Symbolic Arrays in Symbolic PathFinder

Aymeric Fromherz
École Normale Supérieure
aymeric.fromherz@ens.fr

Kasper S. Luckow
Carnegie Mellon University
kasper.luckow@sv.cmu.edu

Corina S. Păsăreanu
Carnegie Mellon University,
NASA Ames
corina.pasareanu@sv.cmu.edu

ABSTRACT

Symbolic Execution is a program analysis technique used to increase software reliability. Modern software often manipulate complex data structures, many of which being similar to arrays. We present a novel approach and implementation in Symbolic PathFinder for handling symbolic arrays in Java. It enables analyzing a broader class of programs that manipulates arrays. We also extend the Symbolic Pathfinder testcase generation to support numeric arrays.

Keywords

Java PathFinder; Symbolic PathFinder; Symbolic Execution; Testcase Generation; Software Engineering

1. INTRODUCTION

Symbolic Execution (SE) [9] is a powerful program analysis technique, widely used in many application domains such as test data generation, partial verification, symbolic debugging, and program reduction. It is increasingly used not only in academic settings, but also in industry, such as in Microsoft, NASA, IBM and Fujitsu [3]. Instead of executing a program P with inputs as concrete values, SE executes the program on symbols, and computes the program effects as *functions* in terms of these symbolic inputs. There are many tools that support symbolic execution for programs [1, 2, 5]. We focus on Symbolic PathFinder—part of Java PathFinder tool set.

In this paper, we describe recent advancement for Symbolic PathFinder (SPF) [12], a symbolic execution tool for Java bytecode, part of the Java PathFinder tool-set[1]. In particular, we introduce a new support for symbolic arrays with a symbolic length using an array theory implemented in several solvers such as Z3 [6].

The previous version of SPF can handle arrays of fixed size, with symbolic elements and indices. The advantage of this simple support is that it works well with model counting too. The disadvantage, however, is that the nondeterministic choices introduced to handle symbolic indices often generates too many paths than can feasibly be explored. We address this, by extending SPF with a "more symbolic" handling of input arrays that leverage the theory of arrays in existing solvers, notably Z3.

Our approach provides a symbolic treatment of arrays (of unspecified length) including arrays of objects, treated using lazy initialization, which is distinguishing our work from previous approaches. We also extended SPF to support automatic JUnit tests generation for methods manipulating arrays. Finally, we re-implemented the constraints handling in SPF to allow the use JConstraints[2] [10] to benefit from previous work.

Our novel implementation allows the symbolic execution of a broader class of programs, and speeds up the execution of some others. Nevertheless, we observed an increase in states explored and analysis time in some specific cases.

2. SYMBOLIC ARRAYS

While formulating a mathematical science of computation, McCarthy proposed in [11] a basic theory of arrays that characterizes arrays using only two axioms. Using the notations *store(a, i, v)* to express that we store the value v in the array a at index i, and $a[i]$ to express the load of the element at index i in the array a, these axioms are:

$$\forall\, a, i, v, \mathit{store}(a, i, v)[i] \simeq i$$
$$\forall\, a, i, j, v, i \simeq j \vee \mathit{store}(a, i, v)[j] \simeq a[j]$$

The first axiom expresses that if we store an element v at index i in the array a, loading from array a at index i returns the element v. The second axiom expresses that storing an element v at index i in a does not modify any other element in a.

The SMT solver Z3 [6] implements the basic array theory presented by McCarthy as well as a powerful extension called *combinatorial array logic* [7]. It provides two operations on arrays: *select a i*, that returns the element stored at index i in the array a, and *store a i v* that returns a new array identical to a, but with v at index i.

To make support for the theory of arrays in SPF, we have to change the implementations of the array instruction—there are two types of array instructions in Java: *ALOAD instructions, that pop an array a and an index i from the stack and returns the element $a[i]$, and *ASTORE instruc-

[1]http://babelfish.arc.nasa.gov/trac/jpf/

[2]http://www.github.com/psycopaths/jconstraints

JPF-Workshop 2016
© 2016 ACM. ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

tions, that store an element in an array at a given index. We provide a symbolic implementation for each of the *ALOAD and *ASTORE instructions.

As part of this effort, we added support for JConstraints [10], a constraint solver abstraction layer that supports a variety of solvers, notably Z3 [6], Coral [13], and SMTInterpol [4]. Note, that we still expose the class PathCondition in SPF to ensure backwards-compability. Using JConstraints instead of the SPF backend is thus optional. JConstraints is maintained as a stand-alone library. The motivation for integrating JConstraints is that we can harness the several facilities for optimizing constraints, e.g. by adding auxiliary definitions and/or interpolations, and manipulating constraints.

We implement the ArrayExpression class from JConstraints, that derives from Variable. An ArrayExpression contains a symbolic integer representing the length of the array. We also add SelectExpression and StoreExpression to JConstraints. The SelectExpression contains an ArrayExpression a, an Expression<Integer> i and a value Expression<E> v, and expresses that $a[i] = v$. Similarly, a StoreExpression contains two ArrayExpression a and $a2$, a symbolic index i and a symbolic value v, and expresses that $a2 = a[i] \leftarrow v$. Using Z3 as a solver, these expressions are translated to (= (v (select a i))) and (= ($a2$ (store a i v))) respectively. We extend JConstraints-Z3 to support these expressions.

2.1 Arrays of Primitive Types

During the symbolic execution of a *ALOAD instruction, e.g. IALOAD or BALOAD, an instance of JPCChoiceGenerator is created. There are three paths to explore: (i) The index is strictly smaller than 0, which throws an ArrayIndexOutOfBoundsException; (ii) the index is greater than the length of the array; or (iii) the index is in bounds and we are loading an element.

The implementation of IALOAD is presented in Listing 1.

```
1  if (!ti.isFirstStepInsn()) {
2    cg = new JPCChoiceGenerator(3);
3      ...
4  } else {
5    cg = ti.getVM().getChoiceGenerator();
6  }
7  ...
8  if ((Integer)cg.getNextChoice()==1) {
9    pc._addDet(NumericBooleanExpression.create(
         indexAttr, NumericComparator.GE,
         se.arrayExpression.length));
10   if (pc.simplify()) {
11     ... // Throw ArrayIndexOutOfBoundsException
12   } else {
13     ... // Ignore state
14   }
15 } else if ((Integer)cg.getNextChoice()==2) {
16   pc._addDet(NumericBooleanExpression.create(
         indexAttr, NumericComparator.LT,
         Constant.create(BuiltinTypes.SINT32, 0)));
17   if (pc.simplify()) {
18     ... // Throw ArrayIndexOutOfBoundsException
19   } else {
20     ... // Ignore state
21   }
22 }
23 else {
24   pc._addDet(NumericBooleanExpression.create(
         indexAttr, NumericComparator.LT,
         arrayAttr.length));
25   pc._addDet(NumericBooleanExpression.create(
         indexAttr, NumericComparator.GE,
         Constant.create(BuiltinTypes.SINT32, 0)));
26   if (pc.simplify()) {
27     ...
28     Variable<Integer> val = Variable.create(
           BuiltinTypes.SINT32, varname);
29     pc._addDet(new SelectExpression(arrayAttr,
           indexAttr, sym_value));
30     frame.setOperandAttr(val);
31   }
32   else {
33     ... // Ignore state
34   }
35 }
```

Listing 1: Symbolic IALOAD implementation

The condition in line 1 evaluates to true on the first execution of the construction, thus adding a path condition choice to the current system state. It returns itself, so that JPF executes this instruction again. This time, the condition in line 1 evaluates to false, and a choice is made on line 8 and 15.

While exploring the paths, SPF calls a concrete solver through JConstraints to determine if the new path condition is satisfiable (lines 10, 17 and 26). If not, JPF backtracks to a previous state, and explores another path.

Constraints on the index and the length of the array are added to the path condition on lines 9, 16, 24 and 25. In case (i) and (ii) from above, an exception is created (lines 11 and 18). In case (iii), we create a new symbolic variable for the loaded element (line 28) and add a SelectExpression to the path condition (line 29). JPF then continues the execution of the program from this state.

To demonstrate this, we symbolically execute the test program presented in Listing 2, with the integer i and the array arr considered symbolic.

```
1 public int comp(int i, int[] arr) {
2   int a = arr[i];
3   return 1/a;
4 }
```

Listing 2: A simple array example

The statement in line 2 will be translated to an IALOAD instruction in Java Bytecode. As shown previously, the symbolic execution of this instruction creates three paths to explore: The index is smaller than 0, greater than the length of the array or in bounds. The original implementation would have generated two paths to check whether the index is out of bounds, and one path for each possible in bounds value of the index, assuming that the length of the symbolic array is the same as the one passed as a parameter. We add the constraints on the index on lines 9, 16, 24 and 25 in Listing 1. Since these constraints are the only ones in each of these path conditions, all of them are satisfiable. The first two paths lead to an erroneous system state thus we raise an ArrayIndexOutOfBoundsException in both cases. The last explored path leads to the execution of the next instruction, on line 3. The division creates two new paths from this state: The denominator is equal to 0 and an ArithmeticException is raised, or the denominator is different from 0 and the result of the division is returned. Since no constraint was added on the value loaded from the array *arr*, both path conditions are satisfiable. Thus, SPF returns four possible execution paths: The index is smaller than 0, greater than the array length, in bounds and the element loaded is 0, or in bounds and the element loaded is different from 0. The last execution path is the only one leading to a correct behaviour.

We execute the IALOAD instruction symbolically as soon as the index or the array is symbolic. If only the array is symbolic, we translate the concrete index to a symbolic Constant<Integer>. If only the index is symbolic, we need to create a symbolic array containing all the elements of the concrete array, as shown in Listing 3.

```
1 ElementInfo arrayInfo = ti.getElementInfo(
      arrayRef);
2 ArrayExpression<Integer> arrayAttr =
      ArrayExpression.create(BuiltinTypes.SINT32,
      arrayInfo.toString(), arrayInfo.arrayLength());
3 for (int i = 0; i < arrayInfo.arrayLength(); i++) {
4   int arrValue = arrayInfo.getIntElement(i);
5   pc._addDet(new SelectExpression(arrayAttr,
        Constant.create(BuiltinTypes.SINT32, i),
        Constant.create(BuiltinTypes.SINT32,
        arrValue)));
6 }
```

Listing 3: Creation of a symbolic array

We retrieve the information about the concrete array in line 1, and create a new ArrayExpression with a fixed symbolic length (line 2). We finally loop over all the elements in the array (line 3) to add a select constraint for each of them (line 5). Therefore, the symbolic array corresponds exactly to the concrete array given as an input.

The implementation of the IASTORE instruction is very similar to IALOAD. We still have three paths to explore depending on the value of the index. If the path condition with an index in bounds is satisfiable, we create a new ArrayExpression *newArrayAttr* with the same symbolic length than the previous ArrayExpression *arrayAttr*, and add a StoreExpression to the path condition that links *arrayAttr* to *newArrayAttr*. The bytecode instructions loading or storing numeric types from an array, e.g. SALOAD, FASTORE are all implemented using this approach. Only the type of the values and arrays will differ. In the implementation of SALOAD, we would for instance replace Expression<Integer> by Expression<Short> for the symbolic value, since we are loading a *short* instead of an *int*.

2.2 Arrays of Reference Types

We implement the AALOAD instruction slightly differently. The AALOAD instruction is used in Java Bytecode to load references to complex objects from an array. An object in the JVM is represented as an integer that references the object itself. The reference is manipulated by the JVM when an object is involved. To handle complex object during symbolic execution, SPF uses **Lazy initialization** [8]: When a symbolic object o is accessed, the lazy initialization for o takes place. The following possibilities are non-deterministically considered: o is equal to *null*; o is a previously initialized object; or o refers to a new object, with uninitialized field values.

The structure of the AALOAD instruction is the same as previously: We have three possible paths, depending on the index. If the index is out of bounds, we throw an exception. If not, we add the constraints on the index and length of the array, and load a reference to an object. To achieve this last step, we use lazy initialization, as shown in Listing 4.

```
1  ...
2  if (currentChoice < numSymRefs) {
3    ArrayHeapNode candidateNode = prevSymRefs[
         currentChoice];
4    pc._addDet(NumericBooleanExpression.create(
         indexAttr, NumericComparator.EQ,
         candidateNode.arrayIndex));
5    if (pc.simplify()) {
6      se = new SelectExpression(arrayAttr, indexAttr,
           candidateNode.getSymbolic());
7      pc._addDet(se);
8      daIndex = candidateNode.getIndex();
9      frame.pop(2);
10     frame.push(daIndex, true);
11     ...
12   } else {
13     ... // Ignore state
14   }
15 } else if (currentChoice == (numSymRefs)) {
16   if (pc.simplify()) {
17     se = new SelectExpression(arrayAttr, indexAttr,
           Constant.create(BuiltinTypes.SINT32, −1));
18     pc._addDet(se);
19     frame.pop(2);
20     frame.push(MJIEnv.NULL, true);
21     ...
22   } else {
23     ... // Ignore state
24   }
25 } else {
26   if (pc.simplify()) {
27     HelperResult hpResult =
           Helper.addNewArrayHeapNode(...);
28     daIndex = hpResult.idx;
29     ArrayHeapNode candidateNode = hpResult.n;
30     se = new SelectExpression(arrayAttr, indexAttr,
           candidateNode.getSymbolic());
31     pc._addDet(se);
32     frame.pop(2);
33     frame.push(daIndex, true);
34     ...
35   } else {
36     ... // Ignore state
37   }
38 }
```

Listing 4: Lazy initialization in AALOAD

If *numSymRefs* objects were previously initialized, we have *numSymRefs* + 2 paths to explore: one for each of the previously initialized objects (lines 2-14); one for the *null* object (lines 15-24); and one for a new object with uninitialized field values (lines 25-37). For each of these paths, we add a select constraint to the path condition to express that the object was loaded from array *arrayAttr* at index *indexAttr* (lines 7, 18 and 31). We finally push the concretized reference on the stack as a result of the AALOAD instruction (lines 10, 20 and 33). Since the reference returned is now concrete, the implementation of AASTORE is highly similar to IASTORE.

To demonstrate how arrays of reference types are handled, we execute the test program presented in Listing 5 with both arguments symbolic.

```
1  public int obj_array(int i, ObjTest[] arr) {
2    ObjTest obj = arr[i];
3    int a = obj.y;
4    return 1/a;
5  }
6
7  public class ObjTest {
8    int x;
9    int y;
10 }
```

Listing 5: A simple object array example

As in the previous example, execution paths leading to ArrayIndexOutOfBoundsException are explored (line 2, Listing 5). We focus on the path in which the index is within bounds. We use lazy initialization to load an object. Since no object was previously generated, two paths are created. We load *null* or a new object. If *obj* is *null*, the instruction on line 3 raises a NullPointerException, since we are referencing a field on a null object. If *obj* is a new object, its field *y* is a new symbolic integer. Thus, as in the previous example, the division creates two execution paths, one of them leading to a ArithmeticException.

The symbolic execution of this program thus creates 5 different paths: Two leading to an index out of bounds, one leading to referencing a field on a null object, one leading to a division by 0 because of *obj.y* being equal to 0 and the last one leading to a correct behaviour.

3. TESTCASE GENERATION

We aim at providing concrete inputs leading to the several execution paths encountered during the symbolic execution. The current version of SPF provides the SymbolicSequenceListener to generate JUnit[3] tests for each of the paths explored when the symbolic variables have numeric types. We present an extension of this listener to support arrays.

Our current work enables the generation of integer arrays matching the constraints in the path condition. JConstraints provides a Valuation of the expressions when we call a solver. To obtain a concrete solution for an expression, we can call the evaluate method with the Valuation passed to the solver instead of retrieving the solution field of the expression. We use this feature to obtain the length of a symbolic array.

We collect all the *select* constraints regarding a symbolic array. Each constraint contains a symbolic array a, a symbolic index i and a symbolic value v. We retrieve concrete values for i and v using the Valuation. We order the *select* constraints to keep only the original elements in the array: If an element is stored at index i, and a *select* constraint occurs after the store instruction, the select constraint is ignored during the construction of the array.

We finally construct an array of the given length, and populate it using the information provided by the *select* constraints. In a future work, we will study how complex data structures can be generated, and thus enhance the array generation to support reference arrays.

The JUnit tests generated for the method presented in Listing 2 are shown in Listing 6.

[3]http://junit.org

```
@Test
public void test0() {
  comp(0,new int[]{1});
}

@Test(expected = java.lang.ArithmeticException.class)
public void test1() {
  comp(0,new int[]{0});
  //leads to java.lang.ArithmeticException
}

@Test(expected = java.lang.ArrayIndexOutOfBoundsException.class)
public void test2() {
  comp(1073741824,new int[]{0});
  //leads to java.lang.ArrayIndexOutOfBoundsException
}

@Test(expected = java.lang.ArrayIndexOutOfBoundsException.class)
public void test3() {
  comp(−2147483648,new int[0]);
  //leads to java.lang.ArrayIndexOutOfBoundsException
}
```

Listing 6: JUnit Tests Generated for the code in Listing 2

4. EVALUATION

We evaluate the effects of the new handling of symbolic arrays. The original implementation allowed the execution of *ALOAD and *ASTORE instructions with symbolic indices with fixed-size arrays. Our new implementation allows the execution of symbolic arrays with a symbolic length. The results presented were obtained on a machine featuring an Intel Core i5 @2.50 GHz and 4 GB of memory.

We compare the two implementations on the method shown in Listing 7, with both arguments symbolic. We call this method with an array of size N. The results are shown in Table 1.

```
public int test1(int i, int[] arr) {
  int a = arr[i];
  int b = arr[i];
  return 1/a;
}
```

Listing 7: Test Program 1.

N	# of states		Analysis time	
	SPF_{orig}	SPF_{array}	SPF_{orig}	SPF_{array}
10	153	11	<1s	<1s
100	10.503	11	2s	<1s
1.000	1.005.003	11	89s	<1s

Table 1: Evaluation on Test Program 1.

Here column SPF_{orig} and SPF_{array} refer to the results obtained with the original implementation and the implementation that uses the new handling of arrays with symbolic length, respectively. The new implementation does not depend on the length of the array; it creates three paths for any *ALOAD and *ASTORE instruction. Thus, we observe a significant difference in speed and states explored when the length of the array grows.

The new implementation also enables the symbolic execution of the method shown in Listing 8.

```
public int test2(int[] arr) {
  int i = arr.length;
  return 1/i;
}
```

Listing 8: Test Program 2.

When given an array with a length different from 0, the original implementation cannot detect a possible division by 0 in the program presented in Listing 8. The original implementation also cannot execute the program presented previously in Listing 5.

Nevertheless, the original implementation is faster in some specific cases such as the example program presented in Listing 9 with results shown in Table 2. seems maybe too silly

```
public void test3(int j) {
  int[] arr = new int[N];
  arr[j] = 0;
  for(int i = 0; i < N; i++) {
    int temp = arr[i];
  }
}
```

Listing 9: Test Program 3.

N	# of states		Analysis time	
	SPF_{orig}	SPF_{array}	SPF_{orig}	SPF_{array}
10	13	35	<1s	<1s
50	53	155	<1s	4s
100	103	305	<1s	11s

Table 2: Evaluation on Test Program 3.

The reason is that when executing an *ASTORE instruction when the index is symbolic, the new implementation will create a new symbolic array while the original will create a path for each possible value of the index. When array instructions with concrete indices are later executed, the new implementation will use symbolic semantics. Three states will be generated testing if the index is in bounds or not. The original implementation will use concrete semantics and will not create new states to explore.

We observe similar results when symbolically executing a Dijkstra algorithm with a fixed-size symbolic array, as presented in Table 3.

N	# of states		Analysis time	
	SPF_{orig}	SPF_{array}	SPF_{orig}	SPF_{array}
2	3	34	<1s	<1s
3	17	598	<1s	8s

Table 3: Evaluation on a Dijkstra algorithm

5. CONCLUSIONS AND FUTURE WORK

We presented enhancements to SPF for handling programs that manipulate arrays. In particular, by incorporating the theory of arrays used in e.g., Z3, we can perform symbolic execution of arrays with symbolic length. The support encompasses both arrays of primitive type and reference types. The latter is made possible through a novel combination with lazy initialization. We also expanded the JUnit tests generation to arrays with numeric elements. Finally, we added an option to use JConstraints, to use constraints optimization features.

In future work we plan to investigate how complex data structures can be generated to use in JUnit tests. Furthermore, we plan to evaluate our work in the context of data structures similar to arrays, such as lists, or collections.

6. REFERENCES

[1] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley. Enhancing symbolic execution with veritesting. ICSE 2014.

[2] E. Bounimova, P. Godefroid, and D. Molnar. Billions and billions of constraints: Whitebox fuzz testing in production. ICSE '13.

[3] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser. Symbolic execution for software testing in practice: Preliminary assessment. ICSE '11.

[4] J. Christ, J. Hoenicke, and A. Nutz. SMTInterpol: An interpolating SMT solver. In *Proceedings of the 19th International Workshop on Model Checking Software (SPIN)*, pages 248–254, 2012.

[5] L. Ciortea, C. Zamfir, S. Bucur, V. Chipounov, and G. Candea. Cloud9: A software testing service. *SIGOPS Oper. Syst. Rev.*, 43(4):5–10, Jan. 2010.

[6] L. De Moura and N. Bjørner. Z3: an efficient smt solver. TACAS'08/ETAPS'08.

[7] L. de Moura and N. Bjørner. Generalized, efficient array decision procedures.

[8] S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 553–568, 2003.

[9] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[10] K. Luckow, M. Dimjašević, D. Giannakopoulou, F. Howar, M. Isberner, T. Kahsai, Z. Rakamarić, and V. Raman. JDart: A dynamic symbolic analysis framework. In M. Chechik and J.-F. Raskin, editors, *Proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2016.

[11] J. Mccarthy. Towards a mathematical science of computation. In *In IFIP Congress*, pages 21–28. North-Holland, 1962.

[12] C. S. Pasareanu, W. Visser, D. H. Bushnell, J. Geldenhuys, P. C. Mehlitz, and N. Rungta. Symbolic pathfinder: integrating symbolic execution with model checking for java bytecode analysis. *Automated Software Engineering*, 20(3):391–425, 2013.

[13] M. Souza, M. Borges, M. d'Amorim, and C. S. Păsăreanu. CORAL: Solving complex constraints for symbolic Pathfinder. In *Proceedings of the 3rd NASA Formal Methods Symposium (NFM)*, pages 359–374, 2011.