Inferring Specifications to Detect Errors in Code

Mana Taghdiri and Daniel Jackson
({taghdiri, dnj}@mit.edu)
Computer Science and AI Lab,
Massachusetts Institute of Technology, Cambridge, MA 02139

Abstract. A new technique is presented to statically check a given procedure against a user-provided property. The method requires no annotations; it automatically infers a context-dependent specification for each procedure call, so that only as much information about a procedure is used as is needed to analyze its caller. Specifications are inferred iteratively. Empty specifications are initially used to over-approximate the effects of all procedure calls; these are later refined in response to spurious counterexamples. When the analysis terminates, any remaining counterexample is guaranteed to be valid. However, since the heap is finitized, the absence of a counterexample does not guarantee the validity of the given property.

1. Introduction

Traditional program verification makes extensive use of modularization. Each procedure is checked against its specification, using the specifications of its called procedures as surrogates for their code.

Automating such approaches has motivated several tools. ESC/-Java [15], for example, extracts verification conditions from a procedure and presents them for proof (or refutation) to a specially tailored theorem prover. However, it requires users to provide specifications for called procedures. This places a significant burden on users and therefore limits the applicability of the tool. Jalloy [33], a SAT-based counterexample detector for Java programs, suffers from the same problem. Although it can inline called procedures to eliminate the need for user-provided specifications, such inlining does not scale to large programs.

On the other hand, software model checkers, such as SLAM [3] and BLAST [18], can find bugs in programs automatically without requiring users to provide any annotations. Their scalability typically comes from the demand-based refinements of predicate abstractions [16] of the code, based on a paradigm known as "counterexample guided abstraction refinement" (CEGAR) [7], which consists of the following four basic steps:

– An abstraction of the program is computed which is an over-approximation of the original code.

- The abstraction is analyzed. If no counterexample is found, the property holds in the code and the analysis terminates.
- If a counterexample is found, it is checked for validity. If valid, a fault has been discovered and the analysis terminates.
- If the counterexample is spurious, the abstraction is refined so that this counterexample is eliminated, and the process starts over.

This paradigm is the basis of a number of software analysis techniques. However, to our knowledge, they all involve refinement of predicate abstractions, and do not exploit the abstraction boundaries defined by procedures.

In this paper, we describe a modular analysis in which a procedure is checked against a given property by substituting specifications for the called procedures. The specifications are inferred automatically from the code rather than being provided by the user. These specifications are exploited in checking like the specifications of ESC/Java, but are refined by a mechanism more similar to that of SLAM and BLAST.

Our analysis is an application of the CEGAR paradigm. However, it differs from all previous applications of this paradigm in that the abstraction and its subsequent refinements follow the abstraction boundaries of the code defined by procedures. Instead of refining a predicate abstraction of the code, our approach refines the specifications that represent the behavior of called procedures.

We do not aim at inferring full specifications that capture the behavior of called procedures completely. Instead, we start with a rough initial specification for each procedure and refine it on demand. The inferred specifications need capture only as much information about procedures as is required to establish the correctness of the main procedure being checked, which depends on not only the calling context, but also the property being checked. As a result, a very partial specification is sometimes sufficient, because even though it barely captures the behavior of the called procedure, it nevertheless captures enough to verify the caller with respect to the property of interest.

An earlier publication [32] presented these ideas. This paper improves on that work in two respects. It presents a new abstract framework that makes it easier to exploit the idea in a wider range of settings. The framework assumes an underlying analysis in which counterexamples are found by solving constraints extracted from the code and the given property. It (1) assumes very little about the programming language except that it supports procedure declaration, (2) does not assume any particular translation of code to constraints, and thus provides the opportunity for exploiting different translation techniques

(e.g. [13, 35]), and (3) provides rigorous conditions for termination, soundness, and completeness.

Furthermore, this paper presents a better technique for handling loops and recursion. The technique does not require users to provide bounds on the number of loop iterations and recursion executions. Instead, it assumes that loops are desugared into recursive procedures, and infers the specification of each recursive call as needed, i.e. unrolls them on demand. Therefore, compared to our previous approach, this approach potentially finds more errors in programs.

Our instantiation of the framework uses the Alloy modeling language [21] as the logic, and a SAT solver as the constraint solver, and handles programs written in a subset of Java. It analyzes procedures with respect to finitized heaps. Consequently, although counterexamples are guaranteed not to be spurious, their absence does not constitute proof of correctness. The framework, however, does not depend on these compromises, and seems to hold promise for application in other contexts.

This paper is organized as follows: Section 2 gives an overview of the method. Section 3 illustrates the method using a small Java program. Section 4 gives the technical details of the abstract framework. Section 5 presents one instantiation of the framework as implemented in our tool. Section 6 gives the experimental results. Section 7 discusses different aspects of our technique. Section 8 compares our method with the related work, and Section 9 concludes the paper.

2. Overview

Our analysis checks a procedure with respect to a given property. Figure 1 shows the analysis framework. It consists of the following phases:

Abstraction: The body of the procedure selected for analysis is translated to a set of logical constraints. These constraints capture the semantics of the procedure except at its call sites. All procedure calls are replaced with approximate specifications so that the abstraction is an over-approximation of the original code.

Solving: The generated constraints and the negation of the property being checked are handed to a constraint solver. If no solution is found, the property holds in the original procedure. On the other hand, if a solution satisfying all the constraints is found, it indicates a potential violation of the property, and must be checked for validity.

Validity check: The validity of a solution is determined by checking the consistency of each procedure call with the found solution, again using a constraint solver. If all procedure calls are consistent, the solu-

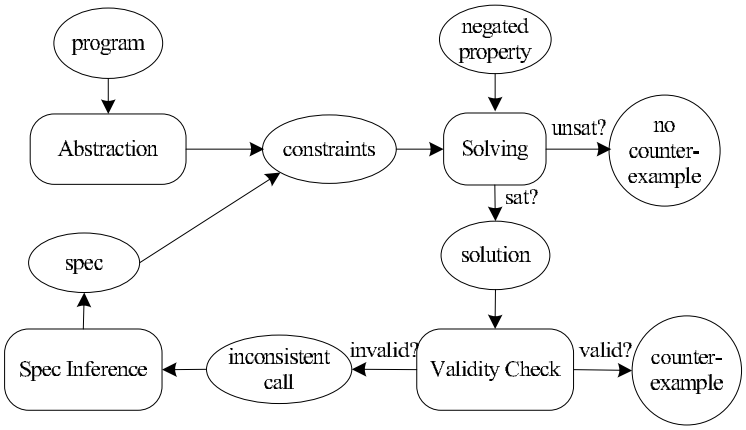

Figure 1. An overview of the framework

tion represents a feasible counterexample, and the analysis terminates. If a procedure call is inconsistent, however, its specification must be refined.

Specification inference: A new partial specification is inferred for an inconsistent procedure call from a proof of invalidity generated by the constraint solver. This specification rules out not only the given invalid solution, but also all solutions isomorphic to that. The new specification is then conjoined with the old specification of the procedure to form a more precise specification. The process then starts over at the solving phase.

Termination, soundness (that is, found counterexamples are feasible), and completeness (that is, all counterexamples are found) of this analysis depend on the particular translation technique that encodes program statements as logical constraints, and the constraint solver used in the instantiation of the framework. However, as proved in Section 4.4, the specification inference approach does not introduce further unsoundness. Moreover, it does not introduce non-termination or further incompleteness unless the analyzed program contains recursive calls that can loop forever.

3. Example

In this section, we illustrate our technique by first describing the user's experience checking a Java program, then giving the specifications inferred to check that program, and finally describing the underlying analysis.

```
class Entry {
  int val;
  Entry next;

  Entry(int v) {
    val = v;
    next = null;
  }
}
class Set {
  Entry elems;

  /* assert:
     ((this.elems = null) || (p.elems = null)) => ($return.elems' = null)
  */
  Set intersect(Set p) {
    Set res = new Set();
    Entry curr = this.elems;
    while (curr != null) {
      boolean found = p.contains(curr.val);
      if (found)
        res.add(curr.val);
      curr = curr.next;
    }
    return res;
  }

  Set() {
    elems = null;
  }

  void add(int v) {
    if (!contains(v)) {
      Entry tmp = new Entry(v);
      tmp.next = this.elems;
      this.elems = tmp;
    }
  }

  boolean contains(int v) {
    Entry l = this.elems;
    while (l != null) {
      if (l.val == v)
        return true;
      l = l.next;
    }
    return false;
  }
}
```

Figure 2. Example

Valid properties: Figure 2 shows a set datatype whose elements are represented by a singly linked data structure. The `intersect` method returns the intersection of the receiver set, `this`, and another set `p`, passed as an argument. The given property asserts that if either one of the intersected sets is empty (i.e. its `elems` field is `null`), the returned set is also empty.

Since our tool checks programs with respect to finite heaps, the user specifies a bound on the number of objects of each type. In this example, we assume that the user-provided bound on the number of entries and sets is 3.

Checking the `intersect` method against the given property does not generate any counterexamples, implying that the property holds in the finite heap.

Invalid properties: If the code were buggy, a counterexample would have been generated. In order to illustrate that, assume that the condition for adding an element to the resulting set is mistakenly written as `if` (!found) rather than `if` (found). The property does not hold any more, and the counterexample given in Figure 3 is generated.

Figure 3(a) shows the values of the program variables in the pre- and post-states. The unprimed and primed names represent the values of variables and fields before and after the execution of the `intersect` method respectively. The values `S0`, `S1`, and `S2` are symbolic set objects. The values `E0` and `E1` are symbolic `Entry` objects, and the value `int0` is a symbolic integer. This counterexample represents the case in which the receiver set contains one integer `int0` and `p` is an empty set. Because of the bug, the resulting set contains `int0` rather than being empty.

Our tool currently outputs counterexamples in the format of Figure 3(a). However, it can easily be improved to generate the trace corresponding to each counterexample by highlighting the statements executed in that counterexample and annotating them by the values of program variables before and after their execution. The trace corresponding to the counterexample of Figure 3(a) is shown in Figure 3(b). In this counterexample, all statements in the `intersect` method are executed; the loop body is executed only once. Instead of rewriting the values of all program variables, the trace describes only the updates after each statement.

In the rest of this section, we describe analyzing the code given in Figure 2 where the property holds. Analyzing the buggy case follows the same general steps, but infers different specifications.

pre-state:
(this = S0) (p = S1)
(S0.elems = E0) (S1.elems = null)
(E0.val = int0) (E0.next = null)

post-state:
($return = S2)
(S2.elems' = E1)
(E1.val' = int0) (E1.next' = null)

(a)

```
Set intersect(Set p) : {
  [ (this = S0), (p = S1), (S0.elems = E0),
                          (S1.elems = null), (E0.val = int0), (E0.next = null)]
  Set res = new Set();
                                                  [.., (res = S2), (S2.elems = null)]
  Entry curr = this.elems;
                                                                  [.., (curr = E0)]
  while (curr != null) {
    boolean found = p.contains(curr.val);
                                                                [.., (found = false)]
    if (!found)              //bug seeded
      res.add(curr.val);
                                  [.., (S2.elems = E1), (E1.val = int0), (E1.next = null)]
    curr = curr.next;
                                                                 [.., (curr = null)]
  }
  return res;
                                                               [.., ($return = S2)]
}
```

(b)

Figure 3. Sample counterexample: (a) symbolic values for pre- and post-states, (b) corresponding trace

3.2. Inferred Specifications

In order to validate the property in the example of Figure 2, our tool infers the specifications given in Figure 4 for the methods reachable from the `intersect` method.

A specification is represented by a set of logical constraints, implicitly conjoined. The variable `$return` represents the return value. A

Set.Set(): { (this.elems' = null) }

Entry.Entry(v): { (this.val' = $?_{int}$)
(this.next' = $?_{Entry}$) }

Set.add(v): { ($?_{Entry}$.val' = $?_{int}$)
($?_{Entry}$.next' = $?_{Entry}$)
($?_{Set}$.elems' = $?_{Entry}$) }

Set.contains(v): { ((this.elems = null) ⇒ ($return = false))
((this.elems != null) ⇒ ($return = $?_{Bool}$)) }

Figure 4. Specifications inferred to validate the example property

question mark $?_T$ is used to represent arbitrary values of type T chosen non-deterministically. For example, $return = $?_{Bool}$ in the Set.contains specification indicates that this method can return any boolean value arbitrarily, and $?_{Entry}$.val' = $?_{int}$ in the Set.add specification denotes an arbitrary change in the value of the val field in any object of type Entry. Any variable or field not mentioned in the specification is assumed to be unchanged.

These specifications are inferred as needed in order to check the given property, so they vary in precision. The specification inferred for the Set constructor, for example, represents its behavior completely. (Our tool desugars an allocation statement x = new T(e) to two consecutive statements x = new T; x.T(e) where the first one allocates a fresh memory location and the second one calls the constructor. Thus, the constructors' specifications do not specify memory allocations; they only initialize the allocated objects.) On the other hand, the specifications of the Entry constructor and the add method are very partial. They only indicate what fields may be updated by those methods. The specification of the contains method also provides only partial information about the method's behavior: if the receiver set is empty (this.elems = null), the contains method returns false, otherwise, the return value is unknown.

Although these specifications barely capture the behavior of their corresponding methods, they are sufficient for validating the property.

3.3. Underlying Analysis

The specifications described in the previous section are inferred in two iterations. The underlying analysis proceeds as follows. Initial specifications (Figure 5) are computed to over-approximate the behaviors of the methods reachable from the intersect method. These specifications

Set.Set(): { (this.elems' = $?_{Entry}$) }

Entry.Entry(v): { (this.val' = $?_{int}$)
(this.next' = $?_{Entry}$) }

Set.add(v): { ($?_{Entry}$.val' = $?_{int}$)
($?_{Entry}$.next' = $?_{Entry}$)
($?_{Set}$.elems' = $?_{Entry}$) }

Set.contains(v): { ($return = $?_{Bool}$) }

Figure 5. Initial specifications

```
    Set intersect(Set p) {
1:    Set res = new Set;
2:    res.Set();
3:    Entry curr = this.elems;
4:    if (curr != null) {
5:      boolean found = p.contains(curr.val);
6:      if (found)
7:        res.add(curr.val);
8:      curr = curr.next;
9:    } <assume curr == null>
10:   return res;
    }
```

Figure 6. Unrolled method

are rough approximations that only specify what fields are updated by each method.

Although in general, our tool expects loops to be written as tail recursion, in this example, we unroll each loop once for simplicity. Figure 6 shows the intersect method after loop unrolling and desugaring the allocation statement.

Figure 7(a) shows the initial abstraction of the intersect method. The body of the method is translated to a set of constraints. Assignment statements become equality constraints and branches become logical implications. The function nextFresh(T) returns a logical expression representing a fresh element of type T. (To keep the example simple, we do not give the formal definition of this function.) The specifications in Figure 5 are used to abstract the call sites. (Line numbers used in this figure correspond to the line numbers in Figure 6.)

First iteration: These constraints along with the negation of the given property (Figure 7(b)) are given to a constraint solver. The solution shown in Figure 8 is found as a counterexample. In this counterexample, the set constructor is called (in line 2). It assigns E0, rather than null, to the elems field of the allocated object S2. Although ac-

```
   Set.intersect(p): {
1:   (res' = nextFresh(Set))
2:   {(res'.elems' = ?Entry)}                                  res.Set()
3:   (curr = this.elems)
4:   (curr != null) ⇒ {
5:       {(found' = ?Bool)}                    found = p.contains(curr.val)
6:       (found') ⇒
7:          {(?Entry.val' = ?int)                             res.add(curr.val)
            (?Entry.next' = ?Entry)
            (?Set.elems' = ?Entry)}
8:       (curr' = curr.next')
9:       (curr' = null)
       }
10:  ($return = res')
   }
```

(a)

```
{ (this.elems = null)
   (p.elems = null)
   ($return.elems' != null) }
```

(b)

Figure 7. (a) Initial abstraction: method body is translated to constraints; specifications are substituted for call sites. (b) Negation of the property being checked.

ceptable in the abstraction, this behavior is invalid in the original code.

The invalidity of this counterexample is determined by checking the original code of the set constructor against the counterexample. The constraint solver generates the following proof of invalidity:

```
(res' = S2)
(S2.elems' = E0)
(this = res')
(this.elems' = null)
-------------------
       false
```

The proof gives an inconsistent subset of the constraints. The first two lines correspond to the values assigned by the counterexample, and the next two lines give the translation of the constructor call in line 2. A new specification for this call is extracted from the proof by removing those constraints that are included in the counterexample, namely (res' = S2) and (S2.elems' = E0). The following specification is inferred:

(this = S0) (p = S1)
(S0.elems = null) (S1.elems = null)
(res' = S2) (S2.elems' = E0)
($return = S2)

(a)

Set intersect(Set p) : {

$[\ (this = S0),\ (p = S1),\ (S0.elems = null),\ (S1.elems = null)]$

1 : **res** = **new Set**;

$[.., (res = S2)]$

2 : **res**.**Set**();

$[.., (S2.elems = E0)]$

3 : **Entry curr** = **this**.**elems**;

$[.., (curr = null)]$

4: if (curr != null) {
5: boolean found = p.contains(curr.val);
6: if (found)
7: res.add(curr.val);
8: curr = curr.next;
9: } **<assume curr == null>**

10 : **return res**;

$[.., ($return = S2)]$

}

(b)

Figure 8. First counterexample: (a) constraint solver's solution, (b) corresponding trace

```
{ (this = res')
  (this.elems' = null)}
```

In this case, the inferred specification represents the full behavior of the constructor. It is conjoined with the constructor's old specification in line 2.

Second iteration: The intersect method is checked against the given property again. A new counterexample, shown in Figure 9, is found. In this counterexample, the set constructor (in line 2), the contains method (in line 5) and the add method (in line 7) are called. The value of curr.val is the symbolic integer int0, and p.elems is null. Despite this, the call to contains returns true, and the call to add assigns int1, rather than int0 to res'.elems'.val'. Although acceptable in the abstraction, again these behaviors are not feasible in the original program.

(this = S0) (p = S1)
(S0.elems = E0) (S1.elems = null)
(E0.val = int0) (E0.next = null)
(res' = S2) (S2.elems = null)
(found' = true)
(S2.elems' = E1)
(E1.val' = int1) (E1.next' = null)
($return = S2)

(a)

Set intersect(Set p) : {

$[\ (this = S0),\ (p = S1),\ (S0.elems = E0),$
$(S1.elems = null),\ (E0.val = int0),\ (E0.next = null)]$

1 : **res = new Set**;

$[.., (res = S2)]$

2 : **res.Set**();

$[.., (S2.elems = null)]$

3 : **Entry curr = this.elems**;

$[.., (curr = E0)]$

4 : **if** (**curr** ! = **null**) {

5 : **boolean found = p.contains(curr.val)**;

$[.., (found = true)]$

6 : **if** (**found**)

7 : **res.add(curr.val)**;

$[.., (S2.elems = E1), (E1.val = int1), (E1.next = null)]$

8 : **curr = curr.next**;

$[.., (curr = null)]$

9 : } **<assume curr == null>**

10 : **return res**;

$[.., ($return = S2)]$

}

(b)

Figure 9. Second counterexample: (a) constraint solver's solution, (b) corresponding trace

To determine this, the contains method is analyzed first, by checking its original code against the counterexample. It should be noted that because the full specification of the set constructor was already inferred, the call to the constructor is not checked again. Figure 10(a) shows the contains method after one unrolling. Figure 10(b) gives its translation into logical constraints. The first line of the translation handles the mapping of the formal parameters to their actual values, and the last

```
  boolean contains(int v) {
1:    Entry l = this.elems;
2:    if (l != null) {
3:       if (l.val == v)
4:          return true;
5:       l = l.next;
6:    } <assume l == null>
7:    return false;
  }
```

(a)

```
  Set.contains(v): {
     (this = p) (v = curr.val)
1:   (l = this.elems)
2:   (l != null) ⇒ {
3:        (l.val = v) ⇒
4:             ($return = true)
          (l.val != v) ⇒ {
5:             (l' = l.next)
6:             (l' = null)
7:             ($return = false)}
     }
     (l = null) ⇒
7:       ($return = false)
     (found' = $return)
  }
```

(b)

Figure 10. The contains method: (a) unrolled code, (b) call site translation

line maps the returned value to the appropriate variable at the analyzed call site. (Our tool actually renames local variables so that they are distinct across call sites.) In order to capture the fact that the return statement in line 4 terminates the method, the statement following the loop (line 7) is duplicated and some branch conditions are added.

The constraints encoding the contains method along with the counterexample given in Figure 9(a) are given to the constraint solver which generates the following proof of invalidity. No further validity checking

is performed at this stage.

```
(p = S1)
(S1.elems = null)
(found' = true)
(this = p)
(l = this.elems)
((l = null) => ($return = false))
(found' = $return)
-------------------------------
      false
```

The first 3 lines represent the values assigned by the counterexample, and the other lines give a subset of the contains encoding. As before, a new specification for the analyzed call site is extracted from the proof by removing those constraints that are included in the counterexample, namely the constraints given in the first 3 lines. Furthermore, intermediate local variables are removed by inlining their values. The following specification is therefore inferred.

```
{ (this = p)
  (found' = $return)
  ((this.elems = null) => ($return = false)) }
```

This specification says that if the receiver set is empty, the method returns false, otherwise, the return value is unconstrained, expressing only those parts of the contains method that are relevant to the found counterexample.

This new specification is conjoined at the call to the contains method in line 5 and analyzing the intersect method starts over. In this example, no further counterexamples are found. Thus, the process terminates and the property has been validated for the finite heap.

4. Methodology

In this section, we explain the essence of our method formally and discuss its termination, soundness, and completeness properties.

We describe an abstract framework for checking a procedure against a user-provided assertion. The assertion can be expressed in any language as long as it can be translated to a set of constraints. The core idea is to infer the specifications of called procedures on demand. In order to do that, the framework relies on a translation of code to a set of constraints whose satisfiability can be determined using a constraint solver. Examples of such a translation are Jalloy [33] that translates the code to a relational first order logic, namely Alloy [21], and Saturn [35]

```
prog ::= proc*
proc ::= name([type var]*):type {stmt}
stmt ::= if pred(expr*) stmt [else stmt]
         name(expr*)
         return expr
         var = expr
         stmt; stmt
```

Figure 11. Abstract syntax for the analyzed language

that translates the code to a set of boolean constraints in conjunctive normal form.

This framework reformulates our approach in a more general setting than our previous work [32]. It is parameterized by basic functions, assumed to satisfy some basic axioms, in order to make the main idea applicable to a wider range of applications.

Our instantiation of the framework explained in Section 5 will provide concrete examples for the basic functions introduced in this section.

4.1. Definitions

Program syntax. We target imperative programs supporting procedure declarations. Figure 11 gives an abstract syntax for a fragment of a basic programming language. A program is a sequence of procedure declarations. A procedure declaration consists of a name, a list of formal parameters, and the type of the return value. The body of a procedure is a statement that can be a branch, procedure call, return statement, or variable mutation where variable is used as a general term that can also include a pointer dereference, or an access to an object's field. There are no loop constructs in this language; they can be expressed as tail recursions.

Program semantics. Let $PVar$ be the set of all variables of a program, and $PVal$ be the set of all concrete values that can be assigned to those variables. A *program state*, $\sigma \in \Sigma$, is a partial function from variables to their values:

$$\Sigma = PVar \rightarrow PVal$$

Each program statement s is viewed as a relation over Σ:

$$[\![s]\!] \subseteq \Sigma \times \Sigma$$

where $(\sigma, \sigma') \in [\![s]\!]$ when executing s in the state σ can result in the state σ'.

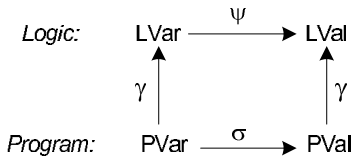

Figure 12. Relationship between different domains.

Variable mapping. Translation of a program to logical constraints is based on a mapping from program variables and their values to logical variables and logical values. Since logical variables are immutable, the same program variable can be mapped to different logical variables at different program points. In fact, the mapping can differ not only at each program point, but also at different executions of the same program point. Therefore, we define a *program moment*, $\pi \in \Pi$, as a unique identifier for each execution of a program point. That is,

$$\Pi = P \times Int$$

where P is the set of program points. A program moment $\pi = (p, i)$ represents the i^{th} execution of the program point p.

We use $\gamma \in \Gamma$ to denote the variable mapping used in the translation:

$$\Gamma = ((PVar \times \Pi) \rightarrow LVar) \quad \cup \quad (PVal \rightarrow LVal)$$

where $LVar$ and $LVal$ respectively represent the set of logical variables and their possible values.

Constraints. Our framework does not rely on any particular property of the logic used to express the constraints. However, it assumes that a constraint solver is available that determines the satisfiability of a set of constraints and generates proofs in case of unsatisfiability.

A *logical assignment* $\psi \in \Psi$ is a partial function from logical variables to values:

$$\Psi = LVar \rightarrow LVal$$

Let $\mathbb{C}$ denote the set of all constraints. A logical assignment ψ is a *solution* to a set of constraints $C \subseteq \mathbb{C}$ if all the constraints in C evaluate to *true* under the value mapping defined by ψ, denoted by $eval_\psi(C) = true$. Figure 12 shows the relationship between different domains defined in this section.

4.2. Basic Operations

Constraint solving. A constraint solver is used to determine whether or not a set of constraints has a solution. We assume that all solutions generated by the constraint solver are *total functions*. That is, they

assign some logical value to every variable that appears in the analyzed constraints even if the constraints can be satisfied regardless of the values of some variables.

Furthermore, if a partial logical assignment already exists, a constraint solver can determine if that assignment can be extended to a solution to a given set of constraints. For a set of constraints C and a partial assignment ψ_0, $solve(C, \psi_0)$ represents the set of solutions to C that preserve the mapping already defined by ψ_0:

$$\begin{aligned} &solve : \mathcal{P}(\mathbb{C}) \times \Psi \rightarrow \mathcal{P}(\Psi) \\ &solve(C, \psi_0) = \{\psi \mid \psi_0 \subseteq \psi \ \wedge \ eval_\psi(C) = true\} \end{aligned}$$

where $\mathcal{P}(A)$ denotes the powerset of A. Thus, $solve(C, \psi_0) = \emptyset$ implies that no such solution exists.

Proof generation. We assume that the constraint solver used in our analysis can generate a proof of invalidity when the given constraints are inconsistent (unsatisfiable). An *invalidity proof* for an inconsistent set of constraints C is a subset of C that is also inconsistent and thus is a witness that C does not have a solution. Although C is an invalidity proof for itself, a good proof contains a smaller number of constraints. We introduce a function *core* to represent the unsatisfiable core of a set of inconsistent constraints returned by the constraint solver as the invalidity proof.

If a set of constraints C is solved with respect to a partial solution ψ and no solution is found, the invalidity proof, $core(C, \psi)$, will denote a subset of C which is unsatisfiable with respect to ψ. That is,

$$\begin{aligned} &core : \mathcal{P}(\mathbb{C}) \times \Psi \rightarrow \mathcal{P}(\mathbb{C}) \\ &core(C, \psi) = C' \ \Rightarrow \ ((C' \subseteq C) \wedge solve(C', \psi) = \emptyset) \end{aligned}$$

Converting states to assignments. A program state σ corresponding to a program moment π can be encoded as a logical assignment using the mapping defined by γ. We use *toAssignment* to represent this encoding.

$$\begin{aligned} &toAssignment : \Sigma \times \Pi \times \Gamma \rightarrow \Psi \\ &toAssignment(\sigma, \pi, \gamma) = \{ \ (v, w) : LVar \times LVal \mid \\ &\quad (\exists x : PVar, y : PVal \mid \sigma(x) = y \ \wedge \ \gamma(x, \pi) = v \ \wedge \ \gamma(y) = w) \ \} \end{aligned}$$

That is, *toAssignment* generates a logical assignment containing all pairs that correspond to some pair in the given program state σ.

Converting assignments to executions. If a set of logical constraints encodes the behavior of a program, each solution to those constraints can be interpreted as an execution of the encoded program. Using a variable mapping γ, the *toState* function extracts a program

state for a program moment π from a logical assignment ψ:

$$
\begin{array}{l}
toState : \Psi \times \Pi \times \Gamma \rightarrow \Sigma \\
toState(\psi, \pi, \gamma) = \{\ (x, y) : PVar \times PVal \mid \\
\quad (\exists v : LVar, w : LVal \mid \psi(v) = w\ \wedge\ \gamma(x, \pi) = v\ \wedge\ \gamma(y) = w)\ \}
\end{array}
$$

That is, the program state contains all pairs of program variables to values that correspond to some pair in ψ.

Using the *toState* function, we define $getTrace$ as a function that generates a program execution for a logical assignment. A program execution is represented by a set of executed program moments and their corresponding program states. A program moment π is executed if all branch conditions leading to it, i.e. its guard conditions, evaluate to true in their corresponding program states. Let $guard(\pi)$ denote the set of guard conditions of π along with their corresponding program moments, and $symEval(e, \sigma)$ denote a symbolic evaluation of an expression e in a program state σ. That is,

$$
\begin{array}{l}
guard : \Pi \rightarrow \mathcal{P}(Expr \times \Pi) \\
symEval : Expr \times \Sigma \rightarrow PVal
\end{array}
$$

The $getTrace$ function can then be defined as follows:

$$
\begin{array}{l}
getTrace : \Psi \times \Gamma \rightarrow \mathcal{P}(\Pi \times \Sigma) \\
getTrace(\psi, \gamma) = \{\ (\pi, \sigma) \mid (toState(\psi, \pi, \gamma) = \sigma)\ \wedge \\
\qquad\qquad (\forall (e, \pi') \in guard(\pi) \mid symEval(e, toState(\psi, \pi', \gamma)) = true)\ \}
\end{array}
$$

We define *calls* to represent the set of executed program moments that correspond to call sites. That is,

$$
\begin{array}{l}
calls : \Psi \times \Gamma \rightarrow \mathcal{P}(\Pi) \\
calls(\psi, \gamma) = \{\pi \mid (\pi\ \texttt{is call site})\ \wedge\ (\exists \sigma |\ (\pi, \sigma) \in getTrace(\psi, \gamma))\ \}
\end{array}
$$

Translation. The translation of program statements to logical constraints is denoted by the *translate* function. Our framework assumes the existence of such a function, but it is independent of how the code is actually translated. Given a variable mapping γ and a statement s at a program moment π, *translate* returns a set of logical constraints C that encodes the behavior of that statement, denoted by:

$$
\begin{array}{l}
translate : Stmt \times \Pi \times \Gamma \rightarrow \mathcal{P}(\mathbb{C}) \\
translate(s, \pi, \gamma) = C
\end{array}
$$

The *translate* function is semantics-preserving iff for each statement s executing at a program moment π and ending at a program moment

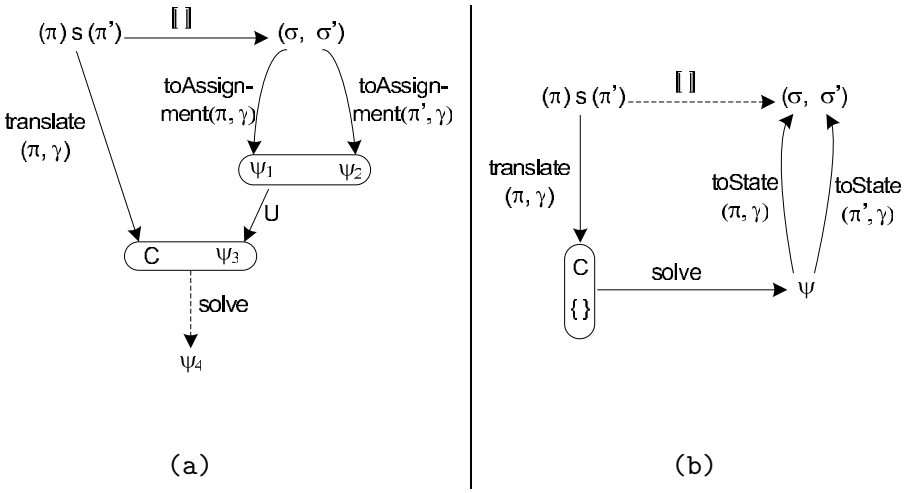

Figure 13. Translation properties: (a) completeness (b) soundness. Each diagram gives an implication read as "if the solid arrows exist, the dashed arrow exists too". Oval boxes denote tupling of function arguments.

π', the set of constraints C generated by $translate(s, \pi, \gamma)$ satisfies the following two rules:

$$\texttt{completeness:} \quad (\sigma, \sigma') \in [\![s]\!] \Rightarrow solve(C, \psi) \neq \emptyset \quad \texttt{where}$$
$$\psi = toAssignment(\sigma, \pi, \gamma) \cup toAssignment(\sigma', \pi', \gamma)$$

$$\texttt{soundness:} \quad \psi \in solve(C, \emptyset) \Rightarrow (\sigma, \sigma') \in [\![s]\!] \quad \texttt{where}$$
$$\sigma = toState(\psi, \pi, \gamma) \ \ \texttt{and} \ \ \sigma' = toState(\psi, \pi', \gamma)$$

That is, (1) any execution of s corresponds to some solution of C (completeness), and (2) any solution to C corresponds to some valid execution of s (soundness). These properties are shown graphically in Figure 13.

We assume that the *translate* function always returns a finite set of constraints. Depending on the translation method, this may require bounding variable domains, heap size, execution length, and/or recursive calls. Therefore, the translation may not be semantics-preserving. However, since our technique is geared at finding bugs, rather than proving the correctness of the code, it can use any translation that satisfies the soundness rule; the completeness rule is not required to hold.

Abstraction. We use the *spec* function to denote the initial specification used for a procedure call. An initial specification must underspecify the effects of its corresponding call site. That is, for a variable mapping γ and a program moment π that corresponds to a call site,

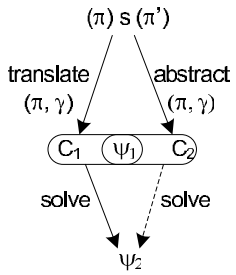

Figure 14. Over-approximation property

$spec(\pi, \gamma)$ must satisfy the following property:

$$spec : \Pi \times \Gamma \rightarrow \mathcal{P}(\mathbb{C})$$
$$solve(spec(\pi, \gamma), \psi) \supseteq solve(translate(callee(\pi), \pi, \gamma), \psi)$$

where $callee(\pi)$ denotes the procedure that is called at π, and ψ is an arbitrary logical assignment. Any specification that satisfies the above condition can be used as the initial specification in our framework. Therefore, even an empty specification that allows arbitrary change in the state of the program can serve as the initial specification. More precise specifications are inferred during the analysis if needed.

The function $abstract(s, \pi, \gamma)$ represents the initial abstraction of a program statement s at a program moment π based on a variable mapping γ. This function generates a set of constraints defined as follows:

$$abstract : Stmt \times \Pi \times \Gamma \rightarrow \mathcal{P}(\mathbb{C})$$
$$abstract(s, \pi, \gamma) = \begin{cases} spec(\pi, \gamma) & \pi \texttt{ is a call site} \\ translate(s, \pi, \gamma) & \texttt{elsewhere} \end{cases}$$

Since the specifications used for call sites are under-specifications, any solution satisfying the constraints generated by *translate* also satisfies the constraints generated by *abstract*. That is,

$$solve(abstract(s, \pi, \gamma), \psi) \supseteq solve(translate(s, \pi, \gamma), \psi)$$

In other words, the *abstract* function gives an over-approximation of the *translate* function. Figure 14 shows this property graphically.

4.3. Algorithm

The abstractAnalyze algorithm shown in Figure 15 describes the analysis in terms of the above functions. It takes a procedure p selected by

```
  datatype Result = Valid + Invalid(Trace)

  function abstractAnalyze(Proc p,
                           ConstraintSet A, VarMap γ): Result {
1:    C = abstract(p, π0, γ) ∪ ¬A                    (abstraction)
2:    checkC = true
3:    while checkC {
4:        if solve(C, ∅) = ∅                              (solving)
5:            return Valid
6:        checkC = false
7:        ψ = choose(solve(C, ∅))
8:        P = calls(ψ, γ)
9:        foreach πi ∈ P {                        (validity check)
10:           pi = callee(πi)
11:           Ci = abstract(pi, πi, γ)
12:           if solve(Ci, ψ) = ∅                      (refinement)
13:               Si = core(Ci, ψ)
14:               C = C ∪ Si
15:               checkC = true
16:               break
17:           else                                (concretization)
18:               ψ' = choose(solve(Ci, ψ))
19:               ψ = ψ ∪ ψ'
20:               P = P ∪ calls(ψ', γ)
21:       }
22:   }
23:   trace = getTrace(ψ, γ)
24:   return Invalid(trace)
  }
```

Figure 15. Analysis by procedure abstraction

the user to verify, a set of constraints A representing the assertion to check, and a variable mapping γ to use in translation. The result of the analysis is either `Valid`, indicating that no counterexample has been found, or `Invalid(t)`, indicating that `t` is a trace of p that violates the assertion.

The analysis starts by abstracting all procedures called in the analyzed procedure p (Line 1). The resulting constraints are then combined with the constraints encoding the negation of the assertion. The result is denoted by the variable C. Throughout the algorithm, this variable represents an abstraction of the code that is refined iteratively along with the negation of the assertion to check.

In the solving phase (Line 4), a constraint solver is used to find a solution to C. The empty set used in $solve(C, \emptyset)$ denotes that no partial solution exists at this point. Thus, the constraint solver can return any solution satisfying C. If no such solution exists, it means that no program execution can be found that violates the assertion and the analysis terminates (Line 5).

Otherwise, a solution ψ is arbitrarily chosen from the set of solutions to C (Line 7). This solution represents a counterexample that should be checked for validity with respect to the original program. The only abstracted statements are procedure calls. Therefore, we only need to check the consistency of the procedure calls with ψ. Variable P (Line 8) denotes the set of all call sites in the program execution corresponding to ψ. These call sites will be checked for consistency.

The validity check phase (Lines 9 - 21) checks procedure calls one by one until either a refinement is needed or all of them are shown to be consistent with the found counterexample ψ. As our abstraction is based on the procedure call hierarchy of the code, the check for validity is also done hierarchically. That is, when a procedure p_i is checked, all of its callees are abstracted. The abstraction of p_i is denoted by the variable C_i (Line 11). If p_i is not consistent with ψ, that is, if C_i does not have a solution that preserves ψ, the specification of p_i should be refined (Lines 12 - 16). In this case, the constraint solver returns a subset of C_i which is still inconsistent with ψ, as an invalidity proof represented by S_i (Line 13). This proof is a partial specification for p_i that rules out the current counterexample ψ, and possibly more. Line 14 adds S_i to the current set of constraints C to guarantee that ψ will never be generated by the constraint solver again. After this refinement, the solving phase starts over.

On the other hand, if the procedure p_i is consistent with ψ, concretization is performed (Lines 17 - 20) by choosing a new solution ψ' from the set of possible solutions (Line 18). This solution indicates an execution of p_i that concretizes the behavior defined for p_i in the counterexample ψ. The current counterexample ψ is augmented by ψ' to include the execution within p_i as well (in Line 19). Furthermore, since the call sites of p_i had been abstracted when it was checked, the consistency of its callees has to be checked. Therefore, the procedures called in ψ' are added to the set P (Line 20) to be checked later.

If all the call sites needed to be checked are consistent with the current counterexample ψ, this counterexample is valid. The function $getTrace$ is then used (Line 23) to map ψ back to an execution in the original program. The result is assigned to the $trace$ variable and returned in Line 24.

It should be noted that abstractAnalyze is just an outline of the process. There are several opportunities for optimization in an actual implementation. We explain some of them in Section 5.

4.4. Properties

Termination. The abstractAnalyze algorithm contains two iteration points: a refine-solve cycle (Lines 3 - 16), and a validity check cycle (Lines 9 - 21). The syntax of Figure 11 allows analyzed programs to contain executions with infinite recursive procedure calls. The analysis of such programs can loop forever because the set of call sites to check can grow without bound. If the program execution corresponding to a found counterexample calls a recursive procedure p infinitely, all those call sites have to be checked for consistency with the counterexample. Therefore, checking the validity of the counterexample will not terminate. Thus, in general the analysis is not guaranteed to terminate.

However, if the analyzed program terminates on all inputs, the set of call sites to check is finite for all counterexamples. Therefore, the validity check cycle terminates. Moreover, the refine-solve loop is guaranteed to terminate, too. Intuitively, this is because the refinement phase monotonically extends the specification of some procedure. The specification is an abstraction of the code, so in the limit, it will be equivalent to the code. The following expresses this more precisely.

LEMMA 1. *The set of constraints C solved in the abstractAnalyze algorithm is bounded.*

Proof. From the over-approximation property of the abstract function, after the abstraction in Line 1,

$$C = (abstract(p, \pi_0, \gamma) \cup \neg A) \subseteq (translate(p, \pi_0, \gamma) \cup \neg A)$$

and after the assignment in Line 11,

$$C_i = abstract(p_i, \pi_i, \gamma) \subseteq translate(p_i, \pi_i, \gamma)$$

Since the call site π_i is reachable from the analyzed procedure p,

$$translate(p_i, \pi_i, \gamma) \subseteq translate(p, \pi_0, \gamma)$$

Thus, in Line 13,

$$S_i = core(C_i, \psi) \subseteq C_i \subseteq translate(p, \pi_0, \gamma)$$

which implies that throughout the analysis the following invariant holds:

$$C \subseteq translate(p, \pi_0, \gamma) \cup \neg A$$

Since translate generates a finite set of constraints, and the assertion A is finite, C is bounded. □

THEOREM 1. *Assuming that translate and solve operations always terminate, the refine-solve cycle is guaranteed to terminate.*

Proof. We show that whenever the set of constraints C is updated, it is strictly extended. That is, S_i (in Line 13) contains some constraints that are not included in C. Proof is by contradiction: Assume that $S_i \subseteq C$. By the assignment in Line 7,

$$\psi \in solve(C, \emptyset)$$

The assumption $S_i \subseteq C$ implies that

$$\psi \in solve(S_i, \emptyset)$$

By definition of solve,

$$\psi \in solve(S_i, \psi)$$

which implies

$$solve(S_i, \psi) \neq \emptyset$$

On the other hand, from the assignment in Line 13,

$$S_i = core(C_i, \psi)$$

which implies

$$solve(S_i, \psi) = \emptyset$$

which is a contradiction.

Therefore, C is strictly extended in each iteration of the refine-solve cycle. On the other hand, from Lemma 1, C is bounded. Thus, the number of refinements is finite, and the refine-solve cycle terminates. □

Completeness. An error detecting analysis is *complete* if and only if whenever there exists a counterexample to a given assertion, the analysis can find it. As discussed before, our method may loop forever if it is given a program which has a non-terminating execution. Therefore, in general, our analysis is not complete.

However, if the analysis terminates, we argue that our `abstractAnalyze` algorithm is as complete as a similar analysis method that does not abstract procedures, namely the `inlinedAnalyze` algorithm given in Figure 16. Therefore, the abstraction phase does not introduce incompleteness unless the analyzed program can loop forever. This is proved by the following theorem.

THEOREM 2. *If `inlinedAnalyze` finds a counterexample and `abstractAnalyze` does not loop forever, it finds a counterexample too.*

```
datatype Result = Valid + Invalid(Trace)

function inlinedAnalyze(Proc p,
                        ConstraintSet A, VarMap γ): Result {
    C = translate(p, π0, γ) ∪ ¬A
    if solve(C, ∅) = ∅
        return Valid
    ψ = choose(solve(C, ∅))
    trace = getTrace(ψ, γ)
    return Invalid(trace)
}
```

Figure 16. Analysis without abstraction

Proof. Assume to the contrary that inlineAnalyze *finds a counterexample, but* abstractAnalyze *terminates without any counterexamples. That is,* abstractAnalyze *terminates in Line 5 implying that*

$$solve(C, \emptyset) = \emptyset$$

However, by Lemma 1, the set of analyzed constraints in abstractAnalyze*, C, is bounded by the set of constraints solved in* inlinedAnalyze*. That is,*

$$C \subseteq translate(p, \pi_0, \gamma) \cup \neg A$$

Therefore,

$$solve(translate(p, \pi_0, \gamma) \cup \neg A, \emptyset) \subseteq solve(C, \emptyset)$$

Consequently, if $solve(C, \emptyset) = \emptyset$ then,

$$solve(translate(p, \pi_0, \gamma) \cup \neg A, \emptyset) = \emptyset$$

That is, inlinedAnalyze does not find any counterexamples either, contradicting the assumption. □

Soundness. A bug finding method is *sound* if and only if all the counterexamples it returns are feasible executions of the analyzed code that violate the analyzed assertion. We show that our abstraction phase does not introduce unsoundness. That is, the following theorem holds:

THEOREM 3. *If all counterexamples returned by* inlinedAnalyze *are feasible executions of the analyzed code, all counterexamples returned by* abstractAnalyze *are also feasible.*

Proof. Assume that $abstractAnalyze(p, A, \gamma)$ returns a counterexample, a trace t, corresponding to a logical solution ψ. According to

the algorithm, all procedures called in t have been checked for validity. Therefore, ψ satisfies all the constraints in $T = translate(p, \pi_0, \gamma)$ that encode those program moments that are executed in t. Any other constraint in T encodes an unexecuted program moment, a program moment whose guard condition evaluates to false under the logical assignment ψ. Since their antecedent conditions are false, these constraints are vacuously true. Thus, ψ satisfies all constraints in T, and its corresponding trace, t, can be returned by inlinedAnalyze, too. Since all counterexamples returned by inlinedAnalyze are feasible executions, t is also feasible. Therefore, all counterexamples returned by abstractAnalyze are feasible too. □

5. Framework Instantiation

In this section we explain our particular instantiation of the proposed framework. This instantiation is implemented in our tool and used in our experiments.

5.1. Inputs

We focus on checking object-oriented programs. Our tool currently supports a subset of Java that does not include exceptions, concurrency, or arithmetic expressions.

The analysis starts with a user-provided assertion that (partially) specifies the behavior of a procedure in a given program. We particularly target *structural properties*, i.e., properties that constrain the configuration of the heap. The given assertion is expressed in Alloy [21], a first order relational logic that includes transitive closure, making it well suited for expressing complex structural properties succinctly.

Programs are checked with respect to finitized heaps. The user bounds the size of the heap by specifying the maximum number of objects to consider for each type; the bound is fixed over the course of the analysis. If a counterexample is returned, it is guaranteed to be non-spurious. However, because the heap is finitized, absence of a counterexample does not constitute proof of correctness.

Since our tool supports recursive procedure calls, the validity check phase may be non-terminating. However, if it terminates, the analysis is sound. Also, it is complete within the given heap bounds.

5.2. Translation

We translate Java programs to boolean constraints in conjunctive normal form (CNF). This is done in two phases: the program is first translated to the Alloy language using a technique previously used in Jalloy [33]; the resulting Alloy formula is then translated to CNF using the Alloy translator [20]. This two-phase translation enables us to take advantage of the optimized boolean encoding offered by the Alloy translator. Examples of these optimizations are symmetry breaking and sharing detection, as explained elsewhere [29]. Here we explain each of the two translation phases briefly.

Java to Alloy. A Java program can be encoded as a conjunction of Alloy formulas whose satisfying solutions denote executions of the program. This encoding is based on a relational view of the heap: a field f of type T defined in a class C is viewed as a relation $f : C \rightarrow T$ that maps elements of type C to those of type T.

The translation starts with the control flow graph (CFG) of the code with nodes corresponding to the control points and the edges corresponding to the statements or control predicates. Figure 17(b) gives a CFG constructed for the small piece of program given in Figure 17(a).

A CFG edge connecting node u to node v is encoded in Alloy by a bit E_{uv} whose truth value indicates whether or not that edge is traversed during an execution. For a node v in a CFG, let $in(v)$ and $out(v)$ be the set of incoming and outgoing edges of v respectively. The control flow corresponding to this node is encoded by the following constraint.

$$\bigvee_{i \in in(v)} E_{iv} => \bigvee_{j \in out(v)} E_{vj}$$

That is, if either one of the incoming edges is traversed, at least one of the outgoing edges has to be traversed, too. The infeasible paths are ruled out by encoding the control predicates as explained below. The first group of constraints in Figure 17(c) encodes the control flow of the example CFG in Alloy. (The constraints are implicitly conjoined. The operator '+' gives the disjunction of bit variables.)

Since Alloy is a declarative language with no notion of mutation, a technique similar to static single assignment (SSA [6]) is used to encode value updates. That is, variables and fields are renamed whenever their values are changed. However, we reuse the same variable/field name in different paths of CFG as long as no path has two updates to the same variable/field instance. This constructs a mapping from occurrences of program variables to Alloy variables.

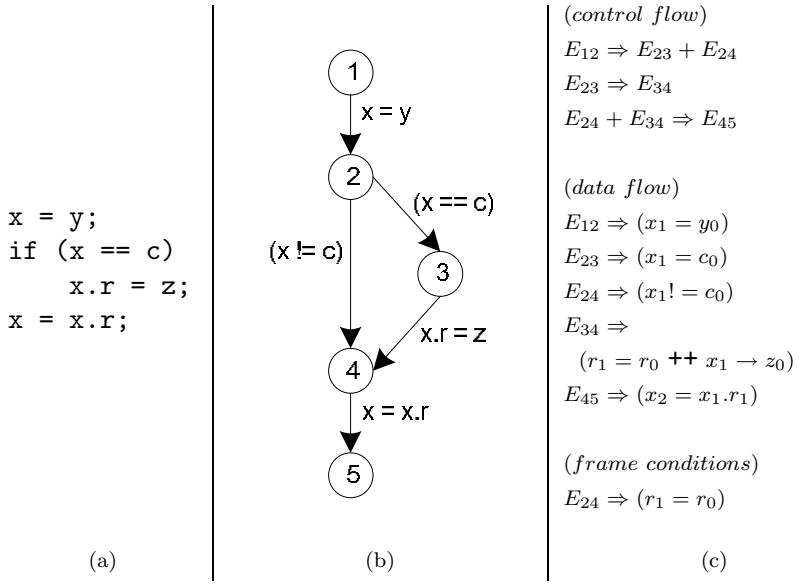

Figure 17. Translating sample code to Alloy: (a) a piece of code, (b) corresponding CFG, (c) Alloy encoding

The data flow corresponding to a CFG edge E_{uv} is encoded by a constraint

$$E_{uv} => f$$

where f is the formula encoding the corresponding statement or control predicate. This ensures that whenever an edge is traversed, the constraint associated with its behavior applies. The second group of constraints in Figure 17(c) are the data flow constraints of the example code. A variable/field name with a 0 subscript denotes the instance of that variable/field prior to this piece of code. The Alloy expression r `++` $(x \rightarrow z)$ gives a relation in which x is mapped to z while other objects have the same mappings as in r.

Because of the declarative nature of Alloy, an unconstrained variable can take any arbitrary value. Therefore, additional constraints, called *frame conditions*, are added to ensure that when a value is updated, other values remain the same. The last constraint in Figure 17(c) is a frame condition that specifies r does not change in the *else* branch. More details about this translation can be found elsewhere [33].

Alloy to Boolean. Since first order logic is undecidable, the translation of Alloy to Boolean is done within a finite *scope*, i.e. a user-provided finite bound on the number of atoms of each type. We use the heap size bounds provided by the user as the scope. The translation is sound and complete for the given scope.

If the scope of a type T is n, an array of n boolean variables is allocated for each variable of type T. For each relation $r : T_1 \rightarrow T_2$ a matrix of $n_1 \times n_2$ boolean variables is allocated where n_i denotes the scope of type T_i. This provides a variable mapping that maps Alloy variables to boolean variables. Using this mapping along with the variable mapping from the first phase of the translation, a variable mapping γ is constructed that maps different occurrences of program variables to matrices of boolean variables. This mapping is later used to translate boolean solutions back to the program executions.

The allocated boolean matrices are then combined into matrices of boolean formulas to represent different Alloy expressions. Alloy constraints are translated by combining these matrices into a single propositional formula.

The Alloy translator then converts the propositional formulas to CNF using a technique based on renaming all intermediate subformulas [26]. Details of the Alloy translation can be found elsewhere [29].

5.3. Abstraction

Initial specifications of procedure calls can be empty specifications that allow arbitrary behaviors. However, starting with more precise specifications can result in fewer refinements. On the other hand, since some procedures are totally irrelevant to the property being verified, computing detailed specifications is unnecessary and wastes resources.

The initial specifications that we compute for a procedure aims at preserving its frame conditions. That is, the value of any variable or field that is not modified by a procedure is preserved by its specification. Frame conditions cannot be computed statically with complete accuracy, of course. Therefore, we compute conservative specifications: no frame condition is generated for a variable or field that *may* be modified at a call site.

A Java method can modify the state of its callers by mutating objects' fields. The algorithm in Figure 18 determines what fields are updated by each method[1]. In order to take care of possible aliasings, if a procedure updates a field f of an object of type T, the initial specification allows f to change in all objects of type T. Therefore, the algorithm does not keep track of the objects whose fields are mutated; it only determines what fields can be changed.

This algorithm computes the set of all fields updated by each procedure directly or indirectly. This is stored in the *mutate* data structure. The algorithm first detects all the fields that each procedure updates

[1] Special cases, e.g. updating arrays which are passed as parameters, can be easily added to this algorithm.

```
datatype UpdateMap = ProcName → Set<Field>

function computeUpdates() : UpdateMap {
  mutate = {}
  foreach p_i ∈ Procedures
      mutate[p_i] = {x ∈ fields | p_i mutates x directly}
  changed = true
  while changed {
      changed = false
      foreach p_i ∈ Procedures
          foreach q_i ∈ callees(p_i)
              if mutate[q_i] ⊈ mutate[p_i] {
                  mutate[p_i] = mutate[p_i] ∪ mutate[q_i]
                  changed = true
              }
  }
  return mutate
}
```

Figure 18. Computing field updates of procedures

directly, i.e. without further procedure calls. Then, for each procedure p_i, all the fields updated by its callees are added to its set of mutations. This is repeated until a fixed point is reached, that is, the set of mutations does not grow for any of the procedures. Since the number of all fields is finite in a program, the set of mutations is bounded. Therefore, the algorithm terminates. It should be noted that in some special cases this algorithm can be optimized. For example, if the program does not have any recursive procedures, field updates can be computed based on the topological order [9] of the procedures instead of iterating until a fixed point is reached.

After computing the fields updated by each procedure, we compute an initial abstraction for the analyzed procedure p. The body of p is translated to Alloy as explained in Section 5.2 except that at each call to a procedure q, we allocate new Alloy variables for the relations (fields) that q may update and its returned value (if any). An example is shown in Figure 19. Variable/field names with a 0 subscript denote the variable/field instances in p prior to the given piece of code. As shown in Figure 19(b), since $mutate[q] = \{r\}$, after the call to q, a fresh relation instance, namely r_1, is allocated for the field r. Also, a fresh variable, $q\$return$, is allocated to hold q's returned value. Leaving these new variables unconstrained allows the constraint solver to assign

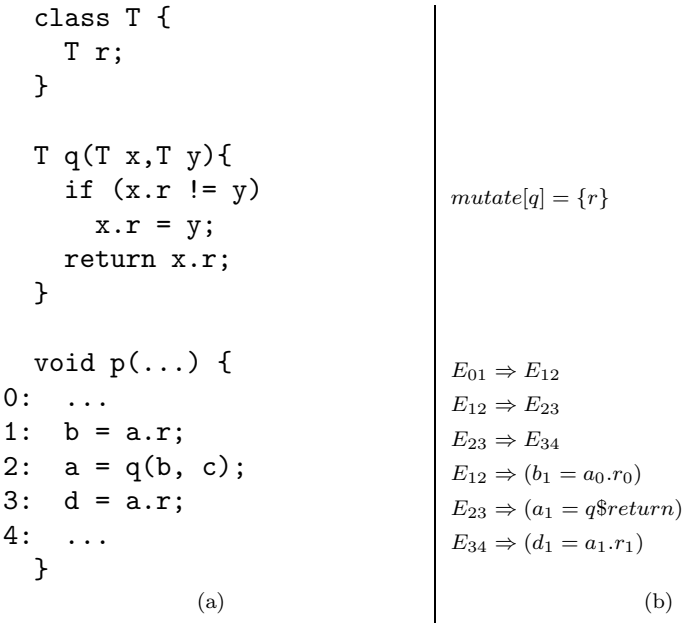

Figure 19. Abstraction example: (a) a piece of code, (b) abstraction of p in Alloy

arbitrary values to them. The generated Alloy model is therefore an over-approximation of the original program.

5.4. Solving

We use the ZChaff [25] SAT solver to check the CNF formula encoding the abstraction of the code and the negation of the assertion. A solution returned by the SAT solver assigns truth values to all boolean variables. Boolean variables assigned to *true* can be interpreted as symbolic values for the variables in the Alloy model based on the variable mapping (γ) constructed during the translation phase. The values of the initial variables give an initial state and the edge variables assigned to *true* give an execution path. Therefore, a solution gives an execution of the abstract program that violates the given assertion.

5.5. Validity Check

We check the validity of a solution by checking the procedures called in the program execution corresponding to that solution in the depth first order, i.e. the order in which they are actually called. Each called procedure is abstracted as explained in Section 5.3, substituting the actual parameters passed at the checked call site for the formal param-

eters. The generated CNF is conjoined with the solution being checked and solved again using ZChaff.

5.6. Refinement

Given an unsatisfiable CNF, ZChaff is capable of generating a proof of unsatisfiability called an *unsat core* [36], which is a subset of the given clauses that is unsatisfiable. Although the core generated by ZChaff is not necessarily minimal, it is usually much smaller than the original CNF.

If the conjunction of the CNF encoding a call to a procedure p and a solution ψ found as a counterexample is unsatisfiable, ZChaff generates an unsat core that encodes the reason for unsatisfiability. However, since this core is unsatisfiable, using it directly as a specification causes any assertion to be vacuously true. The following observation, however, enables us to extract a valid specification from the unsat core. If all clauses of the unsat core are contained in the formula encoding p, that formula itself is unsatisfiable. Due to our translation technique, this means that the user-provided bound on the size of the heap is not sufficient for any valid execution of p. This special case is handled by translating the unsat core back to the code statements and returning it to the user as a witness for insufficient heap objects. However, if the provided heap bound is big enough for some executions of p, the formula encoding p is satisfiable. Therefore, the unsat core must contain some clauses from both the formula encoding p and ψ. We remove the clauses contained in ψ from the unsat core and get a formula that encodes a partial specification of p that not only rules out the current solution ψ, but also rules out all counterexamples that execute the same path in p as ψ.

The resulting CNF formula can be conjoined with the formula encoding the caller of p, and be used in the following iterations of our analysis. However, we translate the resulting CNF back to Alloy using a technique described in a previous paper [30] and then re-translate it to CNF using the Alloy translator to exploit the symmetry breaking feature implemented in the Alloy translator.

6. Experiments

We used our tool to check the validity of some of the components we implemented for the backend of our tool itself. These components manipulate a directed graph data structure which is represented by lists of nodes and edges. Each node has pointers to sets of incoming

and outgoing edges. Since our tool currently handles only a subset of the Java language, these components have been simplified. Although the components are small, their correctness depends on complex manipulations of data structures in the heap. The analyzed components are as follows:

- List operation:
  - **removeList.** This procedure removes all the elements included in a list passed as an argument from the receiver list object. We checked this procedure against the subset property asserting that after executing removeList, the remaining set of elements in the receiver list object is a subset of the elements in the original list.
- Graph operations:
  - **removeNodes.** This procedure takes a list of nodes l and removes all nodes included in it from a graph structure. This is done by first removing all the edges adjacent to the nodes of l from the graph's edge list, and then removing the nodes of l from the graph's node list. Two properties were checked in this code, namely sameEdges and sameNodes, that specify a corner case in the removeNodes component. They assert that if the passed list of nodes to remove is empty, removeNodes does not change the graph's lists of edges and nodes.
  - **spanningTree.** This procedure computes a spanning tree for a graph. The algorithm is similar to the Kruskal's minimum spanning tree algorithm [9] except that the generated tree is not necessarily minimum. We checked the code against the consistency property which specifies a consistency condition of the underlying data structures. It claims that if an edge is included in the resulting spanning tree, both of its nodes are included too. We also checked a corner case specified by the deg0Node property. It asserts that if a node is disconnected from the rest of the graph, i.e., its edge degree is zero, it will not be included in the spanning tree by this algorithm.

Our experiments aim at showing the improvements gained by the specification inference approach. Therefore, we compare the performance of our tool to another version that inlines called procedures. In order to achieve a fair comparison, we shared code between the two versions whenever possible. Therefore, both versions use Alloy as an intermediate language, and the ZChaff SAT solver as the constraint solver.

Table I. Experiment results

component	property	loops / scope	without spec inference			with spec inference				speedup
			vars	clauses	time (sec)	vars	clauses	time (sec)	#refs	
remove List (40 loc)	subset	4/4	8216	18124	15	4928	10260	9	0	1.7
		5/5	14555	34704	162	8611	19002	98	0	1.7
		6/4	13554	30555	40	6702	14013	12	0	3.3
		6/5	18137	43760	234	9857	21776	83	0	2.8
remove Nodes (101 loc)	same Edges	4/4	66566	151323	164	6187	13507	8	0	20.5
		4/5	87710	214959	206	9524	23383	27	0	7.6
		5/4	–	–	> 900	6807	14794	8	0	> 112.5
		5/5	–	–	> 900	10346	25263	36	0	> 25
		6/4	–	–	> 900	7499	16207	9	0	> 100
	same Nodes	3/3	27147	56298	44	5927	11652	7	3	6.3
		4/4	66661	151489	123	11057	23450	13	3	9.5
		4/5	87803	215129	224	15682	36890	107	3	2.1
		5/4	108016	246914	359	13075	27446	17	3	21.1
spanning Tree (120 loc)	consistency	5/4	26917	80214	108	9219	26695	50	4	2.16
		5/5	37727	115109	348	13447	39918	124	6	2.8
		6/4	34425	102841	143	9948	28848	49	4	2.9
		6/5	48306	147732	328	14542	43227	189	6	1.7
	deg0Node	5/4	27191	81094	93	10481	30550	35	11	2.7
		5/5	38380	117285	178	13263	39948	83	10	2.1
		6/4	34699	103721	125	9901	29145	39	10	3.2
		6/5	48959	149908	491	16172	48439	93	13	5.3

The results of the experiments are given in Table I. All the checked properties are valid in the analyzed procedures. Therefore, no counterexamples are found. It should be noted that comparing the performance for cases with counterexamples is not conclusive because it depends on the SAT solver's strategy for finding a solution, whereas for cases with no counterexamples, the SAT solver has to exhaust the search space.

Since the version without specification inference requires loop unrolling, in order to use identical input programs to both versions, we tailored our tool to unroll the loops rather than expecting them to have been written as recursive functions. In Table I, the `loops/scope` column indicates the number of times the loops are unwound and the maximum number of objects considered for each type during the analysis. The numbers of variables and clauses given for the version that does not infer procedure specifications show the size of the generated boolean formula in CNF format; the time column gives the analysis time in seconds. The numbers of variables and clauses for our tool that iteratively infers specifications denote the size of the CNF generated after the last refinement. The time column gives the total analysis time,

including all refinements, in seconds. The refs column shows the number of refinements needed in each case. The last column summarizes the speedup gained as a ratio between total analysis times without and with specification inference.

As shown in Table I, in order to check the first two properties, the initial specifications that only preserve frame conditions are sufficient; no further refinements are needed. To check other properties, procedure specifications are refined a few times. In general, the number of required refinements depends on the amount of code that is related to the assertion being checked. However, as shown in Table I, the number of refinements is not predictable even when checking the same assertion. This is because the number of required refinements also depends on the order in which the solutions are found by the SAT solver: to rule out some invalid solutions, a rich specification is inferred that rules out other solutions, too.

Although in order to check each of the above properties only a small portion of code need be analyzed, the inlining version spends considerable time to translate the whole code into a boolean formula. Consequently, the generated formula is much bigger than needed and, therefore, harder for the SAT solver to check. As shown by dashes in the table, even the translation phase sometimes takes a long time.

Although current experiments involve small procedures, the improvement of analysis time gained by our specification refinement technique is considerable. Our technique improves the analysis time by (1) reducing the translation time, and (2) generating smaller boolean formulas that can be checked faster. More experiments have yet to be done to check the performance of our method on programs with deeply nested procedure calls.

7. Discussion

In this section we discuss some qualities, limitations, and possible improvements of our technique.

Target properties. Most program analysis techniques developed so far (e.g. SATURN [35], SLAM [3], and BLAST [18]) check programs against temporal safety properties that can be described as state machines. Our analysis, on the other hand, checks programs that extensively manipulate data structures stored in the heap against rich structural properties that constrain the configuration of the heap before and after the execution of a procedure. These properties are more similar to the kind of properties that shape analysis techniques (e.g. [1, 27, 28]) target. However, those techniques aim at *proving* a property

of a program, while our technique provides a lightweight analysis geared at finding bugs.

Modularity. Our analysis technique is modular. It checks each procedure using the specifications of its called procedures as surrogates for their code. Therefore, although it infers the specifications of called procedures automatically and does not rely on user-provided annotations, it can benefit from such annotations whenever available. That is, a user can optimize the analysis of a program by providing specifications for some procedures. This becomes handy in analyzing code with calls to methods whose source code is not available. It also enables us to construct optimized specifications for the methods provided by the Java library and use them in our analyses.

Applications. Our work is motivated by the observation that the full specification of a software system can often be written as a conjunction of partial specifications. Furthermore, even when specifying the full behavior of a system is prohibitively difficult, checking some partial specifications will increase one's confidence in the developed system.

Our technique is particularly useful for checking partial specifications. It provides a context-dependent analysis that extracts those parts of the called procedures that are relevant to the assertion; all irrelevant parts are ignored. Consequently, although our technique can be used to check a program against its full specification too, it may require several specification refinements to infer the full behavior of all procedures.

Limitations. As explained before, our current tool handles only a basic subset of the Java language. However, this is not an inherent limitation. Our technique is independent of how the code is translated to the logical constraints. Therefore, the subset of the language that our tool is able to analyze depends on the translation technique that is used in our tool. Several translation techniques (e.g. [13, 35]) are currently available that can be used in our tool. Experimenting with these techniques to handle a larger subset of the Java language efficiently is left as future work.

Possible improvements. Our current implementation computes initial specifications that only express the frame conditions of procedures. Starting the analysis with more precise specifications of called procedures can lower the number of required refinements. On the other hand, computing very detailed initial specifications is not necessarily cost-effective because some procedure calls are irrelevant (or only partially relevant) to the analyzed property. Obtaining initial specifications that are effective in ruling out some invalid executions, but not so costly to compute might improve the performance of our tool.

Furthermore, our current tool checks the validity of a counterexample by checking all of its called procedures in the order in which they are actually called. This phase might be optimized by first inferring some priorities for procedure calls based on their relevance to the analyzed property, and then checking procedure calls in that priority order. Investigating these ideas is left as future work.

8. Related Work

8.1. SAT-Based Analysis

Our method is inspired by previous work, Jalloy [33], and Sitaram's method [31], that translate a program to a boolean formula and use a SAT solver to check a property in a finite scope. However, they inline all called procedures that are not annotated with user-provided specifications. This severely limits their scalability as our experiments indicate.

Saturn [35] is another SAT-based error detection tool that translates a finitized program to CNF. It uses procedure summaries similar to a type signature for interprocedural analysis. In addition to a temporal safety property expressed by a state machine, the user provides a set of states along with the input and output predicates for each procedure. SATURN then computes a relation over the defined states that summarizes the behavior of a procedure. This computation is done by enumerating all possible state transitions and calling a SAT solver several times to check the feasibility of each transition. Saturn is different from our tool in that it unrolls loops and cannot handle recursive procedure calls. Furthermore, it targets temporal safety properties and in general is not capable of checking the kind of rich structural properties that we do.

MAGIC [5] is another modular software analysis tool that checks a program against a specification using a SAT solver. The specifications are expressed as labeled transition systems (LTS), finite state machines in which the transitions are labeled by actions. MAGIC computes a predicate abstraction [16] of the program using a theorem prover, and checks it against the specification. Unlike our method, MAGIC assumes that the user provides specifications for called procedures; all procedures with no specifications are inlined.

8.2. Software Model Checking

The software model checker SLAM [3] over-approximates the code using predicate abstraction. The abstraction is represented by a boolean program which is analyzed using the BEBOP model checker [2]. It is later refined by the NEWTON tool [4] which discovers additional predicates by analyzing the feasibility of paths in the program.

SLAM uses procedure summaries in Bebop to avoid computing the output of a procedure for the same input twice. Procedure summaries are similar to the procedure specifications we generate in that they (partially) represent the behavior of a procedure. However, unlike our method, SLAM uses procedure summaries only in the phase in which the validity of a found counterexample is checked. The abstraction and refinement phases are not based on the abstraction boundaries the procedures define.

BLAST [18] is similar to SLAM. It uses predicate abstraction and constructs a reachability tree which represents a portion of the reachable, abstract state space of the program. If an error node is reachable in the generated tree, the corresponding error is checked for validity. In case of invalidity, a theorem prover suggests new abstraction predicates which are used to refine the program. BLAST differs from SLAM in that it uses lazy abstraction and local refinements to achieve better performance. However, they both target temporal safety properties, and unlike our method, they cannot check rich properties about data structures stored in the heap.

Bandera [8] analyzes Java code by extracting a finite state model of code that can be mapped to the input format of several existing model checkers and theorem provers. The analyzed code is reduced to a relevant subset using static slicing techniques [17]. The analysis is done using existing model checkers such as Spin [19], SMV [24], and JPF [34]. Bandera differs from our technique in that it abstracts data based on the user-provided abstraction rules that may also yield false alarms.

8.3. Theorem Proving

The extended static checker for Java (ESC/Java) [15] checks a user-provided assertion in a finitized program. It translates code to Dijkstra's guarded commands [11] and generates verification conditions using optimized weakest preconditions. A specialized theorem prover, Simplify [10], is then used to check the result against the assertion. Failed proofs are turned into error messages and returned to the user. ESC/Java requires user-provided specifications for all procedures.

Inspired by ESC, Flanagan introduced a method [13] to check properties of code by translating it to a constraint logic program (CLP) [22]. This eliminates the need for user-provided procedure specifications. The resulting formula is checked for satisfiability iteratively based on predicate abstraction. An explicating theorem prover, e.g. Verifun [14], is used to infer new predicates upon finding spurious counterexamples. His method differs from ours in that it first translates the whole code into CLP and then checks its satisfiability iteratively. Our analysis framework might be instantiated with CLP and a proof-generating theorem prover like Verifun.

8.4. Shape Analysis

Like our technique, shape analysis algorithms (e.g. [1, 23, 27, 28]) can check properties about the structure of the heap. They typically represent the heap as a graph in which the nodes represent objects, and edges represent field relations connecting different objects. Parametric shape analysis (PSA) [28] uses a 3-valued logic to represent shape graphs and can prove properties without bounds. It starts with a set of possible input heap shapes and performs an abstract interpretation given the semantics of each statement in the program. It requires the user to specify how each statement affects each predicate of interest. The result of the analysis is a sound 'yes', 'no', or 'don't know'. Our method differs from PSA in that it does not require any user-provided annotations and does not give false alarms. However, unlike PSA, our method cannot prove that a property holds in the code; it only finds counterexamples.

8.5. Specification Extraction

Daikon [12] is a tool that detects likely invariants about programs. It works by running an instrumented program over a test suite and storing all the values taken by variables in those program executions. An offline analysis then processes these values for an extensive set of invariants at each program point. Although both Daikon and our tool infer partial specifications about programs, the specifications returned by Daikon are not necessarily valid in general; they are only valid with respect to the analyzed test suite.

9. Conclusions

In this paper we proposed a framework to statically check a user-provided property in code. The framework exploits the modular structure of the program and is based on constraint solving. We start with

a rough over-approximate specification for each procedure and refine it on-demand. While our method is capable of automatically inferring context-dependent specifications for procedure calls, it can still benefit from user-provided specifications, if available, to reduce the analysis time.

We also explained our implementation of the framework. We target Java programs and use Alloy as an intermediate language to translate Java to boolean constraints. Specification inference is based on the unsat core generated by the SAT solver ZChaff. While further experiments are needed to evaluate the performance of our technique on larger programs, current experiments show that, on small programs, procedure abstraction can considerably reduce the analysis time by analyzing only the parts of the code that are actually relevant to the verified property.

The technique proposed in this paper provides a unified approach in which a constraint solver is used for both the checking and the refinement phases. Furthermore, it can analyze recursive procedures. Therefore, if loops are written as recursion, the program can be analyzed without requiring loops to be unrolled a finite number of times in advance. Consequently, although the analysis is incomplete, the only source of incompleteness is bounding the heap.

Acknowledgements

We are grateful to Mandana Vaziri for helping us to use her algorithm, and for her advice and useful discussions, and to Sharad Malik and Zhaohui Fu for their help for using ZChaff. We would also like to thank the anonymous referees for comments that helped us significantly improve our paper. This work is supported by the National Science Foundation under Grant No. 0086154 and Grant No. 0325283.

References

1. Balaban, I., A. Pnueli, and L. Zuck: 2005, 'Shape analysis by predicate abstraction'. *In proc. of VMCAI*.
2. Ball, T. and S. Rajamani: 2000, 'Bebop: A Symbolic Model Checker for Boolean Programs'. *SPIN 2000 Workshop on Model Checking of Software* pp. 113–130.
3. Ball, T. and S. Rajamani: 2001, 'Automatically validating temporal safety properties of interfaces'. *SPIN Workshop on Model Checking of Software* pp. 103–122.
4. Ball, T. and S. K. Rajamani: 2002, 'Generating Abstract Explanations of Spurious Counterexamples in C programs'. *MSR-TR-2002-09* pp. 113–130.

5. Chaki, S., E. Clarke, A. Groce, S. Jha, and H. Veith: 2003, 'Modular Verification of Software Components in C'. *International Conference on Software Engineering*.
6. Chase, D. R., M. Wegman, and F. Zadeck: 1990, 'Analysis of Pointers and Structures'. *Proc. Programming Languages Design and Implementation*.
7. Clarke, E., O. Grumberg, S. Jha, Y. Lu, and H. Veith: 2000, 'Counterexample-guided abstraction refinement'. *Proc. International Conference on Computer-Aided Verification* pp. 154–169.
8. Corbett, J. C., M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng: 2000, 'Bandera: Extracting Finite-State Models from Java Source Code'. *Proc. International Conference on Software Engineering*.
9. Cormen, T. H., C. E. Leiserson, and R. L. Rivest: 1990, 'Introduction to Algorithms'. *MIT Press*.
10. Detlefs, D. L., G. Nelson, and J. B. Saxe: 2002, 'A Theorem Prover for Program Checking'. *Research Report 178, Compaq SRC*.
11. Dijkstra, E. W.: 1976, 'A Discipline of Programming'. *Prentice Hall, Englewood Cliffs, NJ*.
12. Ernst, M. D., J. Cockrell, W. G. Griswold, and D. Notkin: 2001, 'Dynamically Discovering Likely Program Invariants to Support Program Evolution'. *IEEE Trans. on Software Engineering* **27**(2).
13. Flanagan, C.: 2004, 'Software Model Checking via Iterative Abstraction Refinement of Constraint Logic Queries'. *Workshop on Constraint Programming and Constraints for Verification*.
14. Flanagan, C., R. Joshi, X. Ou, and J. B. Saxe: 2003, 'Theorem Proving Using Lazy Proff Explication'. *International Conference on Computer Aided Verification*.
15. Flanagan, C., K. Leino, M. Lillibridge, G. Nelson, J. Saxe, and R. Stata: 2002, 'Extended Static Checking for Java'. *Proc. Conference on Programming Language Design and Implementation* pp. 234–245.
16. Graf, S. and H. Saidi: 1997, 'Construction of Abstract State Graphs via PVS'. *Proc. International Conference on Computer Aided Verification* pp. 72–83.
17. Hatcliff, J. and M. Dwyer: 1999, 'Slicing Software for Model Construction'. *Proc. ACM Workshop of Partial Evaluation and Program Manipulation*.
18. Henzinger, T. A., R. Jhala, R. Majumdar, G. Necula, G. Sutre, and W. Weimer: 2002, 'Temporal-Safety Proofs for Systems Code'. *Proc. International Conference on Computer-Aided Verification* pp. 526–538.
19. Holzmann, G. J.: 1997, 'The model checker SPIN'. *IEEE Transactions on Software Engineering* **23**(5), 279–294.
20. Jackson, D., I. Schechter, and I. Shlyakhter: 2000, 'Alcoa: The Alloy Constraint Analyzer'. *Proc. International Conference on Software Engineering*.
21. Jackson, D., I. Shlyakhter, and M. Sridharan: 2001, 'A Micromodularity Mechanism'. *Proc. ACM SIGSOFT Conference on Foundations of Software Engineering*.
22. Jaffar, J. and M. J. Maher: 1994, 'Constraint Logic Programing: A survey'. *Journal of Logic Programming* **19**(20), 503–581.
23. Jeannet, B., A. Loginov, T. Reps, and M. Sagiv: 2004, 'A Relational Approach to Interprocedural Shape Analysis'. *In proc. of SAS*.
24. McMillan, K.: 1993, 'Symbolic Model Checking'. *Kluwer Academic Publishers*.
25. Moskewicz, M., C. Madigan, Y. Zhao, L. Zhang, and S. Malik: 2001, 'Chaff: Engineering an Efficient SAT Solver'. *Design Automation Conference*.

26. Plaisted, D. A. and S. Greenbaum: 1986, 'A Structure-Preserving Clause Form Translation'. *Journal of Symbolic Computation* **2**, 293–304.
27. Sagiv, M., T. Reps, and R. Wilhelm: 1998, 'Solving shape-analysis problems in languages with destructive updating'. *ACM Transactions on Programming Languages and Systems* **20**(1), 1–50.
28. Sagiv, M., T. Reps, and R. Wilhelm: 2002, 'Parametric Shape analysis via 3-valued logic'. *ACM Transactions on Programming Languages and Systems* **24**(3), 217–298.
29. Shlyakhter, I.: 2005, 'Declarative Symbolic Pure Logic Model Checking'. *Ph.D Thesis, Electrical Engineering and Computer Science Department, MIT.*
30. Shlyakhter, I., R. Seater, D. Jackson, M. Sridharan, and M. Taghdiri: 2003, 'Debugging Declarative Models Using Unsatisfiable Core'. *Automated Software Engineering.*
31. Sitaraman, M., D. P. Gandi, W. Kuchlin, C. Sinz, and B. W. Weide: 2003, 'The Humane Bugfinder: Modular Static Analysis Using a SAT Solver'. *Technical Report RSRG-03-05, Dept. of Computer Science, Clemson Univ.*
32. Taghdiri, M.: 2004, 'Inferring Specifications to Detect Errors in Code'. *Automated Software Engineering.*
33. Vaziri, M.: 2004, 'Finding Bugs in Software with a Constraint Solver'. *Ph.D Thesis, Electrical Engineering and Computer Science Department, MIT.*
34. Visser, W., G. Brat, K. Havelund, and S. Park: 2000, 'Model Checking Programs'. *Proc. IEEE International Conference on Automated Software Engineering.*
35. Xie, Y. and A. Aiken: 2005, 'Scalable Error Detection using Boolean Satisfiability'. *Proc. symposium on Principles of Programming Languages* pp. 351–363.
36. Zhang, L. and S. Malik: 2003, 'Validating SAT Solvers Using an Independent Resolution-Based Checker: Practical Implementations and Other Applications'. *Design, Automation and Test in Europe(DATE).*