Abstract Pathfinder

Artem Khyzha
IMDEA Software Institute
artkhyzha@gmail.com

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 Abstract Pathfinder, an extension to the Java Pathfinder (JPF) verification tool-set that supports *data abstraction* to reduce the large data domains of a Java program to small, finite abstract domains, making the program more amenable to verification. We use data abstraction to compute an *over-approximation* of the original program in such a way that if a (safety) property is true in the abstracted program the property is also true in the original program. Our approach enhances JPF with an abstract interpreter and abstract state-matching mechanisms, together with a library of abstractions from which the user can pick which abstractions to use for a particular application. We discuss the details of our implementation together with some preliminary experiments with analyzing multi-threaded Java programs, where Abstract Pathfinder achieves significant time and memory savings as compared with plain JPF.

Categories and Subject Descriptors

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

General Terms

Verification

Keywords

Java Pathfinder, state space traversal, abstraction

1. INTRODUCTION

Exhaustive state space traversal techniques such as model checking are popular approaches to program verification and bug finding. Model checking is useful especially for analysing multi-threaded programs. Tools using this approach check all interleavings of program threads for property violations (errors). An example of such a tool is Java Pathfinder (JPF) [8] which targets Java bytecode programs.

The core of JPF is a special Java virtual machine that supports backtracking, state matching, and non-determinism in both data and scheduling decisions. JPF constructs the program state space on-the-fly during the execution of the program in the special virtual machine. 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 corresponding to a thread context switch. At every transition boundary, JPF saves the current JVM state (the program state) in a serialized form for the purpose of backtracking and state matching. The complete JVM state includes all heap objects, stacks of all threads and all static data. Changes of the JVM state are performed inside the interpreter of bytecode instructions, which too is a part of JPF.

Plain JPF contains a *concrete interpreter*, which models faithfully all Java bytecode instructions and keeps concrete values of program variables. We say that plain JPF performs *concrete execution* of instructions during the state space traversal.

The main drawback of JPF with respect to practical usefulness is that it performs an exhaustive traversal which is prone to state explosion. Although JPF supports many optimizations, including partial order reduction and other symmetry reductions, checking of all thread interleavings with concrete execution is time-consuming and requires a lot of memory.

A possible 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. Consider the example in Figure 1. It is a simple variant of the classic producer – consumer problem with a shared object of the Data class. The safety property of interest is absence of data races. Plain JPF would explore the program behavior for all possible values of the variable remaining, i.e. all integer values between 0 and 1000000, and the program state space would therefore be very large.

One can use the *signs* abstraction on the variable remaining to replace the large domain of the Java int type with a small finite domain { POS, ZERO, NEG }, which only encodes the *sign* of variable remaining, while abstracting away the actual value. Consequently, all program states that differ only in the value of the variable would be collapsed to three different states with the corresponding abstract values. The state space explored by JPF with such an abstraction would therefore be much smaller, reducing the time needed to verify the given safety property, while all program behaviors would be still analyzed.

1.1 Contribution

A lot of work has been done in data abstraction (e.g., predicate abstraction [1,2,6]), but only few approaches target Java. A notable exception is the Bandera toolset [4]. It performs finite state abstraction of a given Java program by the means of a source-to-source transformation based on the specific data abstractions selected (and defined) by the user.

In this paper, we present *Abstract Pathfinder* – an extension for

```
public class ProdConsExample {
  public static final int DAILY_LIMIT = 1000000;

  public static void main(String[] args) {
    Data d = new Data();
    new Producer(d).start();
    new Consumer(d).start();
  }
}

class Producer extends Thread {
  public void run() {
    int remaining = ProdConsExample.DAILY_LIMIT;
    while (remaining > 0) {
      synchronized (d) {
        d.value = 10;
        d.isNew = true;
      }
      remaining−−;
    }
  }
}

class Consumer extends Thread {
  public void run() {
    while (true) {
      synchronized (d) {
        if (d.isNew) {
          d.isNew = false;
          System.out.println(d.value);
        }
      }
    }
  }
}
```

Figure 1: Example: producer – consumer

JPF that supports abstraction of numeric data types in Java. The general goal is the same as in [4], but contrary to the Bandera toolset, our approach is based on a custom interpreter of bytecode instructions. In our approach the abstract values are propagated dynamically, during execution, using JPF's attribute mechanisms, thus eliminating the need to type propagation that was necessary in Bandera (to discover which variables and operations to instrument). We further remark that the Bandera project is no longer maintained. We implemented and evaluated several helpful abstractions, including signs and intervals. Other abstractions can be added easily by the user, as we provide a generic and extensible abstraction mechanism. In the next section we describe the main ideas behind Abstract Pathfinder, and then we define supported abstractions. Finally, we describe our implementation, a brief evaluation and we give conclusions. The source-code for our project is availabe from Bitbucket. We plan to make it available from the JPF main open-source repository soon.

2. ABSTRACT MODEL CHECKING WITH JAVA PATHFINDER

The basic ideas of our approach are (1) to use small *abstract data domains* for specific program variables instead of concrete types defined by the Java language, such as int and float, and (2) to replace the interpretation of concrete operations involving the abstracted program variables with a non-standard *abstract interpretation* of Java bytecode instructions that operate on the abstract domains, in such a way that the behavior of the abstracted program is an *over-approximation* of the behaviors of the original program. For example, the result of adding two POS values is POS, while the result of adding a POS and a NEG can be either POS, NEG or ZERO. We follow the theoretical framework of abstract interpretation [5].

One element of an abstract domain represents one or more concrete values (typically a high number of them). Every value in the original program (constant or a concrete value of a program variable) is mapped to a subset of the abstract domain. The size of such subset depends on how much information is available about the value in a program state.

We say that an operation upon a set of input values is abstract if at least one of the inputs has an abstract domain. The result of an abstract operation is always an abstract value. Note that due to over-approximation, the result of an abstracted operation may be a set of two or more abstract values, instead of just a single value (e.g. the result of adding POS and NEG is a set of values). In such a case, we introduce a non-deterministic choice between all the possible abstract values because the subsequent behavior of the program must be explored for all possible results of such operation.

The use of the abstract domains and the abstract interpreter means that Abstract Pathfinder verifies an abstract program that is an over-approximation of the original Java program. One state of the abstract program represents many states of the original concrete program. State matching during traversal considers the abstract values instead of the concrete values for variables that have an abstract domain. If a given safety property is true for an abstract program then it is also true for the original program. On the other hand, an error found in the abstract program may not exist in the original program. This happens when the error occurs on a spurious execution path that is not feasible in a concrete execution of the original program. Therefore, the counterexamples reported by abstract model checking need to be analyzed for feasibility.

The main general benefit of abstract model checking is better performance and scalability. The abstract domains are typically defined as much smaller than the ranges of concrete types, so that state space of the abstract program is much smaller than the state space of the original program, and therefore Abstract Pathfinder has to explore much less states to cover all program behaviors than in the case of the plain JPF and the original program.

3. SUPPORTED ABSTRACTIONS

Abstract Pathfinder provides an extensible library of abstractions for numeric data types of Java. The current version of the library contains the following abstractions: signs, evenness, and two variants of an interval abstraction.

3.1 Signs

The domain of the signs abstraction is the set { POS, ZERO, NEG }, whose elements express the fact that a value is positive, zero, or negative, respectively. A value in the original concrete program is mapped to one element of the abstract domain. Figure 2 shows an abstraction function for values of the Java type int.

```
Signs abstract(int v) {
  if (v > 0) return POS;
  if (v == 0) return ZERO;
  if (v < 0) return NEG;
}
```

Figure 2: Signs — abstraction function

The result of an arithmetic operation over two values such that at least one is abstract can be any subset of the abstract domain.

Figure 3 shows the abstraction of the operation + in the form of a Java-like pseudocode. Each instance of the class Signs represents a certain subset of the abstract domain. The procedures couldBeNeg, couldBeZero and couldBePos are used to check whether an operand contains the respective element of the abstract domain. The procedure constructResult creates an object of the class Signs that represents the result of the operation. If the value of the variable neg is true then the result must contain the abstract element NEG, and similarly for elements ZERO and POS. For example, the result of the operation over the abstract values POS and NEG can be any element of the abstract domain.

```
Signs plus(Signs right) {
  boolean pos = false, neg = false, zero = false;
  if couldBeNeg(this) {
    if couldBeNeg(right) neg = true;
    if couldBeZero(right) neg = true;
    if couldBePos(right) neg = zero = pos = true;
  }
  if couldBeZero(this) {
    if couldBeNeg(right) neg = true;
    if couldBeZero(right) zero = true;
    if couldBePos(right) pos = true;
  }
  if couldBePos(this) {
    if couldBeNeg(right) neg = zero = pos = true;
    if couldBeZero(right) pos = true;
    if couldBePos(right) pos = true;
  }
  return constructResult(neg, zero, pos);
}
```

Figure 3: Signs — abstract operation +

Abstractions of other arithmetic operations supported by Java are defined in a similar way.

3.2 Evenness

The domain of the evenness abstraction is the set { ODD, EVEN }, whose elements represent odd and even values, respectively. This abstraction can be used only for integer values (constants and program variables of Java types such as int and long), as the concepts of oddity and evenness do not make sense for floating-point values with non-zero decimal part.

3.3 Intervals

We support two variants of the interval abstraction. Both are parameterized with two user-defined values MIN and MAX.

The basic interval abstraction is defined as follows. The abstract domain for given two integer or floating-point values MIN and MAX is the set { LESS, INSIDE, GREATER }, whose elements express the fact that a value is less than MIN, between MIN and MAX, or greater than MAX, respectively. This abstraction can be used both for integer values and floating-point values (i.e., for constants and program variables of all primitive numeric types of Java, including long and double).

The second variant of the interval abstraction is more precise as it preserves concrete values in the interval [MIN, MAX]. The abstract domain for two integer values MIN and MAX is the set { LESS, MIN, MIN+1, ..., MAX-1, MAX, GREATER }. Note, however, that this abstraction is intended for use with small intervals.

Other abstractions can be defined similarly.

4. IMPLEMENTATION

We implemented Abstract Pathfinder as a JPF project extension. The project has the following components:

- Generic Abstraction and AbstractionBoolean classes. All the data abstractions are sub-classes of the Abstraction class.
- Library of abstractions.
- Abstract interpreter for all the numeric bytecodes. JPF's attribute mechanism is used for storing and propagating abstract values. It includes an AbstractInstructionFactory.
- FocusAbstractChoiceGenerator for implementing non-deterministic choice among multiple abstract values.
- AbstractionSerializer for abstract state matching.
- AbstractListener for printing the results.

We describe some of these components in more detail below.

4.1 The Abstraction class

Every abstraction must be implemented as a subclass of the Abstraction class that is used in the abstract interpreter of bytecode instructions (see Figure 4). The generic Abstraction class contains skeleton implementations of abstraction functions and helper methods for construction and processing of sets of abstract values, and it also defines several methods for which each particular abstraction must provide a custom implementation; the AbstractBoolean class contains a generic abstraction for boolean values. Abstract Pathfinder allows the user to pick specific abstractions from the library that are then used for a particular application. A new abstraction can be easily added to the library by extending the constructor of the AbstractInstructionFactory class with the abstraction's initialization code.

4.2 The Abstract Interpreter

The abstract interpreter redefines mostly bytecode instructions that perform arithmetic operations for all the primitive types. It operates upon the abstract values if they are available, and falls back to standard concrete interpretations otherwise. Abstract values are stored in attributes for local variables, stack operands, and object fields, and propagated between instructions via attributes. If some program variable has an abstract value then its concrete value is set to 0.

Figure 5 shows the implementation of an abstract interpreter for the IADD bytecode instruction that adds two integer values. Its description follows.

At first, it attempts to retrieve the abstract values of operands from the attributes, and passes them to the respective method of the Abstraction class which performs the actual addition. If the abstract value is not defined for any of the two concrete operands, the standard interpreter is called as a fall back. Finally, the concrete result value 0 is set and the abstract result value is stored as an attribute of the concrete result.

If the result of the arithmetic operation is a set of abstract values (i.e., not a single token), a non-deterministic choice over the values in the result set is created. For this purpose, we introduced a new type of a choice generator that we call *focus choice generator*. Subsequent behavior of the program is checked for all abstract values in the result set one by one. In each branch, one of the abstract values is stored in the attribute as the actual result of the operation. We note that we made our abstractions as precise as possible, e.g. adding POS and NEG results in a non-deterministic choice between POS, NEG and ZERO, while incrementing NEG results only

```
import java.util.Set;

public class Abstraction {
        ...
        public Set<Abstraction> get_tokens() {
                throw new RuntimeException("not␣implemented");
        }

        // returns number of tokens in abstract domain
        public int get_num_tokens() {
                throw new RuntimeException("not␣implemented");
        }

        boolean isTop = false;

        public boolean isTop() {
                return isTop;
        }

       // abstract_map methods need to be provided
       // by specific abstraction classes
        public Abstraction abstract_map(int v) {
                throw new RuntimeException("not␣implemented");
        }
        ...

        public Abstraction abstract_map(long v) {
                throw new RuntimeException("not␣implemented");
        }
        ...

        // abstract numeric operations
        public static Abstraction _add(int v1, Abstraction abs_v1,
                        int v2, Abstraction abs_v2) {
                Abstraction result = null;
                if (abs_v1 != null) {
                        if (abs_v2 != null)
                                result = abs_v1._plus(abs_v2);
                        else
                                result = abs_v1._plus(v2);
                } else if (abs_v2 != null)
                        result = abs_v2._plus(v1);
                return result;
        }
        public static Abstraction _mul(int v1, Abstraction abs_v1,
                        int v2, Abstraction abs_v2) {
                ...
        }
        ...
        // abstract comparison operations
        public AbstractBoolean _lt(Abstraction right) {
            throw new RuntimeException("lt␣not␣implemented");
        }
        ...
}
```

Figure 4: Generic Abstraction class

```
public class IADD extends gov.nasa.jpf.jvm.bytecode.IADD {
  public Instruction execute(
          SystemState ss, KernelState ks, ThreadInfo th) {

    StackFrame sf = th.getTopFrame();

    // retrieve abstract operands stored in the attributes
    Abstraction abs_v1 = (Abstraction) sf.getOperandAttr(0);
    Abstraction abs_v2 = (Abstraction) sf.getOperandAttr(1);

    Abstraction result;

    if (abs_v1 == null && abs_v2 == null) {
      // fall back to a concrete interpretation
      return super.execute(ss, ks, th);
    }
    else {
      int v1 = th.peek(0);
      int v2 = th.peek(1);

      result = Abstraction.add(v1, abs_v1, v2, abs_v2);

      if (!result.isSingleToken()) {
        // result is a set of abstract values
        ChoiceGenerator cg;
        if (!th.isFirstStepInsn()) {
          // first time seen −> create choice generator
          int size = result.getNumberOfTokens();
          cg = new FocusAbstractChoiceGenerator(size);
          ss.setNextChoiceGenerator(cg);
          return this;
        } else {
          // make the next choice −> return the result
          cg = ss.getChoiceGenerator();
          assert (cg instanceof FocusAbstractChoiceGenerator);
          int key = (Integer) cg.getNextChoice();
          result = result.getToken(key);
        }
      }

      // set the concrete result value to 0
      th.pop();
      th.pop();
      th.push(0, false);

      // set the abstract result
      sf = th.getTopFrame();
      sf.setOperandAttr(result);

      return getNext(th);
    }
  }
}
```

Figure 5: Abstract interpreter for IADD

in a non-deterministic choice between NEG and ZERO (since POS is not possible).

Note that both abstract values and concrete values are passed to the addition method of the Abstraction class. This is important for the case when an abstract value is defined only for one operand. The abstract value of the other (concrete) operand is computed inside the addition method.

Use of a custom instruction factory means that Abstract Pathfinder is not compatible with other JPF extensions that also use custom bytecode interpreter (factories).

Variables and constant values to be abstracted are marked in the program code which therefore has to be modified before the use of Abstract Pathfinder. For example, an initialization expression int x = 10 is replaced with int x = Debug.makeAbstractInteger(10). In the future, we will add support for defining abstracted variables in the .jpf configuration files.

4.3 Abstract State Matching

JPF uses a "serializer" to save the current JVM state into a compact form for the purpose of state matching. However, the serializer used in plain JPF takes into account only concrete values of program variables. To perform abstract state matching, we have implemented a custom serializer that processes also attributes that represent abstract values in addition to concrete values, and therefore enables proper consideration of abstract values in state matching.

```
int x,y,z;
x = 1; y = −1; z = 0;

// non−deterministic choice
boolean b = Verify.getBoolean();

if (b) {
  v = x + z;
}
else {
  b = true;
  v = y + z;
}

L1: // transition break and state matching

println("v␣=␣" + v);
```

Figure 6: State matching with abstract values

The program fragment in Figure 6 illustrates the need for a custom serializer that properly considers abstract values. Let x, y, and z be program variables for which Abstract Pathfinder uses the signs abstraction. Both branches of the if-else statement are explored because of the non-deterministic choice. The abstract value of v is POS at the end of the if branch and NEG at the end of the else branch. The concrete value of v, as set by the abstract interpreter of bytecode instructions, is 0 at the end of any branch. The concrete value of b is true at the end of any branch.

If the serializer from plain JPF is used then the println statement would be reached only once. Concrete values of all program variables are the same after both branches of the if-else statement, and therefore the state space search procedure would see an already visited state upon reaching the location L1 for the second time and backtrack prematurely.

However, the correct behavior is to reach the println statement twice, because the abstract value of v at the end of the if branch is different from the abstract value at the end of the else branch. If the custom serializer that processes abstract values is used, then the search procedure correctly sees a new state upon reaching L1 for the second time and continues exploration further.

5. EVALUATION

We performed experiments on small examples, including the producer – consumer example, to find how much the use of abstraction reduces the number of states that JPF must explore and its running time. We set the limit on memory usage to 512 MB.

	Time	Memory	States
plain JPF	> 45 s	> 512 MB	> 141680
Abstract Pathfinder	1 s	15 MB	155

Table 1: Experiments with producer – consumer

Results in Table 1 show that Abstract Pathfinder achieves significant time and memory savings compared to plain JPF (when using the Signs abstraction). Abstract Pathfinder explores the whole state space of the abstract program in one second, while plain JPF runs out of available memory after 45 seconds and processes much more states up to that point.

6. CONCLUSION

We described here Abstract Pathfinder, a new tool for performing data abstraction for Java programs. We gave the main aspects of its implementation, and provided an overview of the currently supported abstractions. Results of our preliminary experiments are very promising, but much work still has to be done to make Abstract Pathfinder even more useful.

The current version of Abstract Pathfinder allows to use only a single particular abstraction from the library. Our first priority is to add support for simultaneous usage of multiple abstractions. We plan to achieve this by implementing a *container* abstraction that will associate two or more abstract values with a concrete value.

In the future, we would like to extend the current abstractions such that they can model tricky aspects of numerical data types, such as integer overflows, infinite values, and precision of floating-point values (rounding). We also plan to support other kinds of abstractions, most notably predicate abstraction. We believe that JPF's symbolic execution framework, a.k.a. Symbolic PathFinder [7], could be leveraged to build such abstractions automatically. Finally we would also like to extend the tool beyond primitive types, to handle arrays and data structures, in a way similar to shape analysis [3].

7. ACKNOWLEDGMENTS

This work was supported by the Google Summer of Code program in the year 2012.

8. REFERENCES

[1] T. Ball, V. Levin, and S. K. Rajamani. A Decade of Software Model Checking with SLAM. Communications of the ACM, 54(7), 2011.

[2] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The Software Model Checker Blast. STTT 9(5-6), 2007.

[3] I. Bogudlov, T. Lev-Ami, T. W. Reps, and M. Sagiv. Revamping TVLA: Making Parametric Shape Analysis Competitive. CAV 2007.

[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. ICSE 2000, ACM.

[5] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In POPL 1977, ACM.

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

[7] C. S. Pasareanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. R. Lowry, S. Person, and M. Pape. Combining Unit-level Symbolic Execution and System-level Concrete Execution for Testing NASA Software. ISSTA 2008.

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