Inferring Specifications to Detect Errors in Code

Mana Taghdiri
Computer Science and AI Lab
Massachusetts Institute of Technology
Cambridge, MA 02139
taghdiri@mit.edu

Abstract

A new static program analysis method for checking structural properties of code is proposed. The user need only provide a property to check; no further annotations are required. An initial abstraction of the code is computed that over-approximates the effect of function calls. This abstraction is then iteratively refined in response to spurious counterexamples. The refinement involves inferring a context-dependent specification for each function call, so that only as much information about a function is used as is necessary to analyze its caller. When the algorithm terminates, the remaining counterexample is guaranteed not to be spurious, but because the program and its heap are finitized, absence of a counterexample does not constitute proof.

1. Introduction

Software model checkers typically work by extracting a state machine from the code. Procedure calls are treated as control constructs; the abstraction boundaries that they represent are not usually exploited in the subsequent analysis. This is odd, since the modularization of the code into procedures was presumably chosen in order to make reasoning easier.

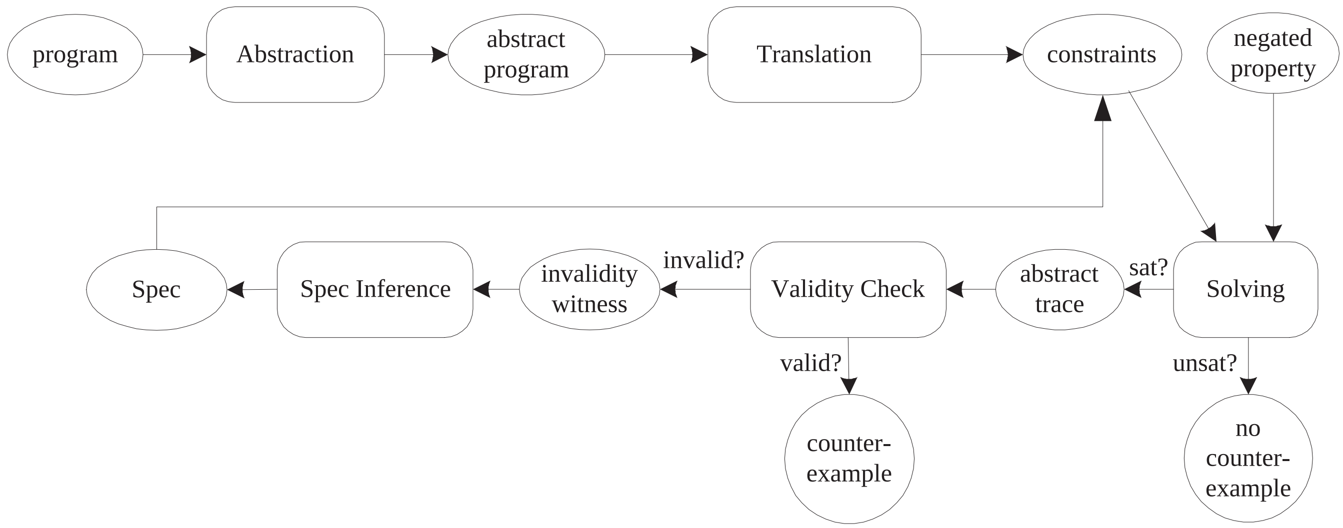
More traditional program verification approaches, in contrast, made extensive use of the program structure in structuring the analysis. Each procedure would be checked against its specification, using specifications of the called procedures as surrogates for their code. If these approaches could be automated, we might have the best of both worlds: a fully automatic analysis that exploits the modularity of the code.

This goal has motivated several tools. ESC/Java[9], for example, extracts verification conditions from a procedure, and presents them for proof (or refutation) to a specially tailored theorem prover. The tool has been applied successfully to substantial programs, but it suffers from an obstacle that limits its applicability. It turns out that the burden of writing specifications for the called procedures is considerable. To analyze a procedure at the root of a large tree, every procedure in the tree must be annotated. Jalloy[20], a counterexample detector for Java programs, suffers from the same problem, and although it can inline called procedures, such inlining does not scale.

This paper proposes a strategy to overcome this obstacle. A procedure-based analysis is performed that requires specifications of called procedures, but the specifications are inferred from the code rather than being provided by the user. Of course, extracting a specification that summarizes in full the observable behaviour of a procedure is not feasible. For this application, however, it is sufficient to capture only those aspects of the behaviour that are relevant in the context of the calling procedure. Our inference scheme exploits this. In fact, the inferred specifications are sensitive not only to the calling context, but also to the property being checked. As a result, a very partial specification is sometimes sufficient, because even though it barely captures the behaviour of the called procedure, it nevertheless captures enough to verify the caller. By starting with the roughest specification first, and refining it as needed, our scheme ensures that no more inference work is done than is necessary.

The fundamental idea underlying the strategy is a familiar one: counterexample-guided refinement of an abstraction[3]. The general scheme is as follows: (1) the analysis is applied to an abstraction of the code; (2) if no counterexample is found, the analysis terminates and has successfully verified the code (against the given property); (3) if a counterexample is found, it is checked for validity; (4) if the counterexample is valid, a fault has been discovered and the analysis terminates; otherwise (5) a more refined abstraction is computed, and the process is repeated.

This scheme has been applied in a number of different contexts[1, 12, 8, 7]. Our approach differs from all of these



in that the abstraction and its subsequent refinements follow the abstraction boundaries of the code itself. To our knowledge, all previous applications of this idea to software analysis involve refinement of predicate abstractions[10]. Our approach, in contrast, refines the specifications used to represent the behaviour of called procedures.

This paper therefore describes a framework for counterexample-guided refinement of procedure specifications. It assumes an underlying analysis in which counterexamples are found by solving constraints extracted from the code and specification. The framework itself is not dependent on any particular properties of the logic used, although we use the Alloy modelling language as the logic, and a SAT solver as the constraint solver. In order to handle undecidable properties of data structures, the program is finitized (by bounding the number of loop unrollings, as in ESC/Java and Jalloy), and the space of possible heaps is finitized too (by limiting the heap's size). 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, such as the method recently proposed by Flanagan[7].

2. Overview

Our analysis is focused on checking finite code against a property. Original code is finitized by unrolling loops and recursion some small number of times. A property is given as a (partial) specification of a procedure selected by the user. We particularly target *structural properties*, i.e., properties that constrain the configuration of the heap after the execution of a procedure. The property can be expressed in any language that can be converted to a set of logical constraints.

The above figure shows our framework. It consists of the following phases:

Abstraction: We construct an initial *abstract program* from the given program by replacing all procedure calls in the analyzed procedure with some approximate *specifications*. This abstraction is an over-approximation of the original code: all feasible executions of the original code are feasible in the abstract program, but not vice versa.

Translation: The abstract program is translated to a set of constraints, preserving the semantics of the code: any execution in the abstract code corresponds to a solution of the constraints and vice versa. Our method is independent of exactly how this translation is done, as long as it is semantics-preserving.

Solving: The constraints generated in the previous phase and the negation of the user-provided assertion are handed to a constraint solver. If no solution is found, the property holds in the abstract program, and thus in the original (finitized) program. If a solution is found, it is called an *abstract trace*, and denotes an execution of the abstract program that violates the assertion.

Validity Check: An abstract trace suggests a behavior for each eliminated procedure by assigning values to its inputs and outputs. The validity of each behavior is checked in the original program, again using a constraint solver. If a behavior is valid, the solution found in this phase is used to concretize the abstract trace. If the behaviors of all procedure calls are valid, the trace is a feasible counterexample and is returned, and the analysis terminates. However, if the inputs and outputs assigned to a procedure denote an invalid behavior, they represent an *invalidity witness*.

Specification Inference: A more precise specification is inferred for the procedure corresponding to the found invalidity witness. We use a constraint solver capable of generating proofs to construct a proof of invalidity for the witness. A specification that rules out the given invalid behavior is

```
class List {
  int val;
  List next;

  List(int v) {
      val = v;
      next = null;
  }
  /** assert:
  (l1 = null) || (l2 = null) => $return = null
  */
  static List intersect(List l1, List l2) {
0:  List res = null;
1:  while (l1 != null) {
2:    boolean found = contains(l2, l1.val);
3:    if (found)
4:      if (res == null)
5:        res = new List(l1.val);
6:      else
7:        res.add(l1.val);
8:    l1 = l1.next;
9:  }
10: return res;
  }
  void add(int v) {
    List c = new List(v);
    List l = this;
    while (l.next != null)
      l = l.next;
    l.next = c;
  }
  static boolean contains(List l, int v) {
    while (l != null) {
      if (l.val == v) return true;
      l = l.next;
    } return false;
  }}
```

Figure 1. Example

```
List.List(v):[$return != null &&
     $return.val <- ? && $return.next <- ?]
List.add(v):[?.val <- ? && ?.next <- ?]
List.contains(l, v):[$return <- ?]

                    (a)
class List {
  int val;
  List next;
static List intersect(List l1, List l2) {
0: List res = null;
1: if (l1 != null) {
2:  [boolean found <-?] //contains(l2,l1.val)
3:  if (found)
4:   if (res == null)
      //res = List(l1.val)
5:    [res!=null && res.val<-? && res.next<-?]
6:   else
7:    [?.val<-? && ?.next<-?]//res.add(l1.val)
8:   l1 = l1.next;
9: } <assert l1 == null>
10:return res;
}}
                    (b)
```

Figure 2. (a) Initial specifications of called functions. (b) Initial abstract program

then extracted from the proof and is conjoined with the old specification of the procedure. The process then starts over at the solving phase.

Each iteration of the algorithm monotonically extends the specification of some procedure. The specification is an abstraction of the code, so in the limit, the specification is equivalent to the code, and termination is therefore guaranteed.

2.1. Motivating Example

The example given in Figure 1 illustrates our method. The `intersect` function is selected for analysis. It takes two lists of integers and returns a list of the elements that appear in both of them. The given property asserts that if either one of the input lists is empty, the returned list is also empty.

For simplicity, assume that we finitize the code by unwinding each loop once. Figure 2.a shows the initial specifications computed for each called function. Question marks denote arbitrary values chosen non-deterministically; the arrow indicates a substitution.

The initial abstract program is presented in Figure 2.b. The line numbers shown in this figure correspond to the line numbers given in Figure 1. The specifications shown in Figure 2.a are inlined at the call points of their corresponding functions.

This abstract program and the negation of the given property are then translated to a set of constraints. The abstract trace shown in Figure 3 is found by a constraint solver as a counterexample. Each line in this figure consists of a line number, a program state and a statement from the abstract program that is executed. The line numbers correspond to the line numbers in Figure 2 and show the control flow of the execution. The program state at each line shows the symbolic values bound to the variables before the execution of the statement in that line. To save space, the unchanged values are not repeated.

In this trace, the `contains` function and the `List` constructor are called. Initially, the value of `l1.val` is the

```
 0: [l1.val = int0, l1.next = null,
     l2 = null] res = null;
 2: [res = null, ...] found <- ?
 5: [found = true, ...] res != null &&
         res.val <- ? && res.next <- ?
 8: [res.val = int1, res.next = null,..]
         l1=l1.next;
10: [l1 = null, ...] $return = res;
```

Figure 3. Abstract Trace for the Example

symbolic integer `int0`, and `l1.next` and `l2` are both `null`. Despite this, the call to `contains` returns `true`, and the call to the constructor assigns `int1`, rather than `int0` to `res.val`. Although acceptable in the abstract program, these behaviours are not feasible in the concrete program.

To determine this, the behavior assigned to the `contains` function is analyzed first, by checking the original code of this function against the values given to its inputs and outputs. Since the behavior is invalid, it is marked as an invalidity witness and no further validity checking is performed at this stage.

A constraint solver is then used to generate a proof for the invalidity of the inputs and outputs assigned to the `contains` function. The following specification is then extracted from the proof:

```
List.contains(l, v): [l <- l2 && v <- l1.val
  && (l = null => $return <- false) &&
  (l != null => ($return <- ? && l <- ?))]
```

In the first line of this specification, formal parameters of the `contains` function get the actual parameters used at the call site corresponding to the invalidity witness. (In our implementation, names are actually prefixed by unique scope identifiers to avoid name clashes.)

The generated specification only refines those parts of the `contains` function that are relevant to the found counterexample; the rest of the function is still abstracted, as shown by the third line of the specification. This new specification is inlined at the call point of the `contains` function and the process starts over. In this example, the analysis of the new abstract program generates no counterexamples. Thus, the process terminates and the property has been validated.

3. Basic Structures

3.1. Abstract Program

Syntax. An *abstract program* is constructed to check the correctness of a procedure selected by the user, called the *initial procedure*. Our framework can be applied to any programming language that supports procedure declaration and

```
program ::= classDcl*
classDcl::= class class{fieldDcl* procDcl*}
fieldDcl ::= class field;
procDcl ::= class proc(paramDcl*){stmt;}
paramDcl ::= class var,
stmt ::= class var | var = new class()
      |  var = expr | expr.field = expr
      |  if (pred) stmt else stmt
      |  return expr | OpaqueProcCall
      |  transparentProcCall | stmt; stmt
opaqueProcCall ::= <spec>
transparentProcCall ::= proc(expr*)
```

Figure 4. Syntax for Abstract Program

can be translated to logical constraints. We use a subset of Java to illustrate our approach.

As shown in Figure 4, an abstract program consists of a set of classes containing field and procedure declarations. A procedure body is a sequence of statements that can be local variable definitions, assignments, branches, return statements, or procedure calls. We assume that expressions are free of side effects. Since the analysis is done on a finitized program, there are no loops in our language.

A procedure call in an abstract program is either *transparent* or *opaque*. A transparent call is treated by inlining the procedure in the calling context during the analysis. An opaque call, on the other hand, is treated by replacing the call by a specification.

Semantics. We use *program points* to denote the control points in a program. A program point corresponding to a procedure call is called a *call point*. The set of program points and call points of a program are represented by Π and Ψ respectively ($\Psi \subseteq \Pi$). Furthermore, we define s_π to denote the statement from the original program that corresponds to a program point π. Thus, for a call point ψ, s_ψ denotes the procedure call corresponding to ψ.

A *program state* σ is defined at each program point π as a mapping from variables accessible at π to some values. The set of all possible states of a program is denoted by Σ.

Each program statement s is viewed as a transition in the state of the program and thus, is represented by a set of pairs of program states, i.e. $[\![s]\!] \subseteq \Sigma \times \Sigma$. A pair (σ, σ') is included in the set defining a statement s if and only if executing s in the state σ can result in the state σ'.

Transformations. Two transformations are defined on abstract programs: *close* and *open*. Given an abstract program, the close transformation makes all procedure calls opaque (i.e., abstracts procedure calls to their specifications). The open transformation, in contrast, takes an abstract program and a call point and makes that call point

```
spec ::= var <- expr | expr.field <- expr |
  var <- ? | expr.field <- ? | ?.field <- ? |
  var = expr | expr.field = expr | !spec |
  spec && spec | spec || spec | spec => spec
```

Figure 5. Syntax for Specification

transparent (i.e., inlines one procedure call).

3.2. Specification

A *specification* describes the behavior of a procedure in a calling context either exactly or over-approximately. As shown in Figure 5, a specification is a logical formula. The equals sign stands for equality predicate. The ← sign denotes substitution. A question mark in a substitution denotes an arbitrary value, which can be replaced with any value of the appropriate type non-deterministically. Thus, `var ← ?` allows the value of `var` to change arbitrarily whereas `?.field ← ?` allows an arbitrary change in the value of the given field in any object of the appropriate type. The logical operators `!`, &&, `||`, and `=>` represent negation, conjunction, disjunction, and implication respectively. Any variable or field not mentioned as the left hand side of any assignment in the specification of a procedure is assumed to have the same value before and after the procedure call.

3.3. Abstract Trace

An *abstract trace* denotes an execution of an abstract program represented by a sequence of pairs of program points and program states, i.e. $\overrightarrow{\Pi \times \Sigma}$. Two consecutive pairs (π, σ) and (π', σ') in a trace t mean that $s_{\pi'}$ is executed immediately after s_π, and that σ and σ' are the program states before the execution of s_π and $s_{\pi'}$ respectively. It should be noted that the program state of the first pair denotes the initial state of the program which defines an initial heap configuration. The program state of the last pair represents the final state of the program. The program point of the last pair is a dummy point indicating the end of the program.

In an abstract trace t, the state of the program at a point π is denoted by $state_t(\pi)$. Furthermore, for a program state σ, $succ_t(\sigma)$ gives the program state immediately following σ in t. (The final program state does not have a successor.)

A trace t is *valid* if and only if at each program point π included in t, the transition of the program state is consistent with the semantics of the statement corresponding to π as expressed in the original program. That is,

$$t \texttt{ is } valid \iff \forall(\pi, \sigma) \in t, (\sigma, succ_t(\sigma)) \in [\![s_\pi]\!]$$

δ[var = new class] = var ← ? (if var ∈ G)
δ[var = expr] = var ← ? (if var ∈ G)
δ[expr.field = expr] = ?.field ← ?
δ[if (pred) s else s′] = δ[s] && δ[s′]
δ[return expr] = $return ← ?
δ[proc(expr*)] = δ[stmt]
(where proc(var*){stmt;})
δ[s; s] = δ[s] && δ[s′]

Figure 6. Abstraction Rules

3.4. Invalidity Witness

An *invalidity witness* is a triple of a program point and two program states (π, σ, σ') where the state transition from σ to σ' is not consistent with the semantics of the original statement corresponding to π, (i.e. $(\sigma, \sigma') \notin [\![s_\pi]\!]$).

4. Basic Computations

4.1. Abstraction

During the abstraction phase, initial specifications are computed for all procedure calls. Initial specifications could allow any arbitrary behavior for the procedures. However, starting with more precise specifications can result in fewer refinements.

The initial specification that we compute for a procedure aims at preserving its *frame conditions*, i.e. any variable or field not mutated by the procedure is not mutated in the specification. However, not all frame conditions can be computed statically. For example, if a program uses dynamic dispatching so that different procedure bodies are bound to a single procedure call in different executions, computing exact frame conditions statically is impossible. Therefore, we compute conservative specifications: any memory location that *may* be changed by a procedure call is allowed to change.

In order to determine what memory locations may be mutated by a procedure p, all of its callees should be abstracted first. Thus, procedures should be abstracted in a certain order. We compute the order by first constructing the call graph g of the initial procedure. Since the program is finitized, g is a directed acyclic graph (DAG). we then compute a topological sort[5] for g that is an ordering l over all procedures so that all callees of a procedure p precede p in l. Therefore, procedures can be abstracted in the order they appear in l.

Figure 6 gives our abstraction rules. For a procedure p called at a program point ψ, the *global set* G_ψ denotes the set of all objects accessible both in p (the callee) and at ψ (the caller). Any change made by p to an object in G_ψ is visible to its caller.

```
procedure validityCheck(t:AbsTrace):
                                       Witness {
 callpoints = cpSet(t);
 forall ψ ∈ callpoints {
  p = s_ψ;
  pre = state_t(ψ);
  post = succ_t(pre);
  D̂ = toConstraint(pre) ∪
       toConstraint(post);
  P̂ = toConstraint(p);
  solution = solve(P̂ ∪ D̂);
  if (solution) {
   t = concretizedOf(t, ψ, solution);
   callpoints = callpoints ∪ cpSet(solution)
  } else return (ψ, pre, post);
 } return null;
}
```

Figure 7. Validity Check Routine

The abstraction function δ constructs a conservative specification for a procedure call based on its global set. In order to take care of possible aliasings in the original program, whenever a field f of an object of type T is modified, the specification allows a change in the field f of all objects of type T. The δ function is not applied to expressions since they are assumed to have no side effects.

After computing initial specifications, an abstract program is generated from the original program by annotating all procedure calls with their computed specifications. The close transformation is then performed to make all procedure calls abstracted.

4.2. Validity Check

A counterexample found in an abstract program is an abstract trace that should be checked for validity in the original program. Since the only abstracted statements are procedure calls, the only state transitions that may be invalid are those corresponding to call points. As our abstraction is based on the procedure call hierarchy of the code, the check for validity is also done hierarchically. A procedure q called within a procedure p is checked for validity only after the validity of the state transition corresponding to p has been validated.

Figure 7 shows how the validity of an abstract trace t is checked. The `cpSet` function takes an abstract trace and returns all of its call points as a set. For each call point ψ in t, the open transformation is applied to the abstract program to get the body of the procedure p called at ψ. All procedures called within p are still abstracted due to the semantics of the open transformation.

```
procedure SpecInference(ψ: CallPoint,
                σ, σ':ProgState): Spec {
 p = s_ψ;
 D̂ = toConstraint(σ) ∪ toConstraint(σ');
 P̂ = toConstraint(p);
 Ĉ = invalidityProof(P̂ ∪ D̂);
 return (conj(Ĉ − D̂));
}
```

Figure 8. Spec Inference Routine

Variables pre and $post$ denote the states of the program before and after the procedure p is called in the trace t. These states are translated to sets of constraints using the `toConstraint` function. The union of these constraints is denoted by $\hat{D}$. The body of the procedure p is also translated to logical constraints denoted by $\hat{P}$. This translation is semantics-preserving. However, since the callees of p are over-approximated, the generated constraints over-approximate the behavior of p.

A constraint solver is then used to find a solution satisfying $\hat{P} \cup \hat{D}$. A solution denotes a trace t' in p validating the assigned state transition. If a solution is found, the abstract trace t is concretized at the call point ψ by inlining t'. However, t' may introduce new call points corresponding to the procedures called in p. Since these call points are abstract, the validity of their corresponding state transitions in t' will be checked in the next iterations of the loop in the validity check routine. Although the exact order in which call points are checked may affect the performance of the analysis, it does not affect its correctness as long as all call points are eventually checked. If the state transitions corresponding to all call points are valid, no invalidity witness is returned and the trace t is a feasible counterexample.

However, if no solution exists for the constraints $\hat{P} \cup \hat{D}$ at some call point ψ, it means that executing the corresponding procedure in the given pre-state can not result in the given post-state. Therefore, the triple $(\psi, pre, post)$ is returned by the routine as an invalidity witness.

4.3. Specification Inference

Given an invalidity witness (ψ, σ, σ'), a more precise specification is generated for the procedure p called at ψ that rules out the invalid state transition from σ to σ'. Figure 8 shows the specification inference routine. The pre-state σ and the post-state σ' are translated to sets of constraints whose union is denoted by $\hat{D}$. The body of p is also translated to a set of semantics-preserving constraints $\hat{P}$.

Since the constraints $\hat{P} \cup \hat{D}$ is unsatisfiable, a constraint solver capable of generating proofs is used to find a proof of invalidity. The proof of invalidity for a set of constraints F highlights some constraints in F that are inconsistent. The set $\hat{C}$ denotes such inconsistent constraints extracted from $\hat{P} \cup \hat{D}$. Thus, $\hat{C}$ encodes the reason that the given state transition is not valid. However, $\hat{C}$ can not be used as a specification because it is unsatisfiable; using it in the analysis causes any assertion to be vacuously true. In order to get the specification, we need to extract a valid tautology from $\hat{C}$.

By definition, $\hat{C} \subseteq \hat{P} \cup \hat{D}$ and $\hat{C}$ is unsatisfiable. However, $\hat{P}$ is satisfiable because any execution of the procedure p is a solution to $\hat{P}$. Furthermore, $\hat{D}$ is also satisfiable because its constraints are all disjoint, i.e. each one defines the value of one variable. Therefore, $\hat{C}$ must include some constraints from both $\hat{P}$ and $\hat{D}$. That is, $\hat{C} = \hat{Q} \cup \hat{R}$ where $\hat{Q} \subseteq \hat{P}$ and $\hat{R} \subseteq \hat{D}$.

The set $\hat{Q}$ denotes the statements in p that show the values defined in $\hat{R}$ do not indicate a valid state transition. Furthermore, $\hat{Q}$ is satisfiable because $\hat{Q} \subseteq \hat{P}$. It can be extracted from $\hat{C}$ by comparing $\hat{C}$ against $\hat{D}$, i.e.

$\hat{Q} = (\hat{Q} \cup \hat{R}) - \hat{D} = \hat{C} - \hat{D}$

The conjuction of the constraints in $\hat{Q}$ is returned as the new specification to be conjoined with the old specification of p. The specification generated in this way is *context-dependent*, i.e. it only encodes those parts of p that are relevant to the found counterexample. The rest of the procedure is still abstracted.

5. Implementation

In this section we explain our particular instantiation of the proposed framework.

Inputs: We assume that the specification is written in the Alloy[14] language which is a first order relational logic that provides transitive closure operators, making it well suited for expressing structural properties. Furthermore, it is assumed that the input program is in Java.

Abstraction: In this phase, an Alloy specification is inferred for each procedure call. Since Alloy is a declarative language with no mutations, variables and fields are renamed whenever their values are updated. This technique was previously used in Jalloy[20].

Translation: The Java parts of an abstract program are translated to Alloy as explained in detail elsewhere[20]. In this translation, each control point in the Java program is encoded as a boolean Alloy variable. Java objects are encoded as Alloy variables and class fields of Java are encoded as Alloy relations. The generated Alloy formula is then conjoined with the initial procedure specifications.

Solving: The Alloy assertion provided by the user is negated and conjoined with the formula encoding the abstract program. The Alloy Analyzer[13] solves this formula by converting it to a boolean formula. This translation is sound. However, since first order logic is undecidable, the translation is done in a finite *scope*– a user-provided finite bound on the number of objects of each type. The translation is complete within the given scope.

We tailor the Alloy Analyzer to use ZChaff[16] as the back-end SAT solver. Any solution found by the SAT solver is an assignment of truth values to all boolean variables in the formula so that the whole formula becomes true. An abstract trace is inferred from a solution based on the truth values of the control flow variables in that solution.

Validity Check: In order to check the validity of an abstract trace, again we use ZChaff since it is capable of generating a proof of unsatisfiability called an *unsat core*[21]. Validity check is done as explained in the previous section.

Spec Inference: If no solution is found during the validity check of a procedure, ZChaff generates an unsat core. The input of ZChaff is a boolean formula in conjunctive normal form (CNF). A CNF formula is a conjunction of a set of *clauses* that are disjunctions of some *literals*. An unsat core is also in CNF format. It gives an unsatisfiable subset of the clauses in the input formula. Those clauses that encode program statements are extracted from the unsat core as explained before. They are then translated back to Alloy using a technique described in a previous paper[18]. The resulting Alloy formula is the inferred specification.

6. Experiments

We applied our method to check some properties of the code given in Figure 9. The code is inspired by our own implementation of the framework and has extensive structural manipulations.

The `NodeList` and `EdgeList` types are two linked lists defined as subclasses of `List` . The function `removeAll` removes all the elements of the given list from the receiver object. The type `Graph` defines a directed graph by its lists of nodes and edges. The sets of incoming and outgoing edges of each node are represented by `inEdges` and `outEdges` fields in `NodeListElem`. The `remove` function deletes the given list of nodes from the graph by first removing it from the `nodes` list and then removing all of the edges adjacent to any of those nodes from the `edges` list.

Figure 10 shows some of the properties checked in this code expressed in Alloy. In these properties, a primed field gives the value of the field after the function is executed whereas an unprimed one gives the value before the function execution. The * sign in Alloy denotes the reflexive transitive closure, i.e. it gives all the values reachable by traversing its following field zero or more times. Furthermore, `this` stands for the receiver object of a function.

The `subset` property is a specification for the `List.RemoveAll` function. The property asserts that the elements of a list after the execution of this function are a subset of its elements before the execution. In

```
class ListElem {
  int id;
  ListElem next; }
class List {
  ListElem first;

  void removeAll(List l) {
    ListElem e1 = first;
    ListElem prev = null;
    while (e1 != null) {
      int id = e1.id;
      if (l != null && l.contains(id)) {
        if (prev != null)
          prev.next = e1.next;
        else
          first = e1.next;
      } else
          prev = e1;
      e1 = e1.next;
    }}
  boolean contains(int id) {
    ListElem e = first;
    while (e != null) {
      if (e.id == id)
        return true;
      e = e.next;
    } return false;
  }}
class EdgeListElem  extends ListElem {
  EdgeListElem next; }
class NodeListElem extends ListElem {
  EdgeList outEdges;
  EdgeList inEdges;
  NodeListElem next; }

class EdgeList extends List {
  EdgeListElem first; }
class NodeList extends List {
  NodeListElem first; }

public class Graph {
  EdgeList edges;
  NodeList nodes;

  void remove(NodeList nl) {
    NodeList nds = nodes;
    nds.removeAll(nl);
    NodeListElem n = nl.first;
    EdgeList el = edges;
    while (n != null) {
      EdgeList e = n.outEdges;
      el.removeAll(e);
      e = n.inEdges;
      el.removeAll(e);
      n = n.next;
    }
  }}
```

Figure 9. Graph Manipulation Code

```
/** subset: List.RemoveAll */
this.first'.*next' in this.first.*next
/** sameEdges: Graph.remove */
no nl.first =>
  edges.first'.*next' = edges.first.*next
/** sameNodes: Graph.remove */
no nl.first =>
  nodes.first'.*next' = nodes.first.*next
```

Figure 10. Graph Manipulation Properties

other words, the `removeAll` function does not add new objects to the receiver list. The `sameEdges` and `sameNodes` properties are assertions for the `Graph.remove` function. They claim that if the input list of nodes is empty, the graph's lists of edges and nodes do not change by executing this function. These properties are valid in the given code and thus, no counterexample is found for any of them during the analysis.

We compare our analysis method with a static bug detector, Jalloy[20], since it is also based on SAT solvers and targets structural properties of Java code. The translation method used in Jalloy is identical to ours. Furthermore, we tailored Jalloy to use the same SAT solver as we do, i.e. ZChaff. However, Jalloy inlines all procedure calls to avoid user-provided specifications. This comparison therefore, shows the improvements gained by the procedure abstraction idea.

Table 1 gives the results of the experiments. LoopUnroll and Scope respectively show the number of times the loops are unwound and the number of objects of each type considered in the analyses. The number of variables and clauses given for Jalloy denote the size of the generated boolean formula in CNF format; the time column gives the analysis time. The number of variables and clauses for our method correspond to the largest boolean formula checked, i.e. the formula constructed after the last refinement. The time column gives the total analysis time including all refinements. The number of iterations shows how many refinements are needed to check each property.

The results show that to check the first two assertions, the initial specifications that only preserve the frame conditions are sufficient; no further refinements are needed. However, Jalloy spends considerable time on translating the whole code into a boolean formula although only a small portion of code is involved in verifying each of these properties. Consequently, the formula generated by Jalloy is too large to be handled by the SAT solver. Although more experiments should be done to see the performance of our method on large programs, current experiments show that it considerably improves the analysis time, even when some refinements are needed. This is intuitive because in real applications usually only a small number of procedures are

Table 1: Experiment Results

Assertion	LoopUnroll	Scope	Jalloy			Our Method			
			Variables	Clauses	Time (sec)	Variables	Clauses	Time (sec)	#iter
subset	4	4	8216	18124	15	4928	10260	9	0
	5	5	14555	34704	162	8611	19002	98	
	6	4	13554	30555	40	6702	14013	12	
	6	5	18137	43760	234	9857	21776	83	
sameEdges	3	3	27112	56241	61	3284	6589	5	0
	4	4	66566	151323	164	6187	13507	8	
	4	5	87710	214959	206	9524	23383	27	
	5	4	–	–	> 900	6807	14794	8	
	5	5	–	–	> 900	10346	25263	36	
	6	4	–	–	> 900	7499	16207	9	
sameNodes	3	3	27147	56298	44	5927	11652	7	3
	4	4	66661	151489	123	11057	23450	13	
	4	5	87803	215129	224	15682	36890	107	
	5	4	108016	246914	359	13075	27446	17	
	5	5	141087	347466	586	18549	42948	191	

needed to check each property and our method only translates those parts of the code that are necessary for the analysis. In this way both the translation time is reduced and a smaller boolean formula is generated that can be solved faster.

7. Related Work

Our method is inspired by previous work [20] and [19] 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.

The software model checkers SLAM[1] and BLAST[12] over-approximate the code using predicate abstraction[10]. An abstraction is refined by automatically inferring new predicates. They target temporal safety properties, and in general are not capable of checking the kind of structural properties that we do. MAGIC[2] is also based on predicate abstraction, but it uses a SAT solver to verify a user-provided specification in C code. However, if the user does not provide specifications for the called procedures, MAGIC will inline all procedure calls.

ESC/Java[9] uses a theorem prover to check properties of code relying on user-provided function specifications. An extension of ESC is proposed by Flanagan[7]. His method checks code properties via translation to a constraint logic (CLP)[15] and checking the satisfiability of the generated formula. It differs from our method in that it first translates the whole code into CLP and then checks for satisfiability iteratively based on predicate abstraction. We believe that our analysis framework can be used with CLP and a proof-generating decision procedure or a theorem prover like Verifun[8].

Bandera[4] analyzes Java code by extracting a finite state model of code, using slicing, which can be mapped into several model checkers and theorem provers. Unlike our method, it supports user-provided data abstractions that may also yield false alarms.

Shape analysis algorithms[17] can check properties about the structure of the heap. Parametric shape analysis[17] uses a 3-valued logic to represent shape graphs and can prove properties without bounds, but it may generate false alarms. It also requires the user to specify how each statement affects each predicate of interest. Our method, in contrast, does not require any user-provided annotations and does not give spurious counterexamples. However, the absence of a counterexample does not constitute proof.

Dynamic slicing (e.g. [22]) extracts the statements contributing to the value of a variable at some point in a given execution of a program. Our specification inference method is similar in that it extracts the statements relevant to the input and output values assigned to a procedure. However, since the execution path is not known, dynamic slicing can not be applied here.

Some specification extraction tools are developed before. Daikon[6] and DIDUCE[11], for example, detect invariants about programs. Unlike our static specification inference method, both of these tools detect invariants dynamically, i.e. by running the code. However, we do not generate general specifications. Our specifications are context-dependent, i.e. based on the property to be checked and on the context in which procedures are called. Furthermore, our specifications are only as precise as they need to be for the verification of their callers.

8. Conclusions

In this paper we proposed a framework to statically check a user-provided property in code. We specifically target the properties that constrain the structure of the objects in the heap. 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. Our experiments show that procedure abstraction can considerably reduce the analysis time by analyzing only the parts of the code that are actually needed to check a property. More experiments has yet to be done to evaluate the number of refinements needed in larger code.

Our initial abstraction currently infers initial specifications that only preserve the frame conditions of the procedures. A more precise initial specification can reduce the number of refinements needed and thus, improve the scalability of our method. Techniques to obtain such specifications will be studied in future.

9. Acknowledgments

We are grateful to Mandana Vaziri for the use of her code, and for her advice, and to Sharad Malik and Zhaohui Fu for their help using ZChaff. This material is based upon work supported by the National Science Foundation under Grant No. 0086154 and Grant No. 0325283.

References

[1] T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. *Proc. POPL 2002*, January 2002.

[2] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. *International Conference on Software Engineering*, May 2003.

[3] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. *Proc. Computer Aided Verification*, pages 154–169, 2000.

[4] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from java source code. *Proc. International Conference on Software Engineering*, June 2000.

[5] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. Introduction to algorithms. *MIT Press*, 1990.

[6] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. on Software Engineering*, 27(2), February 2001.

[7] C. Flanagan. Software model checking via iterative abstraction refinement of constraint logic queries. *Workshop on Constraint Programming and Constraints for Verification*, March 2004.

[8] C. Flanagan, R. Joshi, X. Ou, and J. B. Saxe. Theorem proving using lazy proff explication. *Internation Conference on Computer Aided Verification*, 2003.

[9] C. Flanagan, K. Leino, M. Lillibridge, G. Nelson, J. Saxe, and R. Stata. Extended static checking for java. *Proc. Conference on Programming Language Design and Implementation*, pages 234–245, June 2002.

[10] S. Graf and H. Saidi. Construction of abstract state graphs via PVS. *International Conference on Computer Aided Verification*, pages 72–83, 1997.

[11] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. *Proc. International Conference on Software Engineering*, pages 291–301, May 2002.

[12] T. A. Henzinger, R. Jhala, R. Majumdar, G. Necula, G. Sutre, and W. Weimer. Temporal-safety proofs for systems code. *Proc. International Conference on Computer-Aided Verification*, pages 526–538, 2002.

[13] D. Jackson, I. Schechter, and I. Shlyakhter. Alcoa: The alloy constraint analyzer. *Proc. International Conference on Software Engineering*, June 2000.

[14] D. Jackson, I. Shlyakhter, and M. Sridharan. A micromodularity mechanism. *Proc. ACM SIGSOFT Conference on Foundations of Software Engineering*, 2001.

[15] J. Jaffar and M. J. Maher. Constraint logic programing: A survey. *Journal of Logic Programming*, 19(20):503–581, 1994.

[16] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. *Design Automation Conference*, June 2001.

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

[18] I. Shlyakhter, R. Seater, D. Jackson, M. Sridharan, and M. Taghdiri. Debugging declarative models using unsatisfiable core. *Automated Software Engineering*, October 2003.

[19] M. Sitaraman, D. P. Gandi, W. Kuchlin, C. Sinz, and B. W. Weide. The humane bugfinder: Modular static analysis using a sat solver. *Technical Report RSRG-03-05, Dept. of Computer Science, Clemson Univ.*, June 2003.

[20] M. Vaziri. Finding bugs in software with a constraint solver. *Ph.D Thesis, MIT*, February 2004.

[21] L. Zhang and S. Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. *Design, Automation and Test in Europe(DATE)*, 2003.

[22] X. Zhang, R. Gupta, and Y. Zhang. Precise dynamic slicing algorithms. *Proc. International Conference on Software Engineering*, pages 319–329, 2003.