{C}$, as we did in the previous section.

The program P defines a transition relation on concrete states $\to_P\colon \mathcal{C} \times \mathcal{C}$. For $\sigma, \sigma' \in \mathcal{C}$, we say that σ' is a successor of σ if $\sigma \to_P \sigma'$. Intuitively, σ' is a result of executing a single statement of P in the state σ. We define the function $f_P\colon \mathcal{C} \to \mathcal{C}$ as follows:

$$f_P(X) = \{\sigma' \mid \sigma \to_P \sigma', \sigma \in X\} \cup \{X\}$$

Note that f_P is monotone. We drop the subscript P when it is understood from the context. The set of concrete states reachable from $X \subseteq \mathcal{C}$ is the least fixpoint of f w.r.t. X, denoted by $\mathrm{LFP}_X(f)$.

Let $I \subseteq \mathcal{C}$ be the set of all possible initial states of a program P. The meaning of the program P is the set of all concrete states reachable from some initial state: $\mathrm{LFP}_I(f)$.

An abstract value $a \in \mathcal{A}$ is a **sound** overapproximation of P if a represents all concrete states reachable from I (but possibly other states):

$$\mathrm{LFP}_I(f) \subseteq \gamma(a). \tag{3}$$

```
[1]procedure basic(T0)
[2]   a := ⊥
[3]   T := T0
[4]   while(true) begin
[5]     C := Execute(f,T)
[6]     a := a ⊔ α(C)
[7]     if exists σ ∈ f(γ(a)) s.t.  σ ∉ γ(a)
[8]     then T := {σ}
[9]     else return a
[10] end
```

Figure 3: The basic procedure. Here, $T_0, T, C \subseteq \mathcal{C}$, and $a \in \mathcal{A}$. If $\alpha(T_0) = \alpha(I)$, then the result of the procedure is a sound approximation of P.

An abstract value $a \in \mathcal{A}$ is **invariant under** P if

$$f(\gamma(a)) \subseteq \gamma(a) \tag{4}$$

THEOREM 3.1. **(Soundness)** *If an abstract value $b \in \mathcal{A}$ is invariant under P and $I \subseteq \gamma(b)$ then b is a sound overapproximation of P.*

3.2 Basic Procedure

Fig. 3 shows a high-level description of our method. Implementation details are discussed in the next sections. We assume that the basic procedure is called with a finite set T_0 of initial states ($T_0 \subseteq I$), such that $\alpha(T_0) = \alpha(I)$.

Line [5] corresponds to the *Execute* step, described in Section 2.1. Formally, $Execute(f, T)$ returns a subset of the states reachable from T that contains at least the states in T:

$$Execute(f, T) \subseteq \text{LFP}_T(f) \text{ and } T \subseteq Execute(f, T). \tag{5}$$

Note that it is not necessary (and sometimes impossible) to collect all states reachable from T. In particular, this step allows us to handle non-terminating executions or very long running executions.

In line [6] the abstraction of the obtained concrete states is computed using α. This corresponds to the *Abstract* step described in Section 2.1. The procedure terminates when it is not possible to fabricate a state σ that satisfies the condition in line [7], i.e., a is invariant under P. This implies that a is a sound approximation of the reachable states of P (by Theorem 3.1).

Furthermore, the procedure computes the same abstract value that is computed by the most-precise abstract interpreter of P for the given abstraction, as stated by the following theorem:

THEOREM 3.2. *Let $f^\sharp : \mathcal{A} \to \mathcal{A}$ be defined by $f^\sharp = \alpha \circ f \circ \gamma$. The procedure in Fig. 3 computes the least fixpoint of $f^\sharp$ w.r.t. $\alpha(I)$: $LFP_{\alpha(I)}(f^\sharp)$.*

The particular choice of a fabricated concrete state in line [7] does not affect the final result of the procedure, but it may affect the number of iterations needed to find the result, as explained below.

Let S_a be the set of all possible fabricated states for an abstract value a, i.e., the set of states σ that satisfy the condition in line [7]:

$$S_a \stackrel{\text{def}}{=} \{\sigma \mid \sigma \in f(\gamma(a)), \sigma \notin \gamma(a)\}$$

In line [8], a single fabricated state is chosen from S_a. It is easy to modify the procedure to work with several fabricated states together in the same iteration. Because fabricated states are not covered by the abstract values collected so far, the coverage strictly increases in successive iterations.

THEOREM 3.3. *If the lattice $\mathcal{A}$ has a finite height, the procedure in Fig. 3 terminates.*

Remark. If the lattice $\mathcal{A}$ admits infinite ascending chains (e.g., polyhedra [7]), it is possible to use standard *widening* techniques to enforce and accelerate termination of our procedure, sacrificing the precision of its result. Let $\triangledown : \mathcal{A} \times \mathcal{A} \to \mathcal{A}$ denote the widening operator on $\mathcal{A}$ (see Appendix A). We replace line [6] with $a := a \triangledown \alpha(C)$. The result a of the procedure satisfies $\text{LFP}_{\alpha(I)}(f^\sharp) \sqsubseteq a$, but the result a may be less precise than $\text{LFP}_{\alpha(I)}(f^\sharp)$.

Choosing any state in S_a strictly increases the coverage, but some states increase the coverage more than others. Intuitively, we would like to choose a fabricated state that covers as many new abstract states as possible, "jumping" higher in the abstract lattice. A good choice of fabricated states reduces the number of the iterations of the procedure, and thus, the number of calls to a theorem prover and a model generator.

3.3 Symbolic Procedure

For all abstractions we are aware of, the function α is efficiently computable given a (finite) set of concrete states, represented explicitly. Note that applying α does not require the use of a theorem prover.

Nonetheless, as mentioned before, the procedure in Fig. 3 does not provide an effective algorithm. In particular, the γ operation used in line [7] is not computable, as $\gamma(a)$ may be infinite. We now show how to implement line [7] symbolically, using a theorem prover and a model generator.

Symbolic Characterization. Concrete program states can be represented as logical structures, in which constant symbols model program variables. Thus, sets of concrete states can be described by logical formulas in some logic $\mathcal{L}$ (e.g., first-order logic). The concretization function γ can be expressed symbolically, i.e., for every $a \in \mathcal{A}$, there exists a formula in $\mathcal{L}$, denoted by $\widehat{\gamma}(a)$, that exactly represents a: for all $\sigma \in \mathcal{C}$,

$$\sigma \models \widehat{\gamma}(a) \text{ if and only if } \sigma \in \gamma(a) \tag{6}$$

In the example from Section 2.2, with abstraction function α, $\widehat{\gamma}$ can be expressed as a quantifier-free first-order formula, interpreted over integers. The constant symbols x and y model the values of the corresponding program variables. The constant symbols A–H model each of the program points, and an additional constant symbol pc models the program counter.[1] Finally, the fact $px = NULL$ can be encoded with a corresponding Boolean. For instance, $\widehat{\gamma}(\{(G, f, t)\})$ is the formula $(pc = G) \wedge \neg(x < 5) \wedge (px = NULL)$.

Meaning of Program Statements. The meaning of a program can be expressed as a formula transformer, $\mathcal{SP} : P \times \mathcal{L} \to \mathcal{L}$, which defines the strongest postcondition [11]: for every formula $\varphi \in \mathcal{L}$, a concrete state σ' satisfies $\mathcal{SP}(P, \varphi)$ if and only if there exists a concrete state σ such that σ' is a successor state of σ in P and σ satisfies φ. Intuitively, $\mathcal{SP}$ describes the result of executing a single statement of P on a state that satisfies φ. For example, $\mathcal{SP}(\texttt{x = x + 1}, x > 5)$ is the formula $x > 6$.

Symbolically Checking for Invariance. Using the $\widehat{\gamma}$ and $\mathcal{SP}$ operations, we can symbolically express the fact that abstract value $a \in \mathcal{A}$ is an invariant:

$$\mathcal{SP}(P, \widehat{\gamma}(a)) \Rightarrow \widehat{\gamma}(a) \tag{7}$$

[1]Not every interpretation of these constants is legal; to rule out illegal interpretations of pc, the following axiom can be used: $pc = A \vee pc = B \vee \ldots \vee pc = H$.

The formula (7) is valid if and only if a is an invariant. [2]

Given a program P and an abstract value a, our method can automatically generate the formula in (7). Moreover, it can check the validity of (7) automatically using a theorem prover for $\mathcal{L}$. If the validity check fails, then a model generator can be used to fabricate a state that satisfies $\widehat{\gamma}(a)$ and has a successor that does not satisfy $\widehat{\gamma}(a)$. Formally, the fabricated state satisfies the negation of (7):

$$\mathcal{SP}(P, \widehat{\gamma}(a)) \wedge \neg\widehat{\gamma}(a) \tag{8}$$

For example, in Section 2.2.1, if a is $\{(B, t, t)\}$, then the formula $\widehat{\gamma}(a)$ is $(pc = B \wedge x < 5 \wedge px = NULL)$. The strongest postcondition of this formula and the statement `if (x < 4)` at label `B` is the formula $sp \stackrel{\text{def}}{=} (pc = C \wedge x < 4 \wedge px = NULL) \vee (pc = D \wedge x = 4 \wedge px = NULL)$. Clearly, $sp \Rightarrow \widehat{\gamma}(a)$ is not valid. This allows us to fabricate a state, say $d = (D, 4, 0, NULL)$, such that $d \models sp \wedge \neg(pc = B \wedge x < 5 \wedge px = NULL)$.

Symbolically Checking Safety Properties. In our setting, the safety properties of interest also can be expressed by a formula $\varphi \in \mathcal{L}$. For example, in Section 2.2 the safety property can be expressed by the formula $\varphi \stackrel{\text{def}}{=} \neg(pc = G \wedge px = NULL)$.

If $a \in \mathcal{A}$ is a sound approximation of P, and the formula $\widehat{\gamma}(a) \Rightarrow \varphi$ is valid, then all reachable concrete states of P satisfy the safety properties φ. In Section 2.2.2, our method obtains a set of abstract states a for which $\widehat{\gamma}(a) \Rightarrow \varphi$ is valid. In Section 2.2.3, one of the abstract states covered by our method is (G, t, t), for which $(pc = G \wedge x < 10 \wedge px = NULL) \Rightarrow \neg(pc = G \wedge px = NULL)$ is not valid, and our method reports a potential error.

In practice, there are automatic tools for checking validity and generating models (even if the logic $\mathcal{L}$ is undecidable), which have certain limitations, as discussed in the next section.

4. TOWARDS A REALISTIC IMPLEMENTATION

In this section, we discuss some of the practical issues that arise when implementing the algorithm described in Section 3.

4.1 Program Analysis Infrastructure

Our method requires an infrastructure that supports: (1) monitoring of concrete program states in concrete executions to compute abstract state coverage; (2) symbolic execution of loop-free code fragments to compute $\mathcal{SP}$; (3) state manipulation to create fabricated states.

Explicit-state model checkers such as SPIN [21], CMC [26], JavaPathFinder [34], XRT [18], which perform systematic and exhaustive testing, provide a good starting point (though not all support symbolic execution). A model checker analyzes several executions of the program at once, and controls the order in which these concrete executions advance (e.g., DFS, BFS). A model checker usually manipulates a representation of concrete states, which comes in handy for fabrication of states. Our implementation uses XRT, see Section 4.5.

4.2 Cutpoints

Section 3 simplified the discussion by encoding the program counter in the abstract value. This encoding allows us to keep abstract states for each program point. In practice, it is not necessary to track all program points. We can choose a designated set of program points, called *cutpoints*, where the abstraction is computed. As in deductive verification, a minimal set of cutpoints is a set which cuts every cycle in a program's control flow graph [15].

The runtime overhead of computing abstract state coverage decreases when there are fewer cutpoints. Furthermore, having fewer cutpoints potentially improves the precision of our method, as the abstraction of a composition of two statements is usually more precise than a composition of their abstractions.

The check that an abstract value is an invariant is adapted according to the set of cutpoints: the strongest postcondition formulas $\mathcal{SP}$ in (7) describe the result of executing a sequence of statements from one cutpoint to the next (and not a single statement, as before). Note that states are fabricated only at cutpoints.

Instead of checking the validity of a large formula (7), our method can make several validity checks with smaller formulas, ψ_L, one for each cutpoint L. Intuitively, ψ_L describes a "local" invariant, under the loop-free code fragment which starts at L. The formula ψ_L involves only abstract states related to L and to the cutpoints immediately following L in the control-flow graph of the program. Thus, in successive iterations, only the checks that involve the new abstract states need to be repeated.

4.3 On-the-fly Abstraction

We previously presented execution and abstraction as separate steps (line `[5]` and line `[6]` of Fig. 3). In practice, abstraction can be computed *on the fly* during program execution. The idea is to monitor the execution: when a cutpoint is reached, we pause the execution to compute the abstraction of the current state, conjoin the resulting abstract value with the abstract value collected so far, and continue the execution. This way, concrete states encountered during program execution need not be stored (only abstract values need to be stored).

4.4 Employing a Theorem Prover

The success of our method depends on having a theorem prover and a model generator which can generate a concrete counterexample to validity of a formula. Technically, there are off-the-shelf automatic theorem provers that can be used, e.g., SPASS [35], Vampire [30], Simplify [10]. Unfortunately, most such theorem provers do not produce concrete counterexamples for invalid formulas (with the exception of Darwin [3]). Instead, a separate tool for model generation can be used (e.g., Paradox [4]).

For certain abstractions, the queries posed by our method can be expressed in a decidable logic, which guarantees a (terminating) decision procedure. Additionally, the following difficulties arise when using theorem provers and model generators:

(A) The theorem prover might fail to prove validity of a (valid) formula (e.g., Simplify [10] might return "invalid" for a valid formula with quantifiers).

(B) The theorem prover might timeout without a conclusive answer, because it exceeds the time or the amount of resources allocated for it.

(C) The model generator might fail to fabricate a state that satisfies (8), (e.g., because the formula is, in fact, valid, but theorem prover failed to prove its validity).

If a theorem prover fails (due to (A) or (B) above) when checking safety properties, our method may produce a false error report. Even if a theorem prover fails when checking that the abstract value is an invariant, our method still tries to fabricate a state that satisfies (8). If model generation succeeds, the analysis continues as before.

[2] Alternatively, a formula based on the weakest (liberal) precondition can be used.

Failure of the model generator to fabricate a state can be handled as follows:

- Fabricate a state that satisfies $\neg\widehat{\gamma}(a)$. This guarantees that the coverage increases in each iteration, and the analysis eventually terminates, but it might fail to produce the most-precise result (because the fabricated state may not have a predecessor in any covered state).
- Fabricate some concrete state, say using a random generator, sacrificing both termination of the analysis and its precision. However, if the analysis terminates, its result still is sound.
- Use a hybrid approach which combines concrete execution and abstract interpretation, as discussed in Section 5.3.

4.5 Prototype Implementation based on XRT

We have implemented our method on top of the XRT framework [18], an extensible framework for explicit and symbolic model checking of programs, represented in Microsoft's common intermediate language (CIL). XRT processes .NET managed assemblies, and provides means for analyzing, rewriting, and executing programs. Our implementation takes advantage of all these features.

Our implementation uses predicate abstraction [17] (without refinement), and supports user-defined predicates. It also can automatically generate a default set of predicates by a backwards dataflow analysis from the conditional branches that infers the predicates governing these branches.

In our implementation, predicates are defined as C# methods, called *probes*. Probes return Boolean values, have no side-effects and contain no loops or method calls. Each cutpoint is instrumented to call the probe methods, so as to evaluate the appropriate predicates on the current state. We compute the abstraction of the concrete execution on-the-fly: the XRT runtime pauses execution immediately after a call to a probe method returns, and its return value is used to update the abstract state.

When predicates are given as logical formulas, it is easy to implement $\widehat{\gamma}$ for predicate abstraction, but in our case predicates are given as CIL code. The symbolic execution of probe methods by XRT gives a natural way to construct logical formulas for predicates. The symbolic execution mode of XRT also provides strongest postconditions that our method uses to check whether the abstract value is invariant. To check validity, we use the Simplify theorem prover [10]. To fabricate states, we implemented a naïve model generator within XRT, also based on Simplify.

Our implementation places cutpoints: (i) before each loop body (to cut cycles); (ii) on entry and exit of every method, (iii) before and after every method call, and (iv) at each program point that may potentially violate safety properties, such as a pointer dereference or an array access. To handle method calls, we have implemented 0-CFL Reachability [27], utilizing the cutpoints of (ii,iii).

5. WHY "BETTER TOGETHER"?

This section discusses some of the benefits of our method.

5.1 Unit-Testing with Fabrication

A major application of XRT is in the area of unit testing. Therefore, to evaluate the implementation of our method, we adapted the method to operate on a separate class using unit-tests for that class (rather than analyzing a closed application using its test inputs). A similar approach can be applied to analyze an open system or a component.

```
public class BoundedStack {
 private int[] elems;
 private int size;
 private int max;
 ...
 public BoundedStack(int capacity)   {
    size = 0;
    // fixme: if (capacity <= 0) capacity = 2;
    max = capacity;
    elems=new int[max];
 }
 public void pop() {
   // fixme: if (size >= 0)
   size--;
 }
 public void push(int k) {
   int index;
   bool alreadyMember;
   alreadyMember = false;
   for(index=0; index<size; index++) {
    if(k==elems[index]) {
      alreadyMember = true;
      break;
     }
   }
   if (alreadyMember) {
     for (int j=index; j<size-1; j++) {
       elems[j] = elems[j+1];
     }
     elems[size-1] = k;
   }
   else {
     if (size < max) {
L1:    elems[size] = k;
       size++;
       return;
     } else {
       return;
     }
   }
 }
}
```

Figure 4: Implementation of a bounded stack using fixed-size array (abbreviated). The comments show the code needed to fix the errors.

The idea is to invoke each method of the class on all the concrete states obtained on exit of any method of this class. We illustrate the idea on the implementation of a bounded stack, used previously in the literature.

EXAMPLE 5.1. *Fig. 4 shows an abbreviated version of the code that implements a bounded stack, using a fixed-size array.*

A bounded stack can normally be 'empty', 'partially full' or 'full'. We used predicate abstraction to capture these states, and distinguish them from illegal states in which `size` *is out of the bounds of* `elems`*.*

The bounded stack supports the usual operations, but it does not provide any exceptional behavior. Instead, if an operation is applied in an inappropriate state, it has no effect. For example, if the stack is full, the `push` *operation has no effect. However, the* `pop` *method incorrectly handles popping an empty stack. This problem was not exhibited by the provided unit tests, because* `pop` *is never called with an empty stack.*

We analyzed the class using our implementation based on XRT, checking for `IndexOutOfRangeException`*. In the first iteration, our method fabricates a state σ on exit of* `pop`*, with an empty*

stack. Then, it executes the method `pop` *again on the fabricated state σ, obtaining a new state σ' on the exit of* `pop`*, with $size < 0$ (no runtime exception occurs). Then, it executes the method* `push` *on σ', causing an* `IndexOutOfRangeException` *on line* `L1`*.*

After fixing the error in `pop`*, our analysis automatically proves absence of* `IndexOutOfRangeException` *in this example, using four fabricated states, and the default predicates, as mentioned in Section 4.5. If the maximal size provided to the constructor is negative, it throws* `OverflowException` *exception, but this error should not be reported by the analysis, which tracks different exceptions.*

This example shows that our analysis can deal with unexpected failures, e.g., when a concrete execution throws an exception that is not tracked by the analysis. If such exception is thrown in a concrete execution, our analysis can fabricate any state in the following program point, and continue the execution from it. This fabrication is easy because it does not place any constraints on the fabricated state. It provides a sound and (perhaps, surprisingly) most-precise result, because behavior that is not modeled is treated by a sound abstraction as if "anything can happen".

Upon termination of the analysis, the abstract states on the entry and exit of all methods are the same. This set of abstract states, in fact, represents the class invariants, under certain conditions about the class, stated in [25]. The analysis can output the inferred class invariants in the form of logical formulas, by computing $\widehat{\gamma}$ of the relevant abstract states.

Note that, as we are using state-based abstractions, our approach cannot learn method-call order. Also, our method analyzes each class independently of the actual clients of this class. The approach may also report on potential errors, that do not occur in any actual client of this class. The advantage of this approach is that it identifies potential errors early in the development process, even before the client is written. If our method succeeds, it provides a proof of safety for the class in any client. This proof can be used to perform assume-guarantee reasoning.

Also, in the setting of unit-testing, it is easier to classify a potential error reported by the analysis, because the context of all method calls (a client code) is arbitrary. In the bounded stack example, real errors were detected by a fabricated execution.

Recall that the purpose of fabrication is to find a proof faster. Inherent to this approach is the fact that a fabricated state may be unreachable from any initial state of the program. However, states that are fabricated on a method entry can be used in unit-test generation.

5.2 Avoiding Unnecessary Abstraction Refinement

We are not the first to demonstrate that concrete execution plus abstraction can be used to verify program properties [22, 29, 1]. However, previous work in the area required much stronger abstractions than necessary to verify the safety properties of interest.

One approach is to find an abstract system that is *bisimilar* to the underlying concrete system, using automated refinement of abstractions [22, 29]. For deterministic systems, this means that the concrete system and abstract system have identical execution traces. The advantage of [22, 29] is that it reports only real errors. However, this technique is often too strong for proving safety properties. Even for proving a simple program, bisimulation might require a complex abstraction generated via many iterations of abstraction-refinement, whereas our technique can achieve proofs with a coarser abstraction.

As an example, we apply our method to the Bakery mutual exclusion protocol for two processes, which also was analyzed in [29].

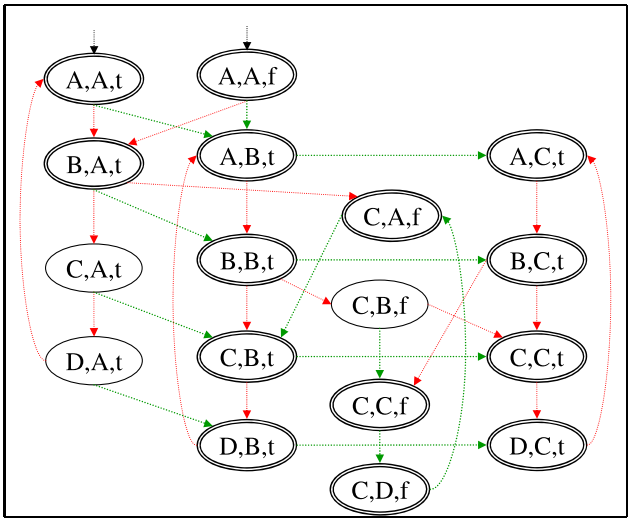

Figure 5: Reachable abstract states of two-process Bakery protocol, using an abstraction function $x \leq y$.

The guarded command representation of the protocol is:

$$
\begin{array}{lcl}
\textbf{Process1}: & & \\
pc_1 = A & \mapsto & x_1 := x_2; \\
pc_1 = B & \mapsto & x_1 := x_1 + 1; \\
pc_1 = C \wedge x_1 \leq x_2 & \mapsto & pc_1 = 3 \\
pc_1 = D & \mapsto & pc_1 = 0 \\
\textbf{Process2}: & & \\
pc_2 = A & \mapsto & x_2 := x_1; \\
pc_2 = B & \mapsto & x_2 := x_2 + 1; \\
pc_2 = C \wedge x_2 < x_1 & \mapsto & pc_1 = 3 \\
pc_2 = D & \mapsto & pc_1 = 0
\end{array}
$$

A concrete state of the program is (pc_1, pc_2, x_1, x_2), where pc_i is the value of the program counter of process i, ranging over $A - D$, and x_i is the integer value of the ticket of process i, for $i = 1, 2$.

The safety property we check is that at most one processor can be in the critical section: $\neg(pc_1 = D \wedge pc_2 = D)$. We use the following abstraction function, based on the predicate $x \leq y$:

$$\alpha_{bake}(C) = \{(pc_1, pc_2, x_1 \leq x_2) \mid (pc_1, pc_2, x_1, x_2) \in C\} \quad (9)$$

Fig. 5 shows the reachable abstract states. The abstract error states (D, D, t) and (D, D, f) are not reachable. An explicit-state model checker easily finds a concrete execution that covers the states marked with bold-circles, starting from the concrete state $(A, A, 0, 0)$. Then, our method fabricates 2 states: $(C, B, 1, 0)$ and $(C, A, 0, 0)$. The first state covers the abstract state (C, B, f), and the second state covers (C, A, t) and (D, A, t). At this point, our method proves that the abstract states are invariant, and that they satisfy the mutual exclusion property.

The initial abstraction used in [29] is the same as ours (9). The method of [29] takes 4 steps of abstraction refinement to find an abstract state space that is a bisimulation of the concrete state space. The bisimilar abstract state space contains 36 abstract states, using abstraction based on 10 predicates. We have shown that the initial abstraction is sufficient to prove mutual exclusion (without any abstraction refinement), and the state space has only 17 abstract states.

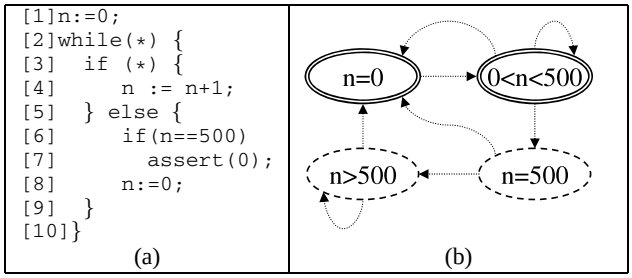

Figure 6: (a) Example program, (b) Abstract state-space.

5.3 Hybrid Approach

For certain programs, concrete execution might go on for a long time without covering a new abstract state, whereas abstract interpretation makes progress in every step, but each step can be expensive or can lose precision. An exciting application of our method is its ability to address limitations of one approach using the other, by interleaving concrete and abstract interpretation.

Recall from Section 3 that stopping concrete execution at any moment does not affect termination or precision of the analysis. In fact, to guarantee termination, it is sufficient to have C `:=` T in line `[5]`. However, covering more abstract states per iteration will reduce the number of iterations of the algorithm.

Our idea is to monitor how many new abstract states are covered during concrete execution. If coverage is not increasing, the concrete execution can be paused to check invariance. If the abstract value obtained so far is not invariant, a hybrid method can increase coverage by fabricating a new concrete state and continue concrete execution from it. In this way, we can control the amount of time spent executing the program vs. the amount of time spent calling the theorem prover.

For example, consider the code in Fig. 6, which uses the nondeterministic choice operation ($*$). A concrete execution that iterates through the loop 500 times, always taking the true branch of the `if` statement in line `[3]`, and then, in the $500 + 1$ iteration, takes the false branch, reaches the assertion in line `[7]`. If the false branch is taken earlier, n is reset in line `[8]`. The assertion in line `[7]` fails in rare executions with a long trace to the error. Such errors are difficult to discover using an explicit-state model checker or random testing. Our hybrid approach can skip the execution of many loop iterations that do not increase coverage.

We use predicate abstraction with the predicates $n = 0$, $0 < n < 500$, $n = 500$, and $n > 500$ for value of n on line `[3]`. Note that the predicates divide the concrete state space into 4 partitions, as shown in Fig. 6(b). Model checking quickly finds concrete execution that covers the states $n = 0$, $0 < n < 500$, but then the concrete execution stays within these abstract states. At this point, the hybrid approach simply fabricates a state with $n = 500$, and continues model checking from it, skipping the long execution trace that leads to it. The model checker can easily find an execution through the loop body to line `[7]`, and reports a potential error.

This example shows how the hybrid approach can help finding errors faster than concrete execution. Furthermore, concrete execution can be combined with abstract interpretation, when a model generator fails to fabricate a new state that satisfies (8).

5.4 Beyond Predicate Abstraction

Our method is applicable beyond predicate abstraction. We have implemented another prototype, based on the TVLA system [24]. The TVLA system performs abstract interpretation using canonical abstraction [31], and supports reasoning about recursive data-structures. We have implemented a special-purpose model generator that uses canonical abstraction to guide the search for models. For concrete execution, we use the TVLA system in a mode where memory abstraction is disabled. This mode faithful simulates concrete state-space exploration for programs, which manipulate fields and pointers, but not integer data.

As a proof of concept, we applied the prototype to TVLA benchmarks that manipulate singly-linked lists. A concrete state describes a memory that contains linked lists. We used four test inputs, each with one linked list in memory: an empty list, and lists of length 1–3. The analysis proved the absence of null-dereferences and absence of memory leaks (i.e., every allocated element is reachable from some program variable). Fig. 7 shows the results: number of fabricated states, abstract states upon termination, and the maximal length of a list used by a concrete execution (either initial test input, or a fabricated state).

In fact, it is sufficient to execute the programs on small lists, with up to 7 nodes, to cover all reachable abstract states. The length depends on the number of program variables that can point into the same list. In other words, if the program has a error on a large list, then it has an error on a small list, up to size 7.

6. RELATED WORK

Automated Construction of Abstract Transformers. Theorem provers have been used for the automated construction of abstract transition systems [2, 20, 36, 33], especially in parametric abstract domains, such as predicate abstraction [17] and canonical abstraction [31], where the abstraction is defined per-program. In many cases, an exponential number of theorem prover calls is needed to compute the effect of a single program statement on an abstract value in the most-precise way.

Compared to these techniques, our method can reduce the number of theorem prover calls: it obtains abstract values via concrete execution, which does not require theorem-prover calls. A theorem prover is used only to check that an abstract value is an invariant (7), which requires one theorem prover call per program statement. If the check fails, then at least one new abstract state is covered in the next iteration. In the worst case, our method might require as many theorem prover calls as other methods. However, if an execution from a fabricated state covers several new abstract states, our method terminates with less theorem prover calls.

The cost of a theorem prover call made by our method is comparable to other methods. However, the cost of a model generation might be higher than the cost of a validity check.

Our method is most-closely related to the algorithm presented in [33]. Both methods rely on a model generator to "fabricate" a concrete state that (i) is not yet represented by the abstract value obtained so far, and (ii) is reachable in a single step from it. In this paper, we have identified a way to cover more abstract states using a single fabricated state, by executing the program. The method of [33] can be described by replacing C `:= execute(`f`,`T`)` with C `:=` T in line `[5]` of Fig. 3.

Combining Dynamic and Static Analyses. Daikon uses dynamic analysis to detect likely invariants [14]. It executes the program on a test set, examining the values of the concrete states, and detects patterns and relationships among those values. It reports properties that hold over execution of the given test set, but not necessary over all program executions. In [28], likely invariants produced by Diakon are used with ESC/Java [23] verifica-

procedure	fabricated states	abstract states	maximal length	description
search	2	21	5	searches a list for an element with a specified value
reverse	4	57	6	reverses a singly-linked list in-situ
insert	3	58	6	creates an element with a specified value and inserts it into an ordered list
getLast	3	36	6	returns a pointer to the last element of a list
deleteAll	1	14	4	deallocates all elements in a list
delete	8	110	7	deletes an element with a specified value from a list

Figure 7: Analysis results for methods that manipulate singly-linked lists.

tion condition generator and Simplify [10] theorem prover to prove that these are indeed invariants. Our work is similar in spirit, but uses fabricated states and abstraction to achieve proof via a fixpoint computation where the Daikon-ESC/Java two-step process may fail to find a proof.

Recent work combines random test generation and concrete execution with symbolic execution and model generation [16, 32, 8]. These methods use symbolic techniques to direct the generation of tests towards unexplored paths in order to find errors faster. However, these methods do not use abstraction, and in general cannot find proofs in presence of loops.

Bisimulation and Weak Reachability. Concrete execution and abstraction are used by [22, 29, 1] to find errors and verify program properties. All errors reported by [22, 29] are real errors, but the technique often is too strong for proving safety, as shown in Section 5.2. Also in [29], concrete exploration stops when it encounters a concrete state whose abstraction was already seen before. Our method continues exploration from such a concrete state, and may discover abstract states that were not covered before.

Another way to achieve verification is to find an abstraction and a set of tests T that cover exactly the reachable abstract states [1]. It requires that every abstract state be testable. This is a weaker property than bisimulation but still stronger than our method, because our method using fabricated states may cover abstract states that are not reachable (but required for a proof).

Test Adequacy. In contrast to the traditional white-box adequacy criteria (e.g., [37]), we choose an abstraction based on the property of interest, and then define adequacy with respect to the abstraction. When used with a powerset abstraction, our adequacy requirement appears to be a formalization of partition-based testing with respect to an abstraction function, where each abstract state represents a partition.

Recently, an abstraction-based adequacy criteria *All-Abstract-States* was introduced in [12], in the context of automatic test generation using a theorem prover, when the abstract states are provided by static analysis. All-Abstract-States criterion implies the adequacy criterion we defined in Section 1. Our algorithm provides an effective way to check adequacy of a given test set.

7. CONCLUSIONS

Our method can be viewed as bridging the gap [19, 13] between testing and verification.

Our method finds the same potential errors as the most-precise abstract interpreter for a given abstraction. Additionally, it provides a new way to tune performance by alternating between concrete execution and symbolic reasoning (theorem proving).

We plan to investigate how the information obtained from fabricate states can be used to (i) generate useful test inputs, (ii) classify potential errors into false alarms and real errors, and (iii) guide abstraction refinement. Another direction is developing metrics based on fabricated states, which provide feedback on the quality of the code and its test set. Also, it is interesting to study the correlations between fabricated states and failures.

8. REFERENCES

[1] T. Ball. A theory of predicate-complete test coverage and generation. In *3rd International Symposium on Formal Methods for Components and Objects*, 2004.

[2] T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic predicate abstraction of c programs. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 203–213, 2001.

[3] P. Baumgartner, A. Fuchs, and C. Tinelli. Implementing the Model Evolution Calculus. In Stephan Schulz, Geoff Sutcliffe, and Tanel Tammet, editors, *Special Issue of the International Journal of Artificial Intelligence Tools (IJAIT)*, International Journal of Artificial Intelligence Tools, 2005. Preprint.

[4] K. Claessen and N. Sorensson. New techniques that improve mace-style finite model finding. In *CADE-19 Workshop: Model Computation - Principles, Algorithms, Applications*, 2003.

[5] E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. *Trans. on Prog. Lang. and Syst.*, 16(5):1512–1542, 1994.

[6] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *Symp. on Princ. of Prog. Lang.*, pages 238–252, New York, NY, 1977. ACM Press.

[7] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Symp. on Princ. of Prog. Lang.*, pages 84–96, 1978.

[8] C. Csallner and Y. Smaragdakis. Check 'n' crash: combining static checking and testing. In *ICSE*, pages 422–431, 2005.

[9] D. Dams. *Abstract Interpretation and Partial Refinement for Model Checking*. PhD thesis, Technical Univ. of Eindhoven, Eindhoven, The Netherlands, July 1996.

[10] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Labs, 2003. http://research.compaq.com/SRC/esc/Simplify.html.

[11] E.W. Dijksta. *A Discipline of Programming*. Prentice-Hall, 1976.

[12] G. Erez. Generating concrete counter examples for arbitrary abstract domains. Master's thesis, Tel-Aviv University, Israel, 2004.

[13] M. D. Ernst. Static and dynamic analysis: Synergy and

duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*, pages 24–27, Portland, OR, May 9, 2003.

[14] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE TSE*, 27(2):1–25, February 2001.

[15] R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics 19*, pages 19–32, Providence, 1967. American Mathematical Society.

[16] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 213–223, 2005.

[17] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV*, pages 72–83, June 1997.

[18] W. Grieskamp, N. Tillmann, and W. Schulte. Xrt exploring runtime for .net: Architecture and applications. In *SoftMC*, 2005.

[19] M. J. Harrold. Testing: a roadmap. In *ICSE - Future of SE Track*, pages 61–72, 2000.

[20] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with blast. In *SPIN*, pages 235–239, 2003.

[21] G. J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.

[22] D. Lee and M. Yannakakis. Online minimization of transition systems (extended abstract). In *STOC*, pages 264–274, 1992.

[23] K. R. M. Leino, G. Nelson, and J. B. Saxe. Esc/java users manual. Technical Report 002, Compaq Systems Research Center, 2000.

[24] T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. In *Static Analysis Symp.*, pages 280–301, 2000. The system is available from www.cs.tau.ac.il/∼tvla.

[25] F. Logozzo. *Modular Static Analysis of Object Oriented Languages*. PhD thesis, LEcole Polytechnique, 2004.

[26] M. Musuvathi, D. Y. W. Park, A. Chou, D. R. Engler, and D. L. Dill. Cmc: A pragmatic approach to model checking real code. In *OSDI*, 2002.

[27] F. Nielson, H.R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[28] J. W. Nimmer and M. D. Ernst. Invariant inference for static checking: An empirical evaluation. In *FSE 2002*, pages 11–20, 2002.

[29] C. Pasareanu, R. Pelanek, and W. Visser. Concrete model checking with abstract matching and refinement. In *CAV*, 2005.

[30] A. Riazanov and A. Voronkov. Vampire 1.1 (system description). In *IJCAR*, pages 376–380, 2001.

[31] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *Trans. on Prog. Lang. and Syst.*, 2002.

[32] K. Sen, D. Marinov, and G. Agha. Cute: a concolic unit testing engine for c. In *ESEC/SIGSOFT FSE*, pages 263–272, 2005.

[33] T.Reps, M. Sagiv, and G. Yorsh. Symbolic implementation of the best transformer. In *VMCAI*, pages 252–266, 2004.

[34] W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda. Model checking programs. *Autom. Softw. Eng.*, 10(2):203–232, 2003.

[35] C. Weidenbach. SPASS: An automated theorem prover for first-order logic with equality. Available at "http://spass.mpi-sb.mpg.de/index.html".

[36] G. Yorsh, T. Reps, and M. Sagiv. Symbolically computing most-precise abstract operations for shape analysis. In *TACAS*, 2004.

[37] H. Zhu, P.A. Hall, and H.R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):336–427, December 1997.

APPENDIX

A. LATTICE OPERATIONS

A lattice is a partially-ordered set closed under meet and join operations, defined as follows. Let $\mathcal{A}$ be a set with partial order $\sqsubseteq$. An element $a \in \mathcal{A}$ is a **lower bound** of a set $X \subseteq \mathcal{A}$ if, for every $x \in X$, $a \sqsubseteq x$. The **meet operator**, denoted by $\sqcap$, yields the greatest lower bound with respect to $\sqsubseteq$; i.e., for every set $X \subseteq \mathcal{A}$, $\sqcap X$ is a lower bound of X, and for every lower bound a of X, $a \sqsubseteq \sqcap X$. Similarly, an element $a \in \mathcal{A}$ is an **upper bound** of a set $X \subseteq \mathcal{A}$ if, for every $x \in X$, $x \sqsubseteq a$. Similarly, the **join operator**, denoted by $\sqcup$, yields the least upper bound with respect to $\sqsubseteq$; i.e., for every set $X \subseteq \mathcal{A}$, $\sqcap X$ is an upper bound of X, and for every upper bound a of X, $\sqcap X \sqsubseteq a$.

A widening operator on $\mathcal{A}$ is defined as a (partial) function $\bigtriangledown : \mathcal{A} \times \mathcal{A} \rightarrow \mathcal{A}$ satisfying: (i) for each $x, y \in \mathcal{A}$, $x \sqsubseteq x \bigtriangledown y$ and $y \sqsubseteq x \bigtriangledown y$; and (ii) for all increasing chains $y_0 \sqsubseteq y_1 \sqsubseteq \ldots$ the increasing chain defined by $x_0 \stackrel{\text{def}}{=} y_0$ and $x_{i+1} \stackrel{\text{def}}{=} x_i \bigtriangledown y_{i+1}$ is not strictly increasing.