Generating Concrete Counterexamples for Sound Abstract Interpretation

Guy Erez
guyerez@tau.ac.il

Eran Yahav
yahave@tau.ac.il

Mooly Sagiv
msagiv@tau.ac.il

School of Computer Science
Tel Aviv University, Tel-Aviv 69978, Israel.

ABSTRACT

Sound abstract interpretation has been successful in proving interesting properties of programs. Commercially available tools are now able to verify the absence of runtime errors in safety critical applications. The ability to verify the absence of errors comes from the fact that these tools use *conservative* methods, i.e., whenever the algorithm verifies a property, it is guaranteed to hold. However, the algorithm may produce false alarms, i.e., reports of errors that never occur. False alarms, which are a result of the abstraction and are unavoidable in general, make sound abstract interpretation hard to use.

In this paper, we develop a new algorithm that can be used to increase the usability of abstract interpretation tools by producing concrete counterexamples for the error messages reported. Our algorithm performs a limited search using a theorem prover. When the algorithm identifies a concrete input instance, it is guaranteed to be an input for which the program yields the reported error message. This allows the user to identify some of the reported messages as real errors, reducing the number of alarms that have to be manually investigated.

Our algorithm can also assist runtime testing by producing a set of inputs which covers the program according to a certain criteria. For example, we can use the algorithm to find an adequate set of inputs, or even a set of inputs that realizes all the results of the analysis.

Our algorithm is generic and applicable to many abstract domains, including polyhedra abstraction, predicate abstraction, and canonical abstraction used in shape analysis. We have implemented a prototype of our algorithm and used it to find counterexamples (and test cases) for several small but interesting example programs, including implementations of sorting algorithms.

1. INTRODUCTION

Sound abstract interpretation has been successful in proving interesting properties of programs. The main idea is to compute an over-approximation of the set of reachable program states [9]. This assures that the algorithm can verify the absence of runtime errors by checking the over-approximations.

Recently, commercial and academic tools using over-approximations have been successfully applied to verify the absence of runtime errors in safety critical applications (e.g., see [5, 14, 1, 2]).

While these tools guarantee that no errors are missed (no "false positives"), they are hard to use due to false alarms ("false negatives") which arise from overly conservative over-approximation. Due to the possibility of false alarms, each reported error has to be manually investigated to determine whether it is an actual error or a false alarm.

Bounded model checking (BMC) has been successfully used to locate errors in hardware and software systems [6, 28]. The basic idea of BMC is to search a counterexample only in executions up to a certain bounded length. Conceptually, bounded model checking computes an under-approximation of reachable program states and uses it to identify bugs. Thus, it may never produce false negatives but may produce false positives, which are intolerable in certain domains (e.g., safety critical code). In general, the undecidability of checking interesting program properties implies that no algorithm can avoid both false positives and false negatives.

In this paper, we describe a tool which allows the users of sound abstract interpretation to enjoy the benefits of both worlds by trying to instantiate a concrete input example for each error message reported by the abstract interpretation. Using this tool combined with a sound abstract interpretation guarantees the absence of false positives while reducing the number of alarms that have to be manually investigated.

Our tool can also assist runtime testing by automatically generating a set of inputs which covers the program according to a certain criteria. For example, we can use the tool to find an adequate [29, 8] set of inputs or even a set of inputs which realizes all the results of the analysis.

We have implemented a prototype of our tool, and applied it to several small but interesting benchmark programs, including implementations of various sorting algorithms [21] analyzed with the TVLA program analysis framework [22].

For the benchmark programs, our tool was able to identify non-trivial concrete input instances for error messages reported by the program analysis tool. It also produced a set of inputs that realizes the results of the program analysis tool.

The algorithm implemented by our tool could be viewed as a new algorithm for bounded model checking of data-intensive software. Bounded model checking exploits the strength of SAT-solvers to simultaneously check all program paths up to a certain depth. Our algorithm verifies each program path separately, enabling the use of large formulae as symbolic state descriptors.

Our approach is generic and is applicable to any abstract domain that satisfies some reasonable requirements (See Section 3.1). Such domains include polyhedra abstraction, predicate abstraction and canonical abstraction used in shape analysis.

1.1 Main Results and Related Work

The contributions of this paper can be summarized as follows:

- A new bounded model checking algorithm for a special class of programs and specifications is presented. The algorithm is designed to behave well on deterministic programs with complicated data such as dynamically allocated data-structures.
- We define reasonable requirements on the abstraction which allows bounded model checking to be used for producing concrete input examples.
- We show how to apply our bounded model checking tool to compute a set of concrete inputs which is adequate according to certain criteria.
- We have implemented a prototype of our algorithm and applied it to several small but interesting example programs, including implementations of sorting algorithms.

Our work is closely related to bounded model checking (BMC) [6, 4], and could be viewed as a new algorithm for BMC which separates path-enumeration from path-verification. BMC operates by constructing a single propositional formula for a program and a temporal logic specification such that the formula is satisfiable only when there exists a program path of up to some bounded length that satisfies the property. An assignment to this formula is both a selection of a program path and a selection of values for state variables. In contrast, our algorithm iterates through the possible paths and constructs a separate formula for each path resulting in a smaller formula. This enables us to use richer (and possibly larger) symbolic state representations.

Jackson and Vaziri [18] present a method for verifying structural properties, such as heap relationships, using a constraint solver. Their method consists of three stages: (i) translation of the specification code into first-order logic; (ii) translation from first-order logic to propositional logic using a bound on the number of objects; (iii) running SAT solver to find counterexamples. In the translation, both the control and data are encoded, thus a counterexample describes both the initial configuration and a path to the violating label. This analysis produces concrete counterexamples and no false alarms, but only performs a bounded exploration and therefore may produce false positives.

In [28], Jackson and Vaziri present various optimization techniques (reducing functional representation size and applying logical simplifications) to reduce the complexity of the resulting logical formula and increasing the tool scalability.

Our method differ from [18, 28] in that our method is applied on the results of a sound abstract interpretation, and therefore can guarantee that there are no false positives. Our method helps reduce the number of false alarms needed to be manually investigated. In addition, we take a different approach for reducing formula complexity. Instead of enumerating all possible paths and data states at once, we verify every path separately thus allowing more complex state description formulae. This comes at the expense of more calls to the theorem prover. Our experience shows that this approach works well for the abstract domain tested.

Microsoft's SLAM [25] implements a process for validating temporal safety properties on software that uses a well defined interface. The SLAM process includes: (i) generating a boolean predicate representation of the C program (C2BP); (ii) using a model checker to find if error states are reachable in the abstract program (BEBOP); (iii) if no reachable error state is found, the property is verified. If a reachable error state is found, NEWTON path simulator is used on the generated abstract trace to check if there exists a directly corresponding concrete trace in the original program. If such corresponding trace does not exist, a new predicate is added to refine the abstraction [7].

The SLAM process resembles ours in that it is sound (does not produce false positives), and can therefore prove the absence of errors. It also has the advantage of refining the abstraction in case a counterexample is not found. However, while NEWTON path simulator only checks a single path through the program leading to the error state, our algorithm checks all paths up to a certain bounded length. Thus, in some cases our algorithm will produce a counterexample when the iterative refinement will require a refinement step. In addition, SLAM uses predicate abstraction, while our algorithm can be applied to many other abstract domains.

Pasareanu et. al. [26] present two techniques for checking the feasibility of a reported abstract counterexample for multithreaded Java programs. Viewing operations on abstract values as being deterministic (returning a single abstract value) or non-deterministic (returning a set of abstract values), the first techniques tries to find program paths in which non-deterministic operations do not occur (*choose-free* paths). Since a choose-free path is guaranteed to be a path in the concrete program, if such path is found, it provides a feasible counterexample. It is interesting to note that our technique applies in many cases in which the operations over the abstract domain are not deterministic. Of course, our technique is admittedly costly due to path-enumeration and the use of a theorem prover. The second technique proposed in [26] performs a counterexample guided simulation similar to NEWTON, but in a setting that allows non-boolean abstractions and handles multithreaded programs.

1.2 A Motivating Example

Figure 1 shows an erroneous implementation of the bubble-sort sorting algorithm. In this program, the assignment at label l_{20} erroneously assigns `y` to `p.n` instead of assigning `y.n`. This erroneous assignment causes the implementation to lose list items during execution. For example, the input of Figure 2 causes a list item to be lost, violating the assertion at label l_{31}.

This bug is tricky. It only occurs in certain cases when the algorithm is applied to an unsorted list.

To guarantee the absence of errors in such programs, one may use a sound abstract interpretation framework such as TVLA [21]. TVLA is a static analysis engine that allows generation of sound program analyses from specifications of a concrete operational semantics. Applying TVLA to the example program, an error is indeed reported at label l_{31}, indicating the possibility that some of the original nodes in the list do not appear in the sorted list. The problem is to determine whether this error report is a false negative or an error that could occur in practice. For the bubble-sort program, our algorithm is able to determine that the error reported by the static analysis is indeed a real error and produce a concrete input for which the error occurs.

1.3 Paper Outline

The rest of this paper is organized as follows. In Section 2, we define the basic terms used throughout the paper. In Section 3, we present an algorithm `Input-Instance` for generating input instances for a program point and a condition. To simplify the presentation, the algorithms are first presented using the simple domain of constant propagation. Section 4 shows how to use the algorithm `Input-Instance` for generating counterexamples for errors reported by abstract interpretation. In Section 5, we show how to

```
   public static ListItem bugSort(ListItem x) {
l1     recordListNodes(x);
l2     boolean change = true;
l3     ListItem p, yn, t, y, head;
l4     if (x == null)
l5        return null;
l6     while (change) {
l7        p = null;
l8        change = false;
l9        y = x;
l10       yn = y.n;
l11       while (yn != null) {
l12          if (y.data > yn.data) { // swap y and yn
l13             change = true;
l14             t = yn.n;
l15             y.n = t;
l16             yn.n = y;
l17             if (p == null) {
l18                x = yn;
l19             } else {
l20                p.n = y; //BUG: correct code is p.n=y.n
l21             }
l22             p = yn;
l23             yn = t;
l24          } else {
l25             p = y;
l26             y = yn;
l27             yn = y.n;
l28          }
l29       }
l30    }
l31    assert permutation(x);
l32    assert ascendingOrder(x);
l33    return x;
   }
```

Figure 1: Erroneous Java implementation of bubble-sort.

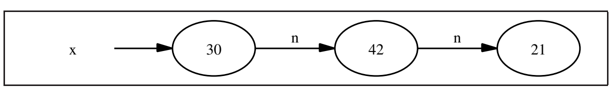

Figure 2: An input causing a violation of the assertion at l_{31}.

apply the algorithms to the domain of shape analysis and how our tool produces a concrete counterexample for the bubble-sort implementation. Section 6 describes our prototype implementation and empirical results. Section 7 shows extensions to our algorithms and an algorithm for generating a set of test cases realizing analysis results in a program point. Finally, we conclude in Section 8.

2. PRELIMINARIES

In this section we define the basic terms used throughout the paper.

A program path is a sequence of program labels starting from an initial ($entry$) label and proceeding such that each label is a successor of its preceding label in the program control flow graph.

DEFINITION 2.1 (PATH). *A path $l_0 \rightarrow l_1 \ldots \rightarrow l_n$ is a sequence of program labels starting with an initial label l_0 and proceeding such that for every $0 \leq i < n$, a label l_{i+1} is an immediate successor of its preceding label l_i in the program control flow graph.*

A labelled program state $\langle \sigma, l \rangle$ consists of a state σ and a program label l. We define an execution of the program to be a sequence of labelled states starting from an initial labelled state and proceeding such that each state is derived by application of a single statement to its preceding state. We label the transitions between states of the sequence with the corresponding statement.

```
l1  y = 1;
l2  x = 1;
l3  while (y < z) {
l4     if (x >= y) {
l5        x = y + 2;
l6     } else {
l7        x = 3;
l8     };
l9     y = y + 1;
l10 }
l11 assert x < 4;
```

Figure 3: A simple program demonstrating constant propagation. z is an input parameter.

DEFINITION 2.2 (EXECUTION PATH). *An* execution path *$\langle \sigma_0, l_0 \rangle \xrightarrow{st_0} \langle \sigma_1, l_1 \rangle \ldots \xrightarrow{st_{n-1}} \langle \sigma_n, l_n \rangle$ is a sequence of labelled states starting with an initial labelled state $\langle \sigma_0, l_0 \rangle$ and proceeding such that for every $0 \leq i < n$, a labelled state $\langle \sigma_{i+1}, l_{i+1} \rangle$ is derived from its preceding labelled state $\langle \sigma_i, l_i \rangle$ by application of a single statement labelled by st_i.*

In this paper, we are interested in finding initial states for which there exists an execution path reaching a given program point with a state satisfying a certain condition. In the sequel we assume that the condition is specified as a formula in some underlying logic $\mathcal{L}$, e.g., propositional logic or first-order logic.

DEFINITION 2.3 (INPUT INSTANCE). *Given a program point l, and a condition formula φ, an* input instance *is an initial state σ_0 for which there exists an execution path $\langle \sigma_0, l_0 \rangle \xrightarrow{st_0} \langle \sigma_1, l_1 \rangle \ldots \xrightarrow{st_{n-1}} \langle \sigma_n, l_n \rangle$ such that $l_0 = entry$, $l_n = l$ and $\sigma_n \models \varphi$.*

To demonstrate the basic concepts presented in this section, for methodological reasons, we consider the simple domain of constant propagation[20] for programs with integer variables.

For these programs, a program state σ is defined as a mapping of program variables into integer values, i.e., $\sigma \colon Var \rightarrow \mathbb{Z}$.

EXAMPLE 2.4. *Consider the simple example program of Figure 3, the state $\sigma = [z \mapsto 2]$ is an input instance for the label l_{11} and condition $x < 4$, because there exists an execution path*

$\langle [z \mapsto 2], l_1 \rangle \xrightarrow{y=1} \langle [z \mapsto 2, y \mapsto 1], l_2 \rangle \xrightarrow{x=1}$
$\langle [z \mapsto 2, y \mapsto 1, x \mapsto 1], l_3 \rangle \xrightarrow{while(y<z)}$
$\langle [z \mapsto 2, y \mapsto 1, x \mapsto 1], l_4 \rangle \xrightarrow{if(x>=y)}$
$\langle [z \mapsto 2, y \mapsto 1, x \mapsto 1], l_5 \rangle \xrightarrow{x=y+2}$
$\langle [z \mapsto 2, y \mapsto 1, x \mapsto 3], l_9 \rangle \xrightarrow{y=y+1}$
$\langle [z \mapsto 2, y \mapsto 2, x \mapsto 3], l_3 \rangle \xrightarrow{while(y<z)}$
$\langle [z \mapsto 2, y \mapsto 2, x \mapsto 3], l_{11} \rangle$

reaching l_{11} and satisfying $x < 4$.

Direct computation of input instances requires computing the set of reachable program states, which is infeasible for many programs.

Abstract interpretation computes a set of abstract states conservatively representing possible program states at a given program point. Given a domain of abstract states, a concretization mapping

γ is defined such that for each abstract state $\sigma^\sharp$, $\gamma(\sigma^\sharp)$ is the (possibly infinite) set of concrete program states represented by s. An abstract interpretation algorithm is *sound* if it produces for every program label l, an abstract state $\sigma^\sharp$ such that $\gamma(\sigma^\sharp)$ contains all the concrete states reachable at label l.

Our technique requires a symbolic representation of the concrete states represented by an abstract state. We therefore require our abstract domain to be equipped with a symbolic concretization function $\hat{\gamma}$ mapping an abstract state to a logical formula. This formula should describe exactly all the concrete states that are represented by the abstract state, i.e., a state $\sigma \in \gamma(\sigma^\sharp) \iff \sigma \models \hat{\gamma}(\sigma^\sharp)$. We assume that the symbolic concretization could be expressed in some logic $\mathcal{L}$, usually first-order logic.

We demonstrate on constant propagation. The abstract domain used for constant propagation is an infinite lattice with a finite-height containing all integer values, and the values $\top$ and $\bot$. $\top$ represents any integer value and $\bot$ represents an infeasible value. An abstract state is a partial mapping from variables to values in the constant propagation lattice. That is, $\sigma^\sharp \colon Var \to (\mathbb{Z} \cup \{\top, \bot\})$.

We define the symbolic domain to contain formulae of propositional logic with integer arithmetic.

A symbolic concretization function $\hat{\gamma}(\sigma^\sharp)$ transforms an abstract state into its symbolic representation as follows:

$$\hat{\gamma}_{v_i}(\sigma^\sharp) = \begin{cases} true, & \sigma^\sharp(v_i) = \top; \\ false, & \sigma^\sharp(v_i) = \bot; \\ (v_i = \sigma^\sharp(v_i)), & \sigma^\sharp(v_i) \in \mathbb{Z}. \end{cases}$$

$$\hat{\gamma}(\sigma^\sharp) = \textstyle\bigwedge_{v_i \in Var} \hat{\gamma}_{v_i}(\sigma^\sharp)$$

EXAMPLE 2.5. *The abstract state* $\sigma^\sharp = [x \mapsto \top, y \mapsto 3, z \mapsto 3]$ *represents an infinite number of concrete states where* x *can have an arbitrary integer value,* y *has the value* 3*, and* z *has the value* 3*. The (finite) symbolic representation of* $\sigma^\sharp$ *is therefore* $\hat{\gamma}(\sigma^\sharp) = (y = 3) \wedge (z = 3)$.

As another example, consider an abstract domain that is based on predicate abstraction [15, 3]. Predicate abstraction domains are based on a finite vocabulary $V = \{B_1, \ldots, B_k\}$ of predicate symbols, each associated with a defining formula, i.e., $B_i \triangleq \varphi_i$ for every $1 \leq i \leq k$. An abstract state of a predicate abstraction domain is a mapping of the predicates in V to their truth values, i.e., $\sigma^\sharp \colon V \to \{0, 1\}$. The symbolic concretization $\hat{\gamma}$ of an abstract state $\sigma^\sharp$ is a conjunction of predicate defining formulae as follows:

$$\hat{\gamma}(\sigma^\sharp) = \bigwedge_{\sigma^\sharp(B_i)=1, 1 \leq i \leq k} \varphi_i \wedge \bigwedge_{\sigma^\sharp(B_j)=0, 1 \leq j \leq k} \neg\varphi_j$$

The symbolic concretization assumption is met by a wide range of existing abstract domains. Table 1 shows a number of widely used abstract domains and their corresponding symbolic concretization mappings.

3. GENERATING INPUT INSTANCES

In this section we describe the algorithm `Input-Instance` for generating an input instance for a label l and a condition φ. The algorithm uses the results of the static analysis as the starting point for a bounded exploration.

The algorithm could be viewed as a new algorithm for bounded model checking in which path-enumeration is separated from path-verification. This allows us to use larger formulae as state descriptors.

The algorithm is bounded by the maximum length of paths to explore p and by the specific bounds, if any, of the theorem prover used. When no input instance is found, one can increase these parameters until either an input instance is found, or the problem becomes practically intractable.

Abstract Domain	Symbolic Concretization
Predicate Abstraction ([15, 3])	conjunction of predicate-defining formulae
Polyhedra Abstraction ([10])	system of linear inequalities
Canonical Abstraction ([27])	first order formula with transitive closure (See [30])
Constant Propagation ([20])	conjunction of variable value equalities

Table 1: Abstract domains and their corresponding symbolic concretization mappings.

3.1 Algorithm Prerequisites

`Input-Instance` has the following requirements:

- symbolic concretization — the abstraction has to be accompanied by the ability to compute a symbolic concretization expressed in some logic $\mathcal{L}$.
- weakest precondition computation — $\mathcal{L}$ should provide the ability to compute the weakest precondition (See Section 3.2.2) of every atomic program statement. This requirement holds for first-order logic with any arbitrary atomic statement.
- finite counterexample generation — a theorem prover that is able to produce a finite counterexample for the validity of formulae in $\mathcal{L}$.

3.2 Algorithm Description

The outline of the algorithm is described in Figure 4.

```
INPUT-INSTANCE(φ_l, l)
path ← GET-NEXT-PATH(l)
while |path| ≤ p
do
   φ_0 ← PATH-WEAKEST-PRECONDITION(φ_l, path)
   model ← THEOREM-PROVER(¬φ_0)
   if model ≠ null
      then return model
   path ← GET-NEXT-PATH(l)
return null
```

Figure 4: Input-Instance. The algorithm assumes the following global parameters are set: exploration depth p and transition system ts.

The input to `Input-Instance` is the condition φ_l, and the label l. The algorithm assumes that the following values are provided as global variables: the exploration depth (p) and the program's transition system (ts).

The output is an input instance for label l and condition φ_l or $null$, if no such input was found.

The algorithm consists of the following stages:

- Path Generation — create paths leading from program entry to the point of interest l. Paths are generated one by one, starting from the shortest and continuing in an ascending order of length. The maximum length of paths is bounded by the parameter p to guarantee termination.

- Backward Symbolic Exploration (Weakest Precondition) — for each path, compute a formula φ_0 describing input states that guarantee that φ is satisfied at point l when the program follows this path. Technically, we use a repeated computation of weakest precondition to compute the formula φ_0.

- Model Generation — given the formula φ_0 describing a (possibly infinite) set of input instances for φ_l at l, we try to find a concrete state (model) that satisfies the formula. We assume the availability of a theorem prover that is able to produce a finite counterexample. In order to obtain a model for φ_0 we use the theorem prover to find a counterexample for the validity of $\neg\varphi_0$. If a model is found, it is an input instance for φ_l at l. To guarantee termination of the theorem prover, we usually have to provide bounds for the theorem prover (for instance, with first-order logic theorem prover, it is usually the maximum model size). If a model is found, we return it as an input instance and stop the search. If a model is not found, we return $null$.

3.2.1 Path Generation

We use a simple BFS starting from l and going backward on the transition system until program entry is reached. Obviously, if the transition system has loops in it, there is an infinite number of execution paths leading to l, therefore some halting criterion is required. For usability purposes, we prefer a global bound, the maximum length of the paths generated, rather than specifying for each loop the number of iterations to unwind.

Since `Input-Instance` stops once a path yields a model and given the possibly large amount of paths whose length $\leq p$, it is more efficient to generate paths on-the-fly, meaning we generate the next path only if no input instance was found for the previous path. Paths are generated in increasing length, starting with the shortest path.

EXAMPLE 3.1. *In the program of Figure 3 there is an infinite number of possible paths for label l_{11}. For a maximum length of 8 we get the following paths:*
$l_1 \rightarrow l_2 \rightarrow l_3 \rightarrow l_{11}$
$l_1 \rightarrow l_2 \rightarrow l_3 \rightarrow l_4 \rightarrow l_5 \rightarrow l_9 \rightarrow l_3 \rightarrow l_{11}$
$l_1 \rightarrow l_2 \rightarrow l_3 \rightarrow l_4 \rightarrow l_7 \rightarrow l_9 \rightarrow l_3 \rightarrow l_{11}$
In the bubble-sort program of Figure 1 there are 335 possible paths for a maximum length of 50.

3.2.2 Weakest Precondition

To calculate φ_0 for a given path we use a repeated computation of the weakest precondition [12].

Given a formula φ and a statement `st`, we denote by WP(`st`, φ) the weakest precondition of φ with respect to `st`. That is, $\psi =$ WP(`st`, φ) is the weakest formula for which any state that satisfies ψ before execution of `st`, is guaranteed to satisfy φ after application of `st`.

Given a path $l_0 \rightarrow l_1 \rightarrow l_2 \rightarrow \ldots \rightarrow l_n$, and an initial formula φ_n, we compute a sequence of formulae $\varphi_{n-1}, \ldots, \varphi_0$ by repeated application of WP. That is, for every $0 \leq i < n$, $\varphi_i = \mathrm{WP}(st_i, \varphi_{i+1})$ where st_i is the statement reaching from l_i to l_{i+1}.

As mentioned earlier, one of the requirements of our algorithm is that WP could be computed for every statement used in the program.

Table 2 shows how to compute the weakest precondition for the statements of the simple constant propagation program we introduced before. Section 5.3 shows how to compute the weakest precondition in a shape analysis domain, such as the one used for analyzing the bubble-sort program.

statement	WP(**st**, φ)
`x = expr`	$\varphi[expr/x]$
`assume expr`	$\varphi \wedge expr$

Table 2: Calculating weakest precondition for the statements used in the constant propagation program of Figure 3.

i	label	statement	φ_i
6	l_3	`while (y < z)`	$x \geq 4 \wedge y \geq z$
5	l_9	`y = y + 1`	$x \geq 4 \wedge (y+1) \geq z$
4	l_5	`x = y + 2`	$(y+2) \geq 4 \wedge (y+1) \geq z$
3	l_4	`if (x >= y)`	$(y+2) \geq 4 \wedge (y+1) \geq z$ $\wedge x \geq y$
2	l_3	`while (y < z)`	$(y+2) \geq 4 \wedge (y+1) \geq z$ $\wedge y < z \wedge x \geq y$
1	l_2	`x = 1`	$(y+2) \geq 4 \wedge (y+1) \geq z$ $\wedge y < z \wedge 1 \geq y$
0	l_1	`y = 1`	$(1+2) \geq 4 \wedge (1+1) \geq z$ $\wedge 1 < z \wedge 1 \geq 1$

Table 3: Results of WP computation along the example path.

For convenience, we translate program conditionals to include the appropriate `assume` statements. That is, given a conditional statement `if (c) st_t else st_f`, we augment it with the appropriate `assume` statements resulting with `if (c) {assume c; st_t } else { assume !c; st_f }`.

EXAMPLE 3.2. *Given the path $l_1 \rightarrow l_2 \rightarrow l_3 \rightarrow l_4 \rightarrow l_5 \rightarrow l_9 \rightarrow l_3 \rightarrow l_{11}$ of length 8, and the initial formula $\varphi_7 = x \geq 4$ at label l_{11}, Table 3 shows the calculation of φ_0.*

3.2.3 Model Generation

Given the formula φ_0 describing a (possibly infinite) set of input instances for φ_l at l, we try to find concrete states (models) that satisfy the formula.

The theorem prover should be capable of finding a finite satisfying model for the formula φ_0. Any theorem prover that can produce a finite counter example (e.g., [17]) could be used to find satisfying models (counterexamples for $\neg\varphi_0$). Our method could be also used with theorem provers that produce symbolic counterexamples (e.g., [11]).

EXAMPLE 3.3. *Given $\varphi_0 = (1+2) \geq 4 \wedge (1+1) \geq z \wedge 1 < z \wedge 1 \geq 1$ from Table 3, the theorem prover Simplify [11] correctly determines that φ_0 does not have a satisfying model (finds that $\neg\varphi_0$ is valid).*

4. GENERATING COUNTEREXAMPLES

One of the main problems in using sound abstract interpretation is the possibility of (conservative) error reports that do not correspond to real errors in the program (false alarms). Manual investigation of each reported error to determine whether it is a real error or a false alarm is a time-consuming process and may deter users from using abstract interpretation.

Given the results of a sound abstract interpretation, and considering a single abstract state $\sigma^\sharp$ satisfying an error condition φ_{err} at a program point l, it may be the case that there is no program execution that produces this violation, and the error report is therefore a false alarm.

However, if we can find an input instance σ_0 for φ_{err} at point l such that it reaches point l with a concrete state represented by $\sigma^\sharp$, this would be a concrete example that establishes the error report as a real error. More formally,

DEFINITION 4.1 (COUNTEREXAMPLE). *Given an abstract state $\sigma^\sharp$ that satisfies an error condition φ_{err} at label l, we say that a concrete state σ_0 is a counterexample when there exists a concrete execution path $\langle \sigma_0, l_0 \rangle \xrightarrow{st_0} \langle \sigma_1, l_1 \rangle \ldots \xrightarrow{st_{n-1}} \langle \sigma_n, l_n \rangle$ such that:*

1. *$l_n = l$ and $\sigma_n \models \varphi_{err}$ (i.e., σ_0 is an input instance for φ_{err} at label l).*
2. *$\sigma_n \in \gamma(\sigma^\sharp)$*

The tool presented in this section examines each error reported by the analysis and tries to produce a concrete counterexample for it. If it succeeds, it is guaranteed that the error is indeed a true error, and there is no need to manually investigate it. However, if the tool fails to find a counterexample, it could still be the case that the error is a real program error and a counterexample was not found due to the bounded exploration.

Increasing the algorithm bound (exploration depth) or theorem prover bounds (i.e. model size when applicable) may lead to the discovery of more concrete counterexamples at the expense of increased runtime and memory requirements. In our experiments, all counterexamples were found at rather low bounds.

GENERATE-COUNTEREXAMPLES()
for each $(l, \sigma^\sharp, \varphi_{err})$ **in** `Analysis-Errors`
do
  `counter` $\leftarrow$ INPUT-INSTANCE($\hat{\gamma}(\sigma^\sharp) \wedge \varphi_{err}, l$)
  if `counter` $\neq$ *null*
    then REPORT-TRUE-ALARM($(l, \sigma^\sharp, \varphi_{err})$, `counter`)

Figure 5: Generate-Counterexamples. The algorithm assumes the following global parameters are set: exploration depth p and transition system ts.

The algorithm for generating counterexamples is shown in Figure 5. The algorithm is generic and can be used with any abstract interpretation tool given an interface that provides access to analysis results. The algorithm requires access to the errors reported by the analysis where each error consists of: (i) the label l at which the error was reported (ii) the abstract state $\sigma^\sharp$ in which the error occurred; (iii) the error condition φ_{err}.

For every error reported the algorithm uses `Input-Instance` to find an input instance that satisfies the $\varphi = \hat{\gamma}(\sigma^\sharp) \wedge \varphi_{err}$ at label l. An input instance satisfying φ is guaranteed to be a counter example according Definition 4.1 because it will reach l with a concrete state σ_n that (i) satisfies φ_{err} and (ii) since it satisfies $\hat{\gamma}(\sigma^\sharp)$ it implies $\sigma_n \in \gamma(\sigma^\sharp)$.

EXAMPLE 4.2. *Consider the program of Figure 3 in which z is given as user input. Using constant propagation in an attempt to verify the assertion at label l_{11} will produce an error since the analysis will reach the abstract state $[x \mapsto \top, y \mapsto \top, z \mapsto \top]$ at this program label. This abstract state represents any concrete state in which the variables x, y, and z have been assigned a value, including states in which $x \geq 4$ that satisfy φ_{err}.*

Table 4 describes the running of the counterexample algorithm with a maximum path length of 14. The counterexample is found on the 4^{th} path checked.

Predicate	Description
$x(u)$	Is u pointed-to by x?
$n(u_1, u_2)$	Does the field n of u_1 point to u_2?
$dle(u_1, u_2)$	Is the data of u_1 less-than or equal to the data of u_2?
$r[n, x](u)$	Is u transitively reachable from x using n field?
$inOrder(u)$	Is u part of a non-decreasing list fragment?

Table 5: Predicates used to define states.

Generally, some programs require a larger number of paths to be verified, for example, in the bubble-sort program of Figure 1 the counterexample to the error at label l_{31} is found on the 143^{th} path checked.

5. APPLYING TO SHAPE ANALYSIS

In this section, we demonstrate the application of our techniques to shape analysis [19, 27]. Shape analysis will allow us to find the bug in the erroneous bubble-sort program of Figure 1 and — using `GenerateCounterexamples` algorithm (Figure 5)— to produce a counter example for it.

In Section 5.1 we give some background on the concrete and abstract domains used in the shape analysis framework of [27]. Then, in Section 5.2 we show how to answer the first requirement of our algorithm and compute the symbolic concretization for shape analysis. Section 5.3 shows how to answer the second requirement of the algorithm and provides a way to compute the weakest precondition for this symbolic domain. Finally, in Section 5.4 we show how to use these ingredients to successfully produce a concrete counterexample for the bubble-sort program.

5.1 Concrete and Abstract Domains for Shape Analysis

In [27], it is shown how a global state of the program can be naturally expressed as a first-order logical structure in which each individual corresponds to a heap-allocated object and predicates of the structure correspond to properties of heap-allocated objects. We use the predicates of Table 5 to record information used by the properties discussed in this paper.

For each reference variable x, we define a unary predicate $x(u)$. The value of $x(u)$ is 1 if the variable x points to the list element represented by u.

Reference fields are represented using binary predicates. For example, the `n` field of a `ListElement` is represented using a predicate $n(u_1, u_2)$ that holds when the `n` field of u_1 points to u_2. Similarly, we use the unary predicate $dle(u_1, u_2)$ to record inequalities between data values of the list elements. The predicate $dle(u_1, u_2)$ holds when the value of the `data` component of u_1 is less than or equal to the value of the `data` of u_2.

The unary predicate $r[n, x](u)$ holds for list elements that are (transitively) reachable from program variable x, possibly using a sequence of `n` fields. This predicate is an instrumentation predicate [27] which is used to refine the abstraction.

Finally, to express sortedness of lists we use the (instrumentation) predicate $inOrder(u)$. The predicate $inOrder(u)$ holds for a node u whose `data` field is less than or equal to the `data` field of its `n`-successor (if one exists).

We depict program states as directed graphs. Each individual of the universe is displayed as a node. A unary predicate $p(u)$ which holds for an individual (node) u is drawn inside the node u. Predicates that can only hold for a single individual (e.g., representing a value of a reference variable) are shown as an edge from the predicate symbol to the node in which it holds. A binary predicate

step	path	φ_0	model
1	$l_1 \to l_2 \to l_3 \to l_{11}$	$1 \geq 4 \wedge 1 \geq z$	no model
2	$l_1 \to l_2 \to l_3 \to l_4 \to l_5 \to l_9 \to l_3 \to l_{11}$	$3 \geq 4 \wedge 2 \geq z \wedge 1 \geq 1 \wedge 1 < z$	no model
3	$l_1 \to l_2 \to l_3 \to l_4 \to l_7 \to l_9 \to l_3 \to l_{11}$	$3 \geq 4 \wedge 2 \geq z \wedge 1 < 1 \wedge 1 < z$	no model
4	$l_1 \to l_2 \to l_3 \to l_4 \to l_5 \to l_9 \to l_3 \to l_4 \to l_5 \to l_9 \to l_3 \to l_{11}$	$1 \geq 1 \wedge 3 \geq z \wedge 2 \geq 1 \wedge 2 < z \wedge 1 < z$	$[z \mapsto 3]$

Table 4: Generating a counterexample for the program in Figure 3 for the label l_{11} and $\varphi_{err} = x \geq 4$.

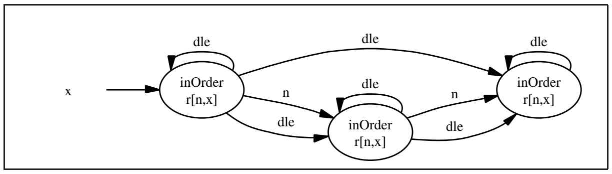

Figure 6: A concrete representation of a sorted linked list.

$p(u_1, u_2)$ which evaluates to 1 is drawn as directed edge from u_1 to u_2 labelled with the predicate symbol.

EXAMPLE 5.1. *The state shown in Figure 6 corresponds to a global state of the program containing a sorted linked list of length 3, and pointed to by the variable* x. *Note how the $dle(u_1, u_2)$ binary predicate records the ordering relation between individuals, and how $inOrder(u)$ holds for all nodes as a result of the list being sorted. Also note that for all elements of this list r[n,x](u) holds since all elements are (transitively) reachable from* x *using a sequence of* n *fields.*

In order to guarantee a finite representation, we conservatively represent multiple concrete program states using a single logical structure with an extra truth-value $1/2$ which denotes values which may be 1 or may be 0.

We allow an abstract state to include a *summary node*, i.e., an individual which corresponds to one or more individuals in a concrete state represented by that abstract state. Technically, we use a designated unary predicate *sm* to maintain summary-node information. A summary node u has $sm(u) = 1/2$, indicating that it may represent more than one node.

To abstract a concrete state, we use *canonical abstraction*. Canonical abstraction maps concrete individuals to an abstract individual based on the values of the individuals' unary predicates. All individuals having the same values for unary predicate symbols are mapped to the same abstract individual. Only summary nodes (i.e., nodes with $sm(u) = 1/2$) can have more than one node mapped to them by the abstraction.

Abstract program states are depicted by enhancing the representation of concrete states with a graphical representation for $1/2$ values: a binary predicate $p(u_1, u_2)$ which evaluates to $1/2$ is drawn as dashed directed edge from u_1 to u_2 labelled with the predicate symbol, and a summary node is drawn as circle with double-line boundaries.

EXAMPLE 5.2. *The abstract state shown in Figure 7 represents the concrete state of Figure 6. Note that this abstract state (finitely) represents infinitely many concrete states. For example, it represents any state containing a sorted linked-list of length at least 2, which is pointed to be* x. *The fact that $r[n, x](u)$ holds for the summary node means that all list elements represented by the summary node are reachable from* x. *Similarly, the fact that $inOrder(u)$ holds for the summary node means that the list suffix represented by the summary node is sorted. The solid (1-valued) $dle(u_1, u_2)$*

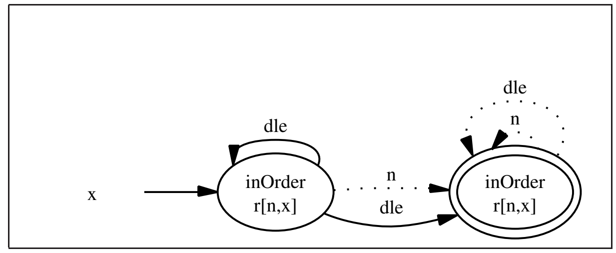

Figure 7: An abstract representation of a sorted linked list.

edge from the first node to the summary node records the fact that the data *element of the first node is less than (or equal to) the* data *element of the rest of the list items.*

5.2 Symbolic Concretization

To compute the symbolic concretization of an abstract state we use the procedure of [30]. The symbolic concretization of an abstract state is a formula in first-order logic with transitive closure (FO^{TC}). The procedure of [30] produces a formula which is linear in the size of the 3-valued structure.

EXAMPLE 5.3. *The formula*

$$\begin{aligned}
&\exists v_1.node_1(v_1) \wedge \exists v_2.node_2(v_2)\\
&\wedge \forall v.node_1(v) \vee node_2(v)\\
&\wedge \forall v.node_1(v) \iff x(v) \wedge r[n,x](v) \wedge inOrder(v)\\
&\wedge \forall v.node_2(v) \iff \neg x(v) \wedge r[n,x](v) \wedge inOrder(v)\\
&\wedge \forall v_1, v_2.node_2(v_2) \wedge node_1(v_1) \implies dle(v_2, v_1)\\
&\wedge \forall v_1, v_2.node_1(v_2) \wedge node_2(v_1) \implies \neg dle(v_2, v_1) \wedge \neg n(v_2, v_1)\\
&\wedge \forall v_1, v_2.node_2(v_2) \wedge node_2(v_1) \implies dle(v_2, v_1) \wedge \neg n(v_2, v_1)\\
&\wedge \forall v_1, v_2.node_2(v_1) \wedge node_2(v_2) \implies v_1 = v_2\\
&\wedge \varphi_{hygiene}
\end{aligned}$$

is the symbolic concretization of the abstract state of Figure 7. Intuitively, $node_1(u)$ represent concrete nodes pointed to by x *and $node_2(u)$ represents concrete nodes not pointed to directly by* x *but are reachable from* x *and are sorted in an ascending order.*

$\varphi_{hygiene}$ is a conjunction of hygiene conditions which guarantee that the represented first-order structures correspond to legitimate heap states. These hygiene conditions require for example that a reference variable points to at most a single object, that a reference field points at most to a single object, etc. In the formula mentioned above, $\varphi_{hygiene}$ will guarantee that at most one node is pointed-to by x.

5.3 Weakest Precondition

For the domain of shape analysis, we use formulae of first-order logic with transitive logic (FO^{TC}) to symbolically represent program states.

We assume that a statement is represented using a precondition formula and a set of update formulae. The weakest precondition of a formula φ with respect to a given statement st, denoted by

statement	update formulae
`x = null`	$x(v) = 0$
`x = y`	$x(v) = y(v)$
`x = y.n`	$x(v) = \exists v_1 : y(v_1) \wedge n(v_1, v)$
`x.n = null`	$n(v_1, v_2) = n(v_1, v_2) \wedge \neg x(v_1)$
`x.n = y` (assuming `x.n==null`)	$n(v_1, v_2) = n(v_1, v_2) \vee (x(v_1) \wedge y(v_2))$

Table 6: Predicate-update formulae for list manipulation statements.

WP(`st`, φ) can be then defined via backward substitution of the update formulae and conjoining the action's precondition.

Table 6 lists the predicate update formulae for the list manipulation statements used in this paper. Predicates not assigned a predicate-update formulae in a statement are assumed to maintain their value. Allocation of new nodes can also be modeled, but it is not used here.

Using this expressive logic allows us to easily compute the weakest precondition even for statements that perform destructive updates of heap references, as shown in the following example.

EXAMPLE 5.4. *Consider the formula* $\varphi = \exists v_1, v_2.x(v_1) \wedge n(v_1, v_2)$ *that requires the existence of an object reference by* `x` *and an object referenced by its* `n` *field.*

The weakest precondition of φ *with respect to the assignment statement* `x=y` *is* WP(`x=y`, φ) $= \exists v_1, v_2.y(v_1) \wedge n(v_1, v_2)$.

More interestingly, even for a statement performing destructive update such as `x.n = y`*,* WP *can be computed using simple backward substitution of the predicate update formulae. For this statement,* WP(`x.n=y`, φ) $= \exists v_1, v_2.x(v_1) \wedge (n(v_1, v_2) \vee x(v_1) \wedge y(v_2))$ [1].

5.4 Generating Counterexamples

We can now use the algorithm of Section 4 to find a concrete counterexample for the bubble-sort running example.

The bubble-sort procedure has two assertions as postconditions. These assertions require that: (i) the resulting list is sorted in ascending order (`assert ascendingOrder(x)`); (ii) the resulting list is a permutation of the input list, i.e., no nodes were lost (`assert permutation(x)`).

These assertions are formulated in TVLA using the following FO^{TC} formulae

$$\Phi_{ascending} = \forall v.r[n, x](v) \implies inOrder(v)$$

$$\Phi_{permutation} = \forall v.r[n, x](v) \iff or[n, x](v)$$

The specification for $\Phi_{permutation}$ uses an auxiliary predicate $or[n, x](v)$ that records the nodes that were reachable from `x` on $entry$ to the sorting procedure. The auxiliary predicate $or[n, x]$ is only updated once by the call to `recordListNodes(x)` at line l_1.

Running TVLA using the negation of these formulae as error conditions, an error is reported at label l_{31} with the abstract state shown in Figure 8. Note that for the rightmost node in the figure, $r[n, x]$ is $false$ while $or[n, x]$ is $true$, indicating that the node was previously reachable from x and is no longer be reachable, i.e. is lost. Also note that $inOrder(u)$ holds for all list elements, meaning that the resulting list is sorted (although it loses list elements).

[1]this assumes that x.n is reset to null before it is given a new value, otherwise, the update formula would have been $((n(v_1, v_2) \wedge \neg x(v_1)) \vee (x(v_1) \wedge y(v_2)))$].

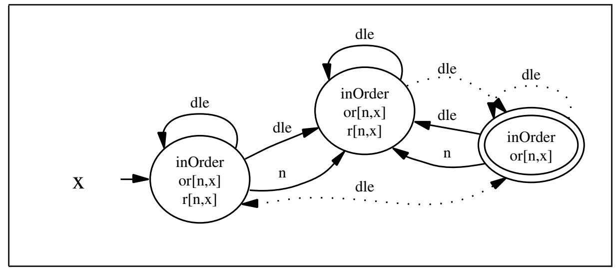

Figure 8: An abstract state on which the analysis reported the error message of line l_{31}.

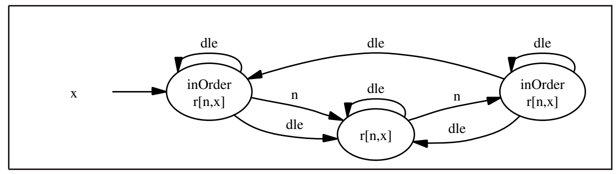

Figure 9: A concrete counterexample demonstrating the error in the bubble-sort program as found by `Generating-Counterexamples` algorithm.

We now invoke the algorithm of Figure 5 using a maximal path length of $p = 50$. We also limit the first-order theorem prover (See Section 6) to a maximal model size of 3.

When reaching line 3 in the algorithm, l has the value l_{31}, $\sigma^\sharp$ corresponds to the structure of Figure 8 and $\varphi_{err} = \neg(\forall v, r[n, x](v) \iff or[n, x](v))$. The algorithm now invokes `Input-Instance` with the conjunction of $\hat{\gamma}(\sigma^\sharp)$ and φ_{err} and the label l_{31}. For brevity, the symbolic representation of the abstract state of Figure 8 is not shown.

`Input-Instance` starts exploring the possible paths from $entry$ to l_{31}. The paths are sorted from the shortest to the longest. For each path, φ_0 is calculated from $\hat{\gamma}(\sigma^\sharp) \wedge \varphi_{err}$ by repeated computation of the weakest precondition.

The theorem prover fails to find a model for all the first 142 paths checked. On the 143^{th} path, it finds a concrete example as shown in Figure 9. The path is $l_1 \to l_2 \to l_3 \to l_4 \to l_6 \to l_7 \to l_8 \to l_9 \to l_{10} \to l_{11} \to l_{12} \to l_{25} \to l_{26} \to l_{27} \to l_{11} \to l_{12} \to l_{13} \to l_{14} \to l_{15} \to l_{16} \to l_{17} \to l_{20} \to l_{22} \to l_{23} \to l_{11} \to l_{12} \to l_{25} \to l_{26} \to l_{27} \to l_{11} \to l_6 \to l_{31}$.

This path corresponds to an execution in which the outer loop (starting at l_6) is visited twice and the inner loop (starting at l_{11}) is visited 4 times (twice for each outer iteration).

The generated counterexample, shown in Figure 9, is a list referenced by `x` containing 3 elements such that the last element has the smallest `data` value, and the second element has the largest `data` value. Note that as a result, the $inOrder(u)$ predicate does not hold for the second element. The generated counterexample corresponds to the violating input we initially presented in Figure 2.

6. PROTOTYPE IMPLEMENTATION

We have implemented a prototype of our tool and used it to generate counterexamples for a set of small but interesting example programs.

Our implementation currently interfaces with the TVLA [22] static analysis engine. The model enumerator we are using is Mace 4 [23, 24] which is a model enumerator for first-order logic.

Table 7 shows our benchmark programs.

The first 2 programs involve sorting of singly linked lists. The

Program	Description
bubble-sort Bug	The running example
insert-sort bug 1,2	Insert Sort
rotate	moves the tail of a list to it's end
search	Searches for an item in a list
merge1, 2	Merges two ordered lists
swap	Swaps the first two elements of a list
simple1	Traverses three steps in a list
simple2	Traverses three steps in a list, reset and re-traverse them in a loop. Causes a false alarm.

Table 7: Description of benchmark programs.

next 4 programs perform manipulation and traversing of lists. All of these programs were previously used as benchmarks for various analyses implemented using TVLA [21, 13]. In some programs, we manually introduced bugs to force error reports by the analysis.

The last two programs, `simple1` and `simple2`, were added to the benchmark set to test a case were there is only one possible counterexample and a case where the error is a false alarm.

In every program, one or more of the errors were reported:

PERM The resulting list is not a permutation of the original list.

NULL A possible null dereferencing, i.e. performing x=x.n where x is NULL.

LEAK There is a possibility that a node is not reachable from any of the program variables.

INIT There is a possibility that a variable value is referenced without being initialized first.

ORDER A possible violation of the list order, meaning the list is not sorted.

Our experiments were conducted on a dual 1Ghz Pentium-III with 2GB of memory running Linux. Table 8 shows the results for running the counterexample algorithm on the benchmark programs. Note that `merge` program had 12 bugs at different labels. For brevity, the table specifies the results of only 4 of them.

The column "CFG Nodes" shows how many nodes are in the control flow graph of the program, "total strct" is the number of abstract structures found by the analysis, "max length" is the maximum length of the paths, "model size" is the maximum model size used with Mace4, "Error" is the error reported.

The next four columns refer to the counterexample.
"fnd" shows whether a counterexample was found, "real err" indicates whether the error is a true or false alarm. "model size" shows the size (in nodes) of the counterexample found. "path length" shows the length of the path on which the counterexample was found.

The next two columns, "ops" and "cls" show the number of operators and number of clauses in the calculated φ_0.

The last two columns are "paths checked", showing how many paths the algorithm checked before finding a counterexample (or stopping because maximum path length was reached), and "Time" — the elapsed time of the running of the algorithm in seconds.

In all the benchmarks except for the last, a counterexample was found. In the last benchmark, `simple2` a counterexample was not found, and this error is indeed a false alarm.

For the Sorting programs, manual investigation of error messages with so many abstract states created is very difficult. For the first two bugs, one would have to traverse paths of more than 40 nodes in length. It is interesting to note that even for small programs such as `merge`, a large number of abstract states is created, making it hard to reconstruct even short execution paths.

7. EXTENSIONS

In this section, we describe possible extensions of our algorithms. In Section 7.1 we show how to use the algorithm `Input-Instance` to automatically generate a set of test cases that cover the program according to a certain criteria. In Section 7.2 we discuss some variants of `Input-Instance` we have investigated.

7.1 Generating Coverage Test Cases

Code Coverage Analysis [8] tries to find areas of the program not realized by a set of test cases. In this subsection we show how to use `Input-Instance` to automatically generate test cases that realize all the results of the analysis.

There is a large number of coverage criteria suggested in the literature [16], among them:

All-nodes Requires that each node in the control flow graph be executed by some test case.

All-edges Requires each edge in the control flow graph be traversed at least once by some test case.

All-paths Requires that every complete path (from $entry$ to $exit$) be traversed.

It is straightforward to use `Input-Instance` algorithm for producing an adequate All-nodes test case set. To do so, one applies `Input-Instance` at every label on every abstract state created by the analysis until an input instance is found for every label [2].

`Input-Instance` allows generating test cases for more general domains. For instance, in our experiments, we were able to produce an adequate All-nodes test set for the bubble-sort running example.

To produce an All-paths test case set, `Input-Instance` should be applied only on the abstract states at label $exit$. Path exploration depth is directly controlled via the p parameter.

Producing All-edges test case set is similar to All-paths, however a small change is required to avoid checking paths not covering new edges.

Leveraging the static analysis results allows considering a new coverage criteria - "All-Abstract-States" defined as follows.

[2]In shape analysis, we found it a good heuristic to try the abstract states in ascending order of "complexity" (the number of list nodes in each state)

						Counterexample				φ_0			
Program	CFG Nodes	total strct	max length	model size	Error	real err	fnd	model size	path length	ops	cls	paths checked	time (sec)
bubble-sort Bug 1	32	1024	50	3	PERM	T	T	3	44	1913	196	142	42
insert-sort Bug 1	31	1132	100	3	PERM	T	T	2	42	1556	153	42	13
insert-sort Bug 2	31	251	100	3	ORDER	T	T	3	19	851	84	3	1
rotate	15	85	50	4	LEAK	T	T	4	21	1751	140	10	212
search	7	42	30	3	NULL	T	T	2	5	829	59	1	1
merge1	34	374	100	5	INIT	T	T	3	11	1945	96	4	85
merge1	34	374	100	5	LEAK	T	T	3	6	3793	134	1	11
merge1	34	374	100	5	INIT	T	T	3	7	3809	138	2	16
merge1	34	374	100	5	LEAK	T	T	3	15	4255	173	8	244
merge2	35	396	200	3	LEAK	T	T	3	17	1918	123	9	90
swap	12	36	30	3	NULL	T	T	2	5	935	66	1	1
Simple1	8	21	10	3	NULL	T	T	3	7	310	30	1	0
Simple2	12	60	30	3	NULL	F	F	N/A	N/A	594	63	524	133

Table 8: Results for finding counterexamples for the benchmark programs. See Section 6.

DEFINITION 7.1 (ALL-ABSTRACT-STATES). *Denote $\Sigma^\sharp$ as the set of labelled abstract states representing all the possible concrete states at all the labels. A test set Σ is* All-Abstract-States *adequate when for each labelled abstract state $\langle \sigma^\sharp, l \rangle \in \Sigma^\sharp$ there exists an input instance in Σ.*

The algorithm for generating an All-Abstract-States adequate test set uses `Input-Instance` as a procedure. `Input-Instance` is applied for every abstract state at every label of the program.

An adequate All-Abstract-States would have a concrete test case for every possible abstract state the program may reach. We believe this criteria might have an advantage over other criteria, especially in domains involving heap manipulation. Further experiments are needed to compare this criteria with the others.

7.2 Variants Of Input-Instance

7.2.1 Extended Interface with the Static Analysis

The algorithm `Input-Instance` of Section 3 uses a limited interface with the static analysis tool. In fact, it only uses information of the reported error — the error label, the error condition, and the abstract state for which the error was reported.

Given a richer interface to the results of the static analysis tool, the algorithm `Input-Instance` could benefit from this interface by restricting the computed weakest precondition only to states that are described by the abstract state descriptors computed by the abstract interpretation.

Technically, at every step of the weakest precondition computation, the result of the weakest precondition is conjoined with the symbolic concretization of the corresponding abstract states.

The resulting φ_0 would therefore be more restricted. This may aid the theorem prover to find a model faster. In our experiments we did not notice any significant performance difference.

7.2.2 Path Pruning

Another interesting variant is *path pruning*. Instead of calculating weakest precondition for the whole path and then run the theorem prover, only to find out the model is inconsistent, one can run the theorem prover after each step of the weakest precondition calculation and stop traversing that path if the formula is unsatisfiable.

Our experiments showed that most paths are pruned relatively late during their exploration, thus the time saved from not exploring path fragments in comparison to the cost of extra calls to the theorem prover made this approach ineffective.

8. CONCLUSION

In this paper we present a tool that allows enjoying the benefits of sound abstract interpretation (no error is missed) while reducing the number of false alarms that need to be manually investigated. The tool can also be used to generate an adequate test case according several coverage criteria.

The tool is generic and is applicable to any abstract domain having a symbolic concretization function, an ability to calculate weakest precondition and a finite-model counterexample generator.

To find input instances, we use a new bounded model checking algorithm that separates path-exploration from path-verification.

We have implemented a prototype of the tool for shape analysis and were able to produce counterexamples for several interesting benchmark programs.

In the future we plan to implement our tool for other domains. It may also be interesting to investigate further optimizations to `Input-Instance` and also compare the effectiveness of the All-Abstract-States criteria relative to other criteria.

9. REFERENCES

[1] AbsInt. http://www.absint.com.

[2] PolySpace technologies. http://www.polyspace.com.

[3] T. Ball, R. Majumdar, T. Millstein, and S. Rajamani. Automatic predicate abstraction of C programs. In *Proc. of the ACM SIGPLAN '01 Conf. on Programming Language Design and Implementation (PLDI-01)*, volume 36.5 of *ACM SIGPLAN Notices*, pages 203–213, N.Y., June 20–22 2001. ACMPress.

[4] A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu. *Bounded Model Checking*, volume 58 of *Advances in Computers*. Academic Press, 2003.

[5] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Mine, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *PLDI*, pages 196–207. ACM, 2003.

[6] E.M. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

[7] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV'00*, pages 154–169, July 2000.

[8] S. Cornett. "code coverage analysis", 2002. "http://www.bullseye.com/webCoverage.html".

[9] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Symp. on Princ. of Prog. Lang.*, pages 269–282, New York, NY, 1979. ACM Press.

[10] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 84–97, Tucson, Arizona, January 1978. ACM Press, New York, NY.

[11] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, Hewlett Packard Laboratories, July 23 2003.

[12] E.W. Dijkstra. *A discipline of programming*. Prentice Hall, 1976.

[13] N. Dor, M. Rodeh, and M. Sagiv. Checking cleanness in linked lists. In *Static Analysis Symposium*, pages 115–134, 2000.

[14] N. Dor, M. Rodeh, and S. Sagiv. CSSV: towards a realistic tool for statically detecting all buffer overflows in c. In *Prog. Lang. Design and Impl.*, pages 155–167, 2003.

[15] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. *LNCS*, 1254:72–83, 1997.

[16] N. Gupta and R. Gupta. Data flow testing. In Y. N. Srikant and Priti Shankar, editors, *The Compiler Design Handbook: Optimizations and Machine Code Generation*, chapter 7, pages 247–267. CRC Press, 2002.

[17] J. G. Henriksen, J. L. Jensen, M. E. Jørgensen, N. Klarlund, R. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In Ed Brinksma, Rance Cleaveland, Kim Guldstrand Larsen, Tiziana Margaria, and Bernhard Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1019, pages 89–110. Springer, 1995.

[18] D. Jackson and M. Vaziri. Finding bugs with a constraint solver. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 14–25. ACM, 2000.

[19] N.D. Jones and S.S. Muchnick. Flow analysis and optimization of Lisp-like structures. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 4, pages 102–131. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[20] G.A. Kildall. A unified approach to global program optimization. In *Symp. on Princ. of Prog. Lang.*, pages 194–206, New York, NY, 1973. ACM Press.

[21] T. Lev-Ami, T. Reps, M. Sagiv, and R. Wilhelm. Putting static analysis to work for verification: A case study. *ACM SIGSOFT Software Engineering Notes*, 25(5):26–38, 2000.

[22] T. Lev-Ami and M. Sagiv. TVLA: A framework for kleene based static analysis. In *Static Analysis Symposium*, pages 280–301. Springer, 2000.

[23] W. McCune. A Davis-Putnam program and its application to finite first-order model search: quasigroup existence problems. Technical Report ANL/MCS-TM-194, Argonne National Laboratory, 1994.

[24] W. McCune. Mace4 reference manual and guide. Technical Report ANL/MCS-TM-263, Argonne National Laboratory, 2003.

[25] Microsoft Research. The SLAM project, 2001. http://research.microsoft.com/slam/.

[26] C.S. Pasareanu, M.B. Dwyer, and W. Visser. Finding feasible counter-examples when model checking abstracted Java programs. In *Proc. of the 7th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2001*, volume 2031 of *LNCS*, pages 284–298, 2001.

[27] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(3):217–298, 2002.

[28] M. Vaziri and D. Jackson. Checking properties of heap-manipulating procedures with a constraint solver. In *Proc. of the 9th Int. Conf. of Tools and Algorithms for the Construction and Analysis of Systems , TACAS 2003*, volume 2619 of *LNCS*, pages 505–520. Springer, January 2003.

[29] E. J. Weyuker. Axiomatizing Software Test Data Adequacy. *IEEE Transactions on Software Engineering*, SE-12(12):1128–1138, December 1986.

[30] G. Yorsh. Logical characterizations of heap abstractions. Master's thesis, Tel Aviv University, 2003.