Predicate Abstraction in Java Pathfinder

Jakub Daniel
Charles University in Prague
daniel@d3s.mff.cuni.cz

Pavel Parízek
Charles University in Prague
parizek@d3s.mff.cuni.cz

Corina S. Păsăreanu
Carnegie Mellon/NASA Ames
corina.s.pasareanu@nasa.gov

ABSTRACT

We present our ongoing effort to implement predicate abstraction in Abstract Pathfinder, which is an extension of Java Pathfinder. Our approach builds upon existing abstraction techniques that have been proposed mainly for low-level programs in C. We support predicates over variables having numerical data types. The main challenges that we have addressed include (1) the design of the predicate language, (2) support for arrays, (3) finding predicates affected by a given statement, (4) aliasing between variables, (5) propagating values of predicates over method call boundaries, and (6) computing weakest preconditions for complex predicates. We describe our solution to these challenges and selected details about the implementation. We also discuss our future plans and research ideas.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification

General Terms

Verification

Keywords

Java Pathfinder, state space traversal, predicate abstraction

1. INTRODUCTION

Explicit-state model checking is a popular approach to program verification and bug finding. Tools using this approach can check systematically the behavior of a program for given test inputs and discover property violations (errors). Examples of such tools include SPIN [6] and Java Pathfinder (JPF) [7]. The focus of our work here is on JPF, which targets Java bytecode programs.

The main practical drawback of JPF is that it performs an exhaustive traversal of a program state space, and this is prone to state explosion. Although JPF supports many optimizations, checking program behavior under all possible test inputs with concrete (explicit) execution is time-consuming and requires a lot of memory.

One solution is to use *data abstraction* to reduce the large domains of selected program variables to smaller domains and make program verification via state space traversal more feasible. Last year we have developed a first version of Abstract PathFinder [8], a project extension to JPF that implemented a simple form of data abstraction. In that work, the large numeric domains of program variables are replaced with smaller abstract domains, and concrete operations on the domain are replaced with corresponding abstract operations. Both the abstract domain and operations are provided manually by the user.

A popular kind of data abstraction, that allows increased automation, is *predicate abstraction* [5]. The technique maps the numeric domains of program variables to a small abstract domain, as defined by a set of abstraction predicates. The abstraction predicates describe conditional relationships between program variables and their values represent the abstract program states over which the state space traversal is performed. Abstract operations capture the effects of individual program statements (concrete operations) on the values of abstraction predicates. The abstract operations are derived automatically with the help of an off-the-shelf decision procedure, e.g. using weakest preconditions. Given a particular concrete statement s and an abstraction predicate p, the weakest precondition $WP(s, p)$ for p is a logic formula that must be true before s for the predicate p to hold after the execution of s. Automatic counterexample-based abstraction refinement [4] can be further used to enrich the set of abstraction predicates.

We illustrate the state space reduction that can be achieved with predicate abstraction on the example Java program in Figure 1. Execution of the while loop can be fully described by several predicates, such as $\mathrm{o1.f} > 0$ and $\mathrm{o1.f} = \mathrm{o2.f}$, which yield much smaller state space than the domain of the primitive type int in Java.

Many techniques and tools involving predicate abstraction have been created in the past, but they target mostly low-level programs in C. The most prominent examples are SLAM [2] and BLAST [3].

1.1 Contribution

In this paper, we present our current project whose goal is to implement support for predicate abstraction of Java programs in Abstract Pathfinder. We describe the main challenges and our solutions to them.

This project builds upon the first release of Abstract PathFinder [8].

2. JAVA PATHFINDER

Java Pathfinder (JPF) [7] is a framework for exhaustive state space traversal of Java programs. The core of JPF is implemented as a special Java virtual machine that supports backtracking, state matching, and non-deterministic choices. JPF constructs the program state space on-the-fly during execution of the given program in the virtual machine. It makes non-deterministic choices at interesting points during program execution — (i) data choices in user-defined test drivers and (ii) thread scheduling choices at bytecode instructions that access or modify global state visible to multiple threads. A transition in the state space is a sequence of bytecode instructions executed by a single thread, where the first instruction in the sequence represents a non-deterministic choice. At every transition boundary, JPF saves the current JVM state (i.e., the full program state) in a serialized form for the purpose of backtracking and state matching. Changes of the JVM state are performed inside the interpreter of bytecode instructions.

Plain JPF contains a *concrete interpreter*, which models faithfully the behavior of all Java bytecode instructions and operates upon concrete values of program variables.

```
package pkg;

class T {
    int f;

    T() { this.f = 42; }

    void load(int a) {
        f = a − 10;
    }
}

class Example {
  static T o1 = new T();

  public static void main(String[] args) {
      int[] data = parseInputs(args);
      T o2 = new T();
      if (data.length == 1) o2.load(data[0]);
      int i = 0;
      while (o1.f > 0) {
          i = o1.f;
          if (o1.f == o2.f) i = −1;
          −−o1.f;
      }
      assert i > 0;
  }
}
```

Figure 1: Example program

3. PREDICATE ABSTRACTION OF JAVA PROGRAMS

The main goal is to support important features of Java (objects, classes, fields, arrays, local variables) and predicates over variables that have numerical data types (byte, short, char, int, long, float, double). We consider only predicates that represent constraints defined using linear arithmetic operators and relational operators.

3.1 Overview

We adapt the existing techniques developed for C. Here we provide an overview of the whole approach. Then, in the rest of Section 3, we describe selected technical details and our solution to the main challenges. We use the symbol APF to denote Abstract Pathfinder extended with the support for predicate abstraction.

An abstract program state consists of the program counter and the value $\nu(p) = \text{true} \mid \text{false} \mid \text{unknown}$ of each predicate p available in the current runtime scope (method call).

We use a single global container to keep the values of all predicates. The container is properly maintained during the state space traversal. A complete snapshot is created when APF makes a new choice, and saved. When APF backtracks, the corresponding snapshot is taken and the container is restored from it. The initial value of each predicate is unknown.

The other important data structure is the stack of symbolic expressions. We use it to determine abstract operands for bytecode instructions. A symbolic value on the stack represents the expression whose concrete value is at the corresponding location in the concrete stack frame.

When processing a bytecode instruction, APF must update the stack of symbolic expressions and values of affected predicates. All operands of the given instruction are removed from the stack and then the result (if there is any) is pushed onto the stack. For example, if the symbolic expression at the top of the stack is this.f and the current instruction is field read (getfield) on g, then APF replaces this.f with the expression this.f.g.

```
1  [static]
2  fread(f, sfread(o1, pkg.Example)) = 42
3
4  [object pkg.T]
5  fread(f, this) > 0
6
7  [method pkg.T.<init>]
8  this.f = 42
9
10 [method pkg.Example.main]
11 class(pkg.Example).o1.f > 0
12 class(pkg.Example).o1.f = i
13 class(pkg.Example).o1.f < i
14 class(pkg.Example).o1.f = o2.f
15 alength(arrlen, data) = 1
16 aread(arr, data, 0) = 35
```

Figure 2: Predicates for the example program

Only the assignment instructions (istore, putfield, . . .) really modify values of predicates. This is done is several steps:

1. APF finds all predicates that are possibly affected by the given assignment instruction (Section 3.3),
2. new values of the affected predicates are computed using the standard approach based on weakest preconditions and calls of the SMT solver (Sections 3.4 and 3.5), and
3. then the values of all the affected predicates are updated atomically to prevent inconsistencies.

The main challenge is to find all aliased variables (expressions) that may refer to operands of the given bytecode instruction.

A special case are the method call and return instructions. APF must correctly propagate values of predicates between the caller scope and callee scope (over the method call boundaries). More specifically, when executing a call to the method m, values of predicates over this and formal parameters of m in the callee must be set according to values of predicates over the actual arguments in the caller, and the stack of symbolic expressions for the callee scope must be properly initialized. Similarly, when processing a return instruction, values of predicates over the returned value and formal parameters of reference types must be propagated back to the caller. We give more details in Section 3.7.

Non-deterministic choices are created by APF only at branching instructions (if, switch) and comparison instructions (e.g., dcmp), if (i) the predicate corresponding to the branching condition has the value unknown and (ii) the precise deterministic value cannot be inferred with the SMT solver based on the current values of available predicates. APF does not make a choice right at assignment statement, where the given predicate gets the value unknown, but leaves this up to the nearest branching instruction that depends on the predicate value.

3.2 Predicate Language

We designed a predicate language that supports linear arithmetic, local variables, fields (both instance and static), accessing array elements, and reading the length of a given array. It is based on the quantifier-free subset of the first-order logic with the theory of arrays and linear arithmetic.

Figure 2 shows predicates (defined for the example program) that illustrate some of the key features of the predicate language. Each predicate belongs to a single context — static (line 1), object (line 4), or method (lines 7 and 10). Contexts represent different runtime scopes. Predicates defined in the static context can refer only to static fields and numeric constants, predicates defined in the object context can refer also to instance fields (accessed via this), and the predicates in method contexts refer also to local variables of specific methods (including parameters).

The language defines functions for expressing accesses to:

- fields of object instances (the function *fread*),
- static fields of classes (the function *sfread*), and
- array elements ($aread$).

These custom functions are internally modeled by the function select defined by the array theory. We also defined a special function alength (line 15), which represents the length of a given array.

As a syntactic sugar, our predicate language supports also the Java-like notation for expressing accesses to fields and array elements. Figure 2 demonstrates usage of this notation for field accesses at lines 8 and 11-14. When using the dot-notation, it is necessary to mark class identifiers (which include the package names) with the term class, so that APF can distinguish accesses to static fields of classes from accesses to instance fields.

A given predicate can refer to multiple heap objects (including arrays) and their fields. In particular, we support predicates over multiple dynamically created instances of a given class, as illustrated by the predicate at the line 14 of Figure 2.

Predicates defined in the object and method contexts are evaluated within the current runtime method call scope — the receiver object pointed to by this and local variables in the current method's stack frame. Note that a predicate defined in the object context for the class T is evaluated during every method call on every instance of the class T, but each time upon the corresponding object (method call receiver).

3.3 Finding Affected Predicates

In this section we describe our approach to identifying predicates that are possibly affected by an assignment instruction. Consider the assignment $v := e$, where v can be any access expression (local variable, instance field, static field, or an array element) and e can be any arithmetic expression over program variables. The task of identifying affected predicates is difficult because of (i) aliasing between different access expressions and (ii) possibly unknown precise values of index expressions for array element accesses.

Symbolic expressions s_v and s_e corresponding to v and e, respectively, are used for reasoning about the effects of the assignment instruction. The symbolic expressions represent (possibly multiple) concrete heap objects and values of primitive numeric data types. If a symbolic expression is an array element access, then it represents all elements of the given concrete array object due to analysis imprecision explained below.

The resulting set U of possibly affected predicates must contain all predicates that may refer to objects and values represented by s_v. We discuss several cases. First, the set U must contain all predicates directly referring to s_v. If s_v is a field access expression on a heap object, then for each prefix of s_v the set U must include all predicates over access expressions aliased with the prefix in the current program state. If s_v is an array element access, then the set U must contain all predicates that refer to the target array variable and all variables possibly aliased with it. Specifically, it must contain every predicate that refers to some array element through the respective array variables. For example, in the case of the assignment a[i] := e, where i is an arbitrary expression, APF may not know the precise value of i based on the current values of available predicates, and therefore it must consider predicates over all elements of the array variable a (and the aliases of a) as possibly affected by this assignment.

APF uses another global data structure — the *symbol table* — to identify aliased symbolic expressions and compute the set U for a given assignment instruction. The symbol table consists of two components: (1) a graph of all objects and values (static fields of classes, heap objects, arrays, values of primitive numeric types) that exist in the program state, and (2) a map from program variable names to graph nodes. The graph contains special nodes that represent local variables and static fields of classes. Edges in the graph associate objects with their fields and arrays with their elements. The map identifies nodes corresponding to local variable names and class names.

The whole symbol table associates symbolic access expressions with concrete heap objects and values of primitive numeric types. APF performs systematic traversal of the graph to get

- all concrete objects and values pointed to by a given access expression in the current scope, and
- all access expressions pointing to a given object or value (i.e., a set of aliased expressions).

When searching for aliases to a given access expression, APF explores only those paths in the graph whose length is bounded by the size of the longest access expression used in the available predicates. This way, we avoid infinite traversal over cyclic data structures, and at the same time we cannot miss any possibly aliased symbolic access expression refered to by some predicate.

When APF processes an assignment instruction, it updates the symbol table as follows. If every prefix of the expression s_v represents only a single object, the previous value of s_v is replaced with the set of possible values of s_e. If the expression s_v has a prefix that represents multiple objects, then all possible values represented by s_e are added into the set of values already represented by s_v. For example, in the case of the assignment a.b.c.d := e, where the prefix a.b.c represents two heap objects $\{o_1, o_2\}$, the symbolic expression a.b.c.d has multiple values $\{x_1, x_2\}$, and the symbolic expression e has the value p_3, then the new updated value of a.b.c.d would be $\{x_1, x_2, p_3\}$. Note that this must be done also for every alias of s_v.

If the symbolic expression s_v represents an array element, then the graph nodes and edges corresponding to all elements of the given array variable must be updated with the new values. For example, suppose that APF is processing an assignment a[i] = e, where a[0] $\rightarrow$ x, a[1] $\rightarrow$ y and e $\rightarrow$ z, then the updated mapping would be a[0] $\rightarrow$ {x, z}, a[1] $\rightarrow$ {y, z}.

3.4 Weakest Preconditions

We use the standard approach based on weakest preconditions to capture the effects of assignment statements on the values of predicates. Table 1 shows the weakest preconditions for simple predicates. The symbol v represents a local variable and u represents an arbitrary expression. A symbol of the form F_x denotes any atomic predicate over the expression x that is not covered by other lines of the table for a given statement.

For complex predicates with nested expressions that involve functions such as *fread*, weakest preconditions are derived by a recursive rewriting process. In each step, a particular expression in the given predicate is rewritten according to the template provided in Table 1. Consider for example the assignment o.f := e and a complex predicate $fread(f, o) + aread(\mathrm{arr}, a, fread(f, p)) > 5$. Both *fread* expressions would be rewrit-

Statement s	Predicate p	$WP(s,p)$
v = v'	F_v	$p[v'/v]$
v = e	v relop u	e relop u
o.f = e	$\mathit{fread}(f,o')$ relop u	$\mathit{fread}(\mathit{fwrite}(f,o,e),o')$ relop u
o = new C	$o = v$	false
	$\mathit{fread}(f,o)$ relop u	false
a[i] = e	$\mathit{aread}(\text{arr},a',i')$ relop u	$\mathit{aread}(\mathit{awrite}(\text{arr},a,i,e),a',i')$ relop u
a = newarray[e]	$a = v$	false
	$\mathit{aread}(\text{arr},a',i)$ relop u	$\mathit{aread}(\mathit{awrite}(\text{arr},a,\mathit{fresh}),a',i)$ relop u
	$\mathit{alength}(\text{arrlen},a')$ relop u	$\mathit{alength}(\mathit{store}(\text{arrlen},a,e),a')$ relop u

Table 1: Weakest preconditions

ten according to line 3 of the table, yielding the weakest precondition $\mathit{fread}(\mathit{fwrite}(f,o,e),o) + \mathit{aread}(\text{arr},a,\mathit{fread}(\mathit{fwrite}(f,o,e),p)) > 5$.

The weakest preconditions also reflect possible aliasing between variables. For example, consider the statement a[i] := e and the predicate $\mathit{aread}(\text{arr},a',i')$ relop u at line 6 of Table 1. Truth value of the corresponding weakest precondition depends on the values of equality predicates $a = a'$ and $i = i'$ over the access expressions. Such equality predicates capture aliasing.

We also use the symbol *fresh* in the weakest preconditions. This symbol represents a newly allocated object that is different from all the existing heap objects.

3.5 Using the SMT solver

APF runs the SMT solver on logic formulas with the structure $\bigwedge l(D) \Rightarrow WP(s,p)$, where $l(D)$ represents literals based on relevant predicates in the current abstract program state just before execution of the statement s. The set D is a transitive closure of predicates that (i) are available in the current runtime scope and (ii) share some access expression with $WP(s,p)$ or with some other predicate already in D. We consider each predicate in the set D as possibly relevant for computation of the new value of the given predicate p. For each predicate $d \in D$ the set $l(D)$ contains either d or $\neg d$ depending on the value of d in the current abstract program state. If some predicate $d \in D$ has the value unknown, then we add true (i.e., nothing) into the sub-formula $\bigwedge l(D)$.

The new value of the predicate p after execution of s is

- true if the formula $\bigwedge l(D) \Rightarrow WP(s,p)$ is valid,
- false if the formula $\bigwedge l(D) \Rightarrow WP(s,\neg p)$ is valid, and
- unknown otherwise, when the SMT solver does not give a precise answer for the input formula.

The input to each call of the SMT solver contains also supporting auxiliary clauses, which express the semantics of Java and ensure the unique value of the symbol *fresh*. For example, there is an auxiliary clause $\mathit{alength}(\text{arrlen},a) \geq 0$ for each variable a of an array type. We use the auxiliary clause $\mathit{fresh} \neq e$ for every symbolic access expression e of a reference type.

3.6 Branching Instructions

When the program execution reaches a branching instruction (if, switch), APF must evaluate the associated condition c. If there exists a predicate representing c, then APF can use its current value. Otherwise, it has to evaluate the condition c using the SMT solver. The value of c may be unknown. In that case APF makes a non-deterministic choice at the branching instruction to enable subsequent exploration of both branches — one with c having the value true and the other for the value false.

In each branch of the instruction with the condition c, APF uses knowledge of the selected precise value of c to improve precision of the predicate abstraction. Either the value of the predicate representing c is updated if it exists, or a new predicate representing c with the selected value is added into the global container.

Values of other predicates are refined in a similar way. For each predicate p that has the value unknown, APF creates the set D_P of predicates that have a precise value and share some access expression with p. If the set D_p is not empty, then the value of p is recomputed using the SMT solver. The new value of p is true if the formula $l(D_p) \Rightarrow p$ is valid according to the SMT solver, and false if the formula $l(D_p) \Rightarrow \neg p$ is valid. This refinement process stops upon reaching the fixpoint over the values of all predicates.

```
if (a < 0) { //  $a < 0$ unknown, $a < 1$ unknown
   ... //  $a < 0$ true, $a < 1$ true
} else {
   ... //  $a < 0$ false, $a < 1$ unknown
}
```

Figure 3: Refinement of predicate values

Figure 3 illustrates the refinement on a simple example. Note that the predicate $a < 1$ gets the value true in the if-branch because the condition predicate $a < 0$ also has the value true in that branch.

3.7 Method Call Boundaries

Here we describe in more detail (1) how APF propagates the values of predicates over method call boundaries, and (2) necessary updates of important data structures (e.g., the symbol table).

Call parameters. At method invocation, values of predicates about formal parameters in the callee scope are derived from values of predicates about arguments in the caller scope in a way that resembles processing of assignment statements. For each pair (v_i, e_i) of the local variable v_i representing a formal parameter and the expression e_i representing the corresponding actual argument in the caller, APF evaluates the assignment $v_i := e_i$ with respect to current values of predicates referring to e_i, and then sets the values of all predicates referring to v_i. It uses the standard mechanism based on weakest preconditions and calls to the SMT solver.

Note, however, that this approach can determine precise values just for predicates that refer only to formal parameters $(v_1, \ldots, v_n)$, actual arguments $(e_1, \ldots, e_n)$, and other entities visible both in the caller scope and the callee scope — this includes static fields, integer constants, and the method call receiver. Predicates referring also to object fields via this and to local variables other than formal parameters in the callee will typically get the value unknown.

Return value. Values of predicates over the returned expression are propagated from the callee scope to the caller scope also in a way that resembles assignment. We define the special expressions ret1, ret2, ... that represent values returned from different method calls on the stack of symbolic expressions in the caller.

For a method that returns some value, a user can define predicates over the special symbol return in the corresponding method context. The symbol represents the expression that will be returned from the method. Values of predicates referring to this symbol are computed by APF just before it executes the return instruction.

Output parameters. When processing the return instruction, APF must also propagate values of predicates over formal parameters of a reference type back to the caller, and update values of predicates referring to corresponding actual arguments. Such formal parameters point to objects that are visible both from callee and caller, and fields of those objects can be modified in the callee.

However, the propagation must be done only for variables representing formal parameters that were not overwritten in the callee method. In all other cases, predicates referring to the actual argument in the caller will get the value unknown.

Internal data structures. The symbol table is updated at each method call boundary to reflect the new scope. A copy of the graph that is a part of the symbol table exists in each scope. It contains pointers to and from local variables, which is an information specific to a given scope (currently executing method).

Therefore, APF must do the following at a method invocation:

- efficiently create a new copy of the graph,
- drop nodes specific to the caller scope,
- create nodes that will represent local variables of the new callee scope, and
- replace names of local variables (including formal parameters in the callee scope and this).

The graph is in fact a global structure, to which only the local variable bindings are added in each scope.

After the return from a method, the copy of the graph associated with the caller scope, which contains the correct information regarding local variables of the caller, will be used again.

4. IMPLEMENTATION

We implemented support for predicate abstraction in a new version of Abstract Pathfinder (APF) [8]. It uses (i) a non-standard interpreter of bytecode instructions and (ii) the attribute system to maintain symbolic expressions associated with the concrete values. Other information used by the predicate abstraction is in the custom global data structures (e.g., the symbol table). We also reused some of the components introduced in the first version of Abstract Pathfinder and made only small changes to the system architecture.

We impose certain restrictions on the input Java programs. The right hand side of an assignment statement cannot have any side effects. Such assignments would break our current implementation based on symbolic access expressions. For example, the statement b := (a + 4) + (++a) would not be processed correctly because it contains store operation (++a) on the right hand side. We recommend to replace them with equivalent sequences of statements that use temporary variables, and to define necessary predicates over the temporary variables. The statement b := (a + 4) + (++a) can be replaced with the sequence t1 = a + 4 ; a = a + 1 ; b = t1 + a.

Debugging symbols must be present in Java class files that make the input program to allow proper functioning of APF. However, note that the debugging symbols are required only for the program-specific classes — not for the Java standard libraries.

5. FUTURE WORK

Our immediate priority is to implement state matching and serialization over predicate values. A simple option might be to compare boolean vectors of current values of all relevant predicates. We will also try an approach similar to abstract state matching with subsumption for symbolic execution [1] — create formulas based on the current values of relevant predicates and ask the SMT solver if the implication between the formulas holds. In addition, we will modify the serializer such that it ignores concrete values of program variables that are referenced in some predicates.

The list of possible extensions and research projects based on APF includes dynamic inference of predicates needed to verify properties, counterexample-based abstraction refinement (CEGAR), and usage of the symbolic execution machinery implemented in the Symbolic Pathfinder [9].

6. ACKNOWLEDGMENTS

This work was partially supported by the Grant Agency of the Czech Republic project P202/11/0312. This work was also partially supported by the Google Summer of Code program in the year 2013.

7. REFERENCES

[1] S. Anand, C.S. Pasareanu, and W. Visser. Symbolic Execution with Abstraction. Journal of Software Tools for Technology Transfer, 11(1), 2009.

[2] T. Ball, R. Majumdar, T. Millstein, and S.K. Rajamani. Automatic Predicate Abstraction of C Programs. In Proceedings of PLDI 2001, ACM.

[3] D. Beyer, T.A. Henzinger, R. Jhala, and R. Majumdar. The Software Model Checker BLAST: Applications to Software Engineering, Journal of Software Tools for Technology Transfer, 9(5-6), 2007.

[4] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In Proceedings of CAV 2000, LNCS, vol. 1855.

[5] S. Graf and H. Saidi. Construction of Abstract State Graphs with PVS. In Proceedings of CAV 1997, LNCS, vol. 1254.

[6] G.J. Holzmann. The Model Checker SPIN. IEEE Transactions on Software Engineering, 23(5), 1997.

[7] Java Pathfinder: framework for verification of Java programs. http://babelfish.arc.nasa.gov/trac/jpf/.

[8] A. Khyzha, P. Parizek, and C.S. Pasareanu. Abstract Pathfinder. ACM SIGSOFT Software Engineering Notes, 37(6), 2012.

[9] C.S. Pasareanu and N. Rungta. Symbolic PathFinder: Symbolic Execution of Java Bytecode. In Proceedings of ASE 2010, ACM.