# Adding Constraint Building Mechanisms to a Symbolic Execution Engine Developed for Detecting Runtime Errors

István Kádár, Péter Hegedűs, and Rudolf Ferenc

University of Szeged, Department of Software Engineering
{ikadar,hpeter,ferenc}@inf.u-szeged.hu

**Abstract.** Most of the runtime failures of a software system can be revealed during test execution only, which has a very high cost. The symbolic execution engine developed at the Software Engineering Department of University of Szeged is able to detect runtime errors (such as null pointer dereference, bad array indexing, division by zero) in Java programs without running the program in real-life environment.
In this paper we present a constraint system building mechanism which improves the accuracy of the runtime errors found by the symbolic execution engine mentioned above. We extend the original principles of symbolic execution by tracking the dependencies of the symbolic variables and substituting them with concrete values if the built constraint system unambiguously determines their value.
The extended symbolic execution checker was tested on real-life open-source systems as well.

**Keywords:** Software engineering, symbolic execution, Java runtime errors, constraint system building

## 1 Introduction

Nowadays, producing great, reliable and robust software systems is quite a big challenge in software engineering. About 40% of the total development costs go for testing and maintenance activities, moreover, bug fixing of the system also consumes a considerable amount of resources. The symbolic execution engine developed at the Software Engineering Department of University of Szeged supports this phase of the software engineering lifecycle by detecting runtime errors (such as null pointer dereference, bad array indexing, division by zero) in Java programs without running the program in real-life environment.

According to the theory of symbolic execution [1] the program does not run with specific input data, but the inputs are handled as symbolic variables. When the execution of the program reaches a branching condition containing a symbolic variable, the execution continues on both branches. At each branching point both the affected logical expression and its negation are accumulated on the true and false branches, thus all of the execution paths will be linked to a unique formula over the symbolic variables.

The paper describes a constraint system construction mechanism, which improves the accuracy of the runtime errors found by the symbolic execution engine

mentioned above by treating the assignments in the program as conditions too. Thus we can track the dependencies of the symbolic variables extending the original principles of symbolic execution. The presented method also substitutes the symbolic variables with concrete values if the built constraint system unambiguously determines their value. To build and satisfy the constraint systems we used the open-source Gecode constraint satisfaction tool-set [2].

The paper explains in detail how the algorithm is implemented that builds the constraint system for each execution path, how it is integrated into the symbolic execution engine of the Department of Software Engineering, and how the algorithm enhanced the effectiveness of the engine. The extended symbolic execution checker was tested on real-life open-source systems as well and we compared it with our previous tool [3] based on SymbolicPathFinder.

## 2 Background

### 2.1 Symbolic Execution

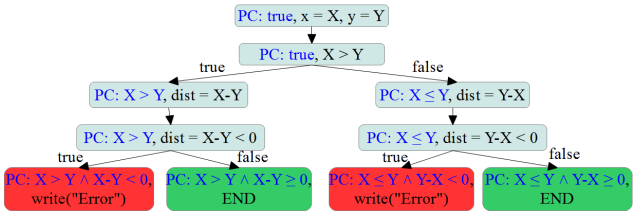
During its execution, every program performs operations on the input data in a defined order. Symbolic execution [1] is based on the idea that the program is operated on symbolic variables instead of specific input data, and the output will be a function of these symbolic variables. A symbolic variable is a set of the possible values of a concrete variable in the program, thus a symbolic state is a set of concrete states. When the execution reaches a selection control structure (e.g. an if statement) where the logical expression contains a symbolic variable, it cannot be evaluated, its value might be also true and false. The execution continues on both branches accordingly. This way we can simulate all the possible execution branches of the program.

During symbolic execution we maintain a so-called *path condition (PC)*. The path condition is a quantifier-free logical formula with the initial value of true, and its variables are the symbolic variables of the program. If the execution reaches a branching condition that depends on one or more symbolic variables, the condition will be appended to the current PC with the logical operator *AND* to indicate the true branch, and the negation of the condition to indicate the false branch. With such an extension of the PC, each execution branch will be linked to a unique formula over the symbolic variables. In addition to maintaining the path condition, symbolic execution engines make use of so called *constraint solver* programs. Constraint solvers are used to solve the path condition by assigning values to the symbolic variables that satisfy the logical formula. Path condition can be solved at any point of the symbolic execution. Practically, the solutions serve as test inputs that can be used to run the program in such a way that the concrete execution follows the execution path for which the PC was solved.

SymbolicChecker, the symbolic execution engine developed at the Software Engineering Department does not aim to generate test inputs, but to find as many true positive runtime errors in the program as possible. In accordance with this goal we changed and extended the standard path condition building method described above.

```
1. int x, y, dist;
2. ...
3. if (x > y) {
4.   dist = x - y;
5. } else {
6.   dist = y - x;
7. }
8. if (dist < 0)
9.   write("Error");
```

(a)




(b)

Fig. 1: (a) Sample code that determines the distance of two integers on the number line (b) Symbolic execution tree of the sample code handling variable x and y symbolically

Figure 1 (a) shows a sample code that determines the distance of two integers x and y. The symbolic execution of this code is illustrated on Figure 1 (b) with the corresponding symbolic execution tree. We handle x and y symbolically, their symbols are X and Y respectively. The initial value of the path condition is true. Reaching the first if statement in line 3, there are two possibilities: the logical expression can be true or false; thus the execution branches and the logical expression and its negation is added to the PC as follows:

$$true \wedge X > Y \Rightarrow X > Y, \quad \text{and} \quad true \wedge \neg(X > Y) \Rightarrow X \leq Y.$$

The value of variable *dist* will be a symbolic expression, X-Y on the true branch and Y-X on the false one. As a result of the second if statement (line 8) the execution branches, and the appropriate PCs are appended again. On the true branches we get the following PCs:

$$X > Y \wedge X - Y < 0 \Rightarrow X > Y \wedge X < Y,$$

$$X \leq Y \wedge Y - X < 0 \Rightarrow X \leq Y \wedge X > Y.$$

It is clear that these formulas are unsolvable, we cannot specify such X and Y that satisfy the conditions. This means that there are no such x and y inputs with which the program reaches the *write("Error")* statement. As long as the PC is unsatisfiable at a state, the sub-tree starting from that state can be pruned, there is no sense to continue the controversial execution.

### 2.2 SymbolicChecker, the Symbolic Execution Engine

The goal of SymbolicChecker is to detect those real runtime errors that other audit tools cannot detect and those which mostly can be discovered by a large amount of testing only. It is important for us that the detected errors be as accurate as possible, so we can eliminate the false positive hits and find more numerous true positives that helps the software developers to create a higher-quality product and makes the maintenance tasks easier. Generating test cases which lead to the errors is not a goal here, much more to produce a descriptive designation of the execution path that led to a fault.

In the present paper we do not give a detailed description of the Symbolic-Checker analysis tool, but in order to understand our new constraint building concept, its basic understanding is needed. SymbolicChecker is written in C++,

the development is still in progress. Currently the detection of null pointer dereferences, array over-indexing, division by zero, and type cast errors are implemented.

SymbolicChecker performs the analysis by symbolically executing each method of the system one-by-one. The parameters of the method under execution and the referred but not initialized variables are handled as symbols at the beginning of the analysis. It is important that we only report an error if it is guaranteed that during the execution the value that causes the problem can be determined by constant propagation. For example, if a method call is guaranteed to pass a null value, and it is guaranteed that the called method dereferences this parameter, we will fire an error, but if the dereferenced variable is a symbol we will not, because its value is unknown or uncertain. To limit the size of the symbolic execution tree, its maximum depth and the maximum number of states can be specified.

The symbolic execution is performed using the language-dependent abstract semantic graph (ASG) [4] of the program by interpreting the the nodes of this graph in a defined order. The order is defined by the language-independent control flow graph (CFG) [5]. The output of the SymbolicChecker contains the detected errors indicating their type, the execution path from the entry point to the exact location where the error occurred and a probability that estimates how likely the analyzed method runs onto the detected fault.

In SymbolicChecker *Definition* is a comprehensive name for all the data that appears during the symbolic execution. For example, the concrete or symbolic variables, constants, parameters of methods, their return value, or the result of sub-expressions are also Definitions. Actually, the symbolic execution of the program is the propagation of these Definition objects. Basically there are two types of Definitions: ValueDefinition and SymbolDefinition. ValueDefinition objects store specific, concrete values and SymbolDefinition instances represent the symbolic variables.

### 2.3 JPF Checker

In one of our previous works [3] we used SymbolicPathFinder [6] to create a tool named JPF Checker with the same goal as with SymbolicChecker: to detect runtime errors in Java programs without modifying the source code and without having to run it in a real-life environment. This SymbolicPathFinder based tool used the conventional constraint building mechanism. In Section 4 we compare this approach to our new concept which is implemented in SymbolicChecker.

## 3 Constraint Building

### 3.1 Principles

In this work we developed a constraint building mechanism and integrated it into SymbolicChecker which allows us to detect runtime errors that a conventional symbolic execution system cannot.

As we described in Section 2.2, SymbolicChecker reports errors only if the value causing the problem becomes concrete. The tool does not fire for symbolic

variables because if a variable is a symbol it actually means that it's value is doubtful, not known. It may occur that during the symbolic execution of a program most of the variables turn into symbols, which makes finding runtime errors rather difficult.

The main idea behind the developed constraint building mechanism is that if during the analysis the program sets up conditions (constraints) that unambiguously determines the value of one or more symbolic variables, then we can convert these symbols into concrete values and the symbolic execution can be continued on the actual path using the concreted variables. Since these variables handled like concrete data, it is possible to detect errors that otherwise SymbolicChecker could not find. The conditions we mentioned above are determined by the conditional control structures (if, switch, while, etc.) and expressed by the assignments of the program, including the impacts of the increment and decrement operators $(++, --)$ of the Java language.

Overall, the goal of the implemented constraint building mechanism is the concretion of as many symbols as possible, which helps to find more runtime errors. In order to achieve this (1) it is necessary to build a special path condition (PC), that contains the dependencies of the symbolic variables too determined by the assignments of the program, and (2) if the constraints in the PC determine the values of some symbols unambiguously, the execution has to be continued using these concrete values on the actual path. Since such an extended path condition includes – in some form – also those conditions that are in the PC built the conventional way, as long as it is infeasible, the code parts that are unreachable can also be skipped. Therefore, false positive defects can be eliminated.

To demonstrate the basic idea of extending the PC, consider the code snippet in Figure 1a. In this example, the conventional path condition of the path which passes through the true branch of the if statement in line 3, and the false branch of if in line 8 is the following:

$$X > Y \ \wedge \ \neg(X - Y < 0) \ \Rightarrow \ X > Y \ \wedge \ X - Y \geq 0.$$

According to our concept the extended PC of the same path is the following:

$$X > Y \ \wedge \ \neg(dist < 0) \ \wedge \ dist = X - Y.$$

It can be seen that variable *dist* is also included in the constraint system as a symbol, on the one hand in the negation of condition in line 8 where X-Y was not substituted, on the other hand the constraint that expresses the assignment in line 4. As a result, the constraint system contains information about variable dist as well that could be useful in the later stages of the execution.

In this example, the extended PC does not contain constraints which could unambiguously determine any variable, thus the benefit of the extension is not obvious here. The code snippet in Figure 2a shows an example where the extended PC indeed has some gains.

Executing the code symbolically in Figure 2a handling variable c as a symbol, the following constraint system will be built in the program state at line 7:

$$a > 8 \ \wedge \ a = b + 9 \ \wedge \ b = 2 \cdot c + 4 \ \wedge \ a < 10.$$

The constraint system above includes constraints that are introduced by the if statements of the code and the dependencies of symbol $a$, i.e. those constraints

```
1. // c is an int symbol
2. double b = 2*c + 4;
3. int a = b + 9;
4. if (a > 8) {
5.   ...
6.   if (a < 10) {
7.      // concretion of b
8.      int p = 1/b;
9.   }
10. }
```

(a) Sample code that provides symbol concretion, which helps to find runtime errors that a conventional symbolic execution tool cannot.

```
1. // a and b are symbols
2. if (b > 0) {
3.   ...
4.   if (a == 0) {
5.     // concreting a?
6.     ...
7.   }
8. }
```

(b) Code snippet which points out a path condition that has more solutions, but concreted symbol a

Fig. 2

that are given by the assignments that defines variable $a$. After satisfying this constraint system it can be obtained that $a$ can only be 9, which implies that the value of b and c symbols are unambiguous too: $b = 0.0$ and $c = -2$. In such a situation the execution continues on that path for which the extended PC was satisfied. In case of the considered example if the execution continues with the b=0.0 value, at line 8 a division by zero error can be detected. As long as symbol $b$ would not be included in the PC, and if its unambiguous value would not be used, the detection of division by zero would fail.

In real-life programs, quite big constraint systems are be built, which contain lots of symbols. Easy to see that satisfying such a large set of constraints as a whole, it has a low probability that there is only one possible solution.

Figure 2 shows a code snippet that highlights the problem in question. Considering the path that passes along the true branches of both if statements, the path condition is $b > 0 \wedge a = 0$. Although there are infinite number of solutions of this formula, because the $b > 0$ constraint can be satisfied by any positive integer, the formula determines the value of symbol $a$ unambiguously, which would be preferred to use in later stages of the execution.

To overcome the problem we decompose the path condition into connected components that is, to constraint sets that are independent i.e. does not contain the same variables. The connected components can be satisfied individually and if some of them determines a variable unambiguously then the obtained values can be used later in the execution. Two constraints are in the same component if they contain at least one common variable. After such a decomposition the path condition becomes a set of constraint sets.

The essential steps of the algorithm of our constraint system building is shown in Figure 3. This algorithm will be executed after each branching point in the symbolic execution tree.

First of all, it is determined if the accumulation of the PC happens on the true or on the false branch then dependent upon this the created logical expression or its negation is stored in variable *constraint* (lines 3-7) (the handling of switch statement of the Java programming language is not shown in the pseudo code). It is important to note that we build constraint exactly from that logical expression that is determined in the source code, there is no substitution

```
 1. Constraint constraint;
 2. set<Constraint> actualConstraints;
 3. if (onTrueBranch()) {
 4.   constraint = constraintBuilder.createConstraint();
 5. } else if (onFalseBranch()) {
 6.   constraint = constraintBuilder.createNegatedConstraint();
 7. }
 8. actualConstraints.insert(constraint);
 9. actualConstraints.union(dependenciesOfSymbolsInConstraint);
10. pathCondition.insert(actualConstraints);
11. decomposedPC = decompose(pathCondition);
12. foreach (set<Constraint> s : decomposedPC) {
13.   constraintSolver.solve(s);
14.   if (s.hasSolution) {
15.     if (!s.hasMoreSolution) {
16.       buildBackSolutions(s);
17.     }
18.   } else {
19.     weight = 0.0;
20.     break;
21.   }
22. }
```

Fig. 3: Pseudo code of the algorithm of constraint system building

of variables like in case of variable *dist* in Section 2.1, in example 1. Next, the created constraint is added to the *actualConstraints* constraint set (line 8), and also the dependencies of the symbols included in this constraint are inserted (line 9). These dependencies are defined by the assignments of the code, later we will discuss how they are created. The next step is that the path condition of the current execution path is extended by the *actualConstraints* constraint set (line 10), then the PC will be decomposed into connected components in line 11. As long as one of the connected components cannot be satisfied the weight of the current path is set to 0.0 indicating that there is no sense to continue the execution because of the contradictory conditions (line 19). On the other hand, if there is at least one solution, the algorithm examines its uniqueness (line 13), if the solution is unique, the concrete values are built back into the current symbolic state (line 16).

It have to be emphasized that a concreted symbol is built back into the a state only once and only into that state for which the constraint system concreted it.

### 3.2 Implementation

We used the Gecode constraint solver tool-set [2] for building and satisfying our constraint systems. Basically, we can differentiate two kinds of constraints: (1) conditions in the conditional control structures of the program (including the loops too) and (2) the dependencies of symbols which are included in these conditions. In the following, we describe how did we implement the building of the constraint system integrated into SymbolicChecker.

As we described in Section 2.2, for every kind of data that appears in a program during the symbolic execution (e.g. variables, literals, sub-expressions, etc.) a *Definition* object is created. In fact, the execution of the program is nothing else than the proper propagation of these Definition objects. The task is to achieve the tracking that determines what other Definitions a Definition object is created from and what operations it uses. This is how the relations between symbolic variables are described.

For the implementation we added a so-called *constraintSolverExpression* data member to the class Definition and a dependency set too, which is a set of constraints. These attributes are propagated with the Definitions along the program by the symbolic execution.

The constraintSolverExpression represents an expression object created using the Gecode constraint solver. With such an expression object Gecode can represent the inner structure of expressions that is, which operands are they created from using which operators. The constraintSolverExpressions is propagated in the following way: when an operation is performed on Definition objects we take the constraintSolverExpressions of the operands and perform the operation on the expressions too, and the resulting compound constraintSolverExpression will be set in the resulting Definition object. In case of operations performed on ValueDefinitions for efficiency reasons the operation on the constraintSolverExpressions is not performed, instead we simply create a new Gecode expression which stores the calculated value.

The dependency set contains the dependencies of those symbols which are in the constraintSolverExpression which are defined by the assignments of the program. In case of those assignments where the right side is a SymbolDefinition, for the variable which is on the left we create a new symbol. This new symbol will not take over the constraintSolverExpression of the right side, but the relation between left and right side Definitions is expressed by an equality constraint between them. The dependency set is propagated in the following manner. After performing an operation the dependency set of the resulting Definition will be the union of the dependency sets of the operands. In case of an assignment which has SymbolDefinition on the right side, the dependency set of the newly created SymbolDefinition on the left will be the dependency set of the right side symbol extended by the constraint which defines equality between the two sides.

```
1. // d is a symbol
2. int b = d + 3;
3. int c = 2*d;
4. int a = b { c;
5. if (42 == a) {
6.   ...
7. }
```

Fig. 4: Sample code for demonstrating the propagation of dependency sets.

The branching conditions in selection control structures which defines the branching points of the symbolic execution tree are also expressions in the program, thus they appear as Definition objects (actually as SymbolDefinitions) in SymbolicChecker. Variable *constraint* in the algorithm shown in Figure 3 is created from the constraitnSolverExpression of such a Definition object, and constraint set *dependenciesOfSymbolsInConstraint* is the dependency set of this Definition too. Constraint set *actualConstraints* by which the path condition will be extended is the union of the above mentioned *constraint* and *dependenciesOfSymbolsInConstraint*.

In the followings, we demonstrate the building of the constraint system and the propagation of dependency sets for the example code in Figure 4. Variable $d$ is handled as symbol, it is a SymbolDefinition which dependency set is empty

and its constraintSolverExpression is a Gecode expression which contains only a simple unknown variable. Firstly, in line 2 a ValueDefinition is created for literal 3, which dependency set is empty, then operation + creates the *d+3* SymbolDefinition. The dependency set of *d+3* is the dependency set of the left and the right side, which is also an empty set:

$$SymbolDef(d{+}3).depset \;=\; SymbolDef(d).depset \cup ValueDef(3).depset \;=\; \emptyset.$$

After the execution of the assignment the dependency set of $b$ is:

$$SymbolDef(b).depset \;=\; SymbolDef(d{+}3).depset \cup \{b = d + 3\} \;=\; \{b = d + 3\}\,.$$

Dependency set of symbol $c$ created at line 3 is quite similar:

$$SymbolDef(c).depset \;=\; SymbolDef(2 * d).depset \cup \{c = 2 \cdot d\} \;=\; \{c = 2 \cdot d\}\,.$$

At the left hand side of assignment at line 4, dependency set of SymbolDefinition $b$-$c$ is the union of dependency set of $b$ and $c$:

$$\begin{aligned} SymbolDef(b - c).depset \;&=\; SymbolDef(c).depset \;\cup\; SymboldDef(b).depset \\ &=\; \{b = d + 3,\; c = 2 \cdot d\}\,. \end{aligned}$$

Then the dependency set of $a$:

$$\begin{aligned} SymbolDef(a).depset \;&=\; SymbolDef(b - c).depset \;\cup\; \{a = b - c\} \\ &=\; \{b = d + 3,\; c = 2 \cdot d,\; a = b - c\}. \end{aligned}$$

Dependency set of SymbolDefinition created from expression $42 == a$ at line 5 is the same as the dependency set of $a$, thus the path condition on the true branch of if statement is the following:

$$\begin{aligned} PC \;&=\; \{42 = a\} \;\cup\; \{b = d + 3,\; c = 2 \cdot d,\; a = b - c\} \\ &=\; \{42 = a,\; b = d + 3,\; c = 2 \cdot d,\; a = b - c\}. \end{aligned}$$

## 4 Evaluation

SymbolicChecker with our constraint building mechanism was tested in a variety of ways. This section contains the results of these tests. First of all, we demonstrate the advantages of our algorithm through two examples emphasizing the difference of JPF Checker and SymbolicChecker without using the constraint building mechanism. After that, we write about the experiences got form the tests we have performed on large, real-life systems.

In *run()* method of the example code shown in Figure 5 SymbolicChecker with constraint building detects an array over-indexing fault. First of all, we follow what is the cause of the runtime error, then we look at how the new approach helps detecting it. Line 5 defines an array called *arr* with size of *max*. As long as parameter $n$ is grater than 0 (line 6), a sequence of operations will be performed which aims to calculate two sums based on the content of the array. This sequence of operations fills the array at first (lines 7-9), then starting from $n$, summarizes the *member* data members of objects on every second index into the variable *sum1* (lines 11-16). Next, the code calls method *gcd()* with arguments $n$ and the 0th element of array *arr* (line 18) and summarizes the elements of the array starting from the negation of the return value of *gcd()* (lines 20-22). Method *gcd()* calculates the greatest common divisor of the numbers and

```
1. class Example {
2.
3.   public void run(int n) {
4.     int max = getCharPos('w');
5.     A[] arr = new A[max];
6.     if (n > 0) {
7.       for (int i = 0; i < max; ++i) {
8.         arr[i] = new A(max - i);
9.       }
10.      int sum1 = 0;
11.      while (n < max) {
12.        if (n % 2 == 0) {
13.          sum1 += arr[n].getMember();
14.        }
15.        n++;
16.      }
17.      System.out.println("Sum1: " +  sum1);
18.      int negOfGcd = -gcd(n, arr[0]);
19.      int sum2 = 0;
20.      while (negOfGcd < max) {
21.        sum2 += arr[negOfGcd++].getMember();
22.      }
23.      System.out.println("Sum2: " +  sum2);
24.    }
25.  }
26.
27.   public int getCharPos(char c) {
28.     return c { 'a' + 1;
29.   }
30.
31.   private int gcd(int x, int y) {
32.     while (y != 0) {
33.       int m = x % y;
34.       x = y;
35.       y = m;
36.     }
37.     return x;
38.   }
39.
40. }
41.
42. class A {
43.   private int member;
44.
45.   public A(int member) {
46.     this.member = member;
47.   }
48.
49.   public int getMember() {
50.     return member;
51.   }
52. }
```

Fig. 5: Example code with the analysis of method run().

its return value must be a positive integer if the arguments are $n$ and *arr[0]*. Because of this, variable negOfGcd guaranteed to be negative which causes an ArrayIndexOutOfBoundsException runtime error that results in the halt of the program.

When starting the analysis with method run(), variable $n$ is the only symbol. Variable *max* is concrete, array arr is also instantiated concretely and all of its elements are concrete values too. However, the execution of the loop in line 11 depends on $n$. On the false branch the execution continues from line 16, on the true branch we enter into the body of the loop, and after executing it we will branch again depending on the condition at line 11. This operation will continue until it reaches the maximum depth of the symbolic execution tree. If the execution paths entered into the loop at least once and then exited, the following constraints must be part of the extended path condition:

$$n_{prev} < max \ \wedge \ n = n_{prev} + 1 \ \wedge \ \neg(n < max) \Rightarrow$$

$$n_{prev} < max \ \wedge \ n = n_{prev} + 1 \ \wedge \ n >= max.$$

In this formula $n_{prev}$ means the instance of symbol $n$ when the execution just entered the loop. At this state the $n_{prev} < max$ constraint can be defined. In line 15 as the result of the incrementation a new symbol is created and the $n = n_{prev} + 1$ constraint is built. Since in the next iteration the execution do not enters into the loop, but continues on the false branch it is necessary to create the $\neg(n < max)$ constraint too. After satisfying the constraint set above, symbol $n$ will be determined unambiguously, and its value is equal to the value of variable *max*. This means that if the execution exits the while loop, the value of $n$ must be *max*.

Building back the unambiguous value of $n$ into the current symbolic state, the arguments of method call *gcd()* are both concrete values, thus it will be executed concretely and its return value will also be a concrete number. As we assumed the return value must be a positive integer, this leads to a bad array indexing in line 21.

The example detailed above highlights that a concreted symbolic variable can make a significant part of the execution concrete. The spread of symbols can be reduced, thus fewer variables have to be handled as unknown and uncertain data. As a result, the analysis becomes faster, because fewer execution paths have to be examined. In the shown example without concreting variable $n$ we should have explored the whole symbolic execution tree of method gcd(), which is rather expensive because of the loop inside.

The demonstrated ArrayIndexOutOfBoundException cannot be detected nor by the JPF Checker, nor by SymbolicChecker without constraint building.

In the second example we show a real code part from the log4j logging system. Consider method *org.apache.log4j.net.SMTPAppender.sendBuffer()* in Figure 6 from log4j version 1.2.11, in which we point out that our new approach can also eliminate false positive faults as the conventional path condition construction.

```
// SMTPAppender.java
public class SMTPAppender extends
                           AppenderSkeleton {
  ...
  protected Layout layout;
  protected CyclicBuffer cb =
                new CyclicBuffer(bufferSize);
  ...
  protected
  void sendBuffer() {
    ...
224.     int len =  cb.length();
225.     for(int i = 0; i < len; i++) {
226.
227.       LoggingEvent event = cb.get();
228.       sbuf.append(layout.format(event));
229.       if(layout.ignoresThrowable()) {
230.         String[] s =
                      event.getThrowableStrRep();
231.         if (s != null) {
232.           for(int j = 0; j < s.length; j++) {
233.             sbuf.append(s[j]);
234.           }
235.         }
236.       }
237.     }
    ...
  }
  ...
}
```

```
public class CyclicBuffer {
  int numElems;
  ...
101.   public
102.   LoggingEvent get() {
103.    LoggingEvent r = null;
104.    if(numElems > 0) {
105.    numElems--;
106.    r = ea[first];
107.    ea[first] = null;
108.    if(++first == maxSize)
109.    first = 0;
110.    }
111.    return r;
112.   }

  ...
119.   public
120.   int length() {
121.     return numElems;
122.   }
  ...
}

public class SimpleLayout extends Layout {
  ...
56.   public
57.   String format(LoggingEvent event) {
58.
59.     sbuf.setLength(0);
60.     sbuf.append(event
                        .getLevel().toString());
61.     sbuf.append(" - ");
62.     sbuf.append(event
                         .getRenderedMessage());
63.     sbuf.append(LINE_SEP);
64.     return sbuf.toString();
65.   }
  ...
}
```

Fig. 6: Method org.apache.log4j.net.SMTPAppender.sendBuffer() and its environment

In line 228 of method *sendBuffer()*, *get()* method of class *CyclicBuffer* is called, which returns a *LoggingEvent* reference. First of all, method get() initializes the reference $r$ to *null* (line 103), then if the *numElems* data member is greater than 0, $r$ gets a new value. However, on the false branch it returns the null-initialized $r$ reference. Following this false branch, in method *SMTPAppender.sendBuffer()* variable event is initialized to null in line 227, this null value will be propagated into method *SimpleLayout.format()*, which dereferences it in line 60.

However this null dereference would be a false positive error, because in line 60 the null value never occurs. In line 224 we get the *numElems* member of object *cb* for which the first iteration of the for loop at line 225 defines a constraint. The PC looks like this:

$$0 < len \ \wedge \ len = cb.numElems.$$

Nevertheless, method *get()* called at line 227 returns null only on the false branch where the $numElems > 0$ constraint is not satisfied, thus the path condition is the following:

$$0 < len \ \wedge \ len = cb.numElems \ \wedge \ \neg(numElems > 0).$$

This formula, however, unsatisfiable, which means that the execution can not continue on this path. Variable *event* will not get the null value in line 227, so the method *format()* of class *SimpleLayout* cannot dereference it. This actually means that the execution enters the for loop in line 225 only if the value of variable *len* is at least 1, but in this case method *get()* cannot return null on the false branch of the if statement in line 104.

The elimination of the discussed false positive error would fail using SymbolicChecker without the constraint building mechanism, but the JPF Checker would also eliminate it, because in this case no symbols are concreted and the unsatisfiability of the path condition is also tested by the JPF/SPF based tool.

We have run SymbolicChecker with the presented constraint building mechanism on large Java systems too, however the evaluation of the results is not entirely finished yet. Manually reviewing the reported errors is rather time-consuming because of the difficulty of interpreting the long execution paths from the entry point to the point where the error was detected in the source code. Which can be seen in the results so far is that there are significantly fewer runtime errors in the resultant report obtained by SymbolicChecker that uses the constraint building mechanism compared to the ones that JPF Checker produces. This does not mean that the report of SymbolicChecker does not contain false positive results, but most of them draw attention to real errors and potential sources of errors.

Considering the duration of the analyzes, the run-time of SymbolicChecker using the constraint building stays below the run-time of JPF Checker, but this duration is about twice longer then the run-time of SymbolicChecker running it without the constraint building mechanism. The analysis of the log4j logging library took slightly less than half an hour without constraint building, and the

duration is about an hour using the new approach. Of course, we expected such a time requirement of our constraint building algorithm because the building of the constraint system, decomposing it to connected components and especially its satisfaction are rather computation intensive tasks.

## 5 Related Work

In this section we present works that are related to our research. First, we introduce some existing tools and technicques for runtime error detection mainly in Java programs, then we show the possible applications of the symbolic execution. We also summarize the problems that have been solved successfully by SymbolicPathFinder that we used for implementing our approach. Finally, we present works that completed or modified the symbolic execution technique.

The work of Weimer and Necula [7] focuses on proving safe exception handling in safety critical systems. They generate test cases that lead to an exception by violating one of the rules of the language. Unlike they do not generate test inputs based on symbolic execution but solving a global optimization problem on the control flow graph (CFG) of the program.

The JCrasher tool [8] by Csallner and Smaragdakis takes a set of Java classes as input. After checking the class types it creates a Java program which instantiates the given classes and calls each of their public methods with random parameters. This algorithm might detect failures that cause the termination of the system such as runtime exceptions. The tool is capable of generating JUnit test cases and can be integrated to the Eclipse IDE. JCrasher creates a driver environment but it can analyze public methods only and instead of symbolic execution it generates random data which is obviously not feasible for examining all possible execution branches.

The DART [9] (Directed Automata Random Testing) by Godefroid et al. tries to eliminate the shortcomings of the symbolic execution e.g. when it is unable to handle a condition due to its unlinear nature. DART executes the program with random or predefined input data and records the constraints defined by the conditions on the input variables when it reaches a conditional statement. In the next iteration taking into account the recorded constraints it runs the program with input data that causes a different execution branch of the program. The goal is to execute all the reachable branches of the program by generating appropriate input data.

The idea of symbolic execution is not new, the first publications and execution engines appeared in the 1970's. One of the earliest work is by King that lays down the fundamentals of symbolic execution [1] and presents the EFFIGY system that is able to execute PL/I programs symbolically. Even though EFFIGY handles only integers symbolically, it is an interactive system with which the user is able to examine the process of symbolic execution by placing breakpoints and saving and restoring states.

Starting from the last decade the interest about the technique is constantly growing, numerous programs have been developed that aim at dynamic test

input generation using symbolic execution. The EXE (EXecution generated Executions) [10] presented by Cadar et al. at the Stanford University is an error checking tool made for generating input data on which the program terminates with failure. The input generation is done by the STP built-in constraint solver that solves the path condition of the path causing the failure. The basic difference between Symbolic Checker and EXE is that for running EXE one needs to declare the variables to be handled symbolically.

Further description and comparison of the above mentioned and other tools can be found in the work of Coward [11] and Cadar[12].

Song et al. applied the symbolic execution to the verification of networking protocol implementations [13]. The SymNV tool creates network packages with which a high coverage can be achieved in the source code of the daemon, therefore potential rule violations can be revealed according to the protocol specifications.

The main application of the Java PathFinder [14] and its symbolic execution extension, the SymbolicPathFinder [6] is the verification of the internal projects in NASA. Bushnell et al. describes the application of Symbolic PathFinder in TSAFE (Tactical Separation Assisted Flight Environment) [15] that verifies the software components of an air control and collision detection system. The primary target is to generate useful test cases for TSAFE that simulates different wind conditions, radar images, flight schedules, etc.

In our previous work [3] we used Symbolic PathFinder to create a tool named JPF Checker with the same goal as we have in case of Symbolic Checker: to detect runtime errors in Java programs without modifying the source code and without running it in a real-lif environment. This Symbolic PathFinder based tool used the conventional constraint building mechanism. In section 4 we compare this approach to our new concept which implemented in Symbolic Checker.

System MIX [16] combines symbolic execution with static type checking based techniques. It designates type and symbolic blocks in the program, which determines which code-part should be analysed using symbolic execution and wich one using static type checking. In the border of these blocks so-called mix-rules are used to convey the neccessary information. MIX is intended to provide a compromise between the precise but resource intensive symbolic execution and the less precise but faster type checking.

Shannon and others [17] built an abstraction layer above the Java string handling using finite state automatas. In addition to the implementation of the java.lang.String class StringBuilder and StringBuffer classes are included. As a result, the system is able to handle constraints that contains strings and string operations, thus it can be applied to programs that are working on more complex strings, such as SQL queries. Currently Symbolic Checker does not handle string constraints, we plan to deliver this development in the future.

Durring symbolic execution it may occur that the built path condition contains function calls, e. g. $if(y >= f(x))$. The so-called concolic (concrete-symbolic) [18] execution provides a possible solution for this problem using a special constraint builden mechanism. The main idea of this approach is that two path conditions are maintained at the same time. One of them contains

those conditions witch do not includes function calls, and the other is the so-called complex PC, in which there are conditions that includes function calls too. First, the algorithm satisfies the simple PC and assnigs values to the included symbols, then these values are used to execute those included functions concretely which execution depended on those symbols which values have been determined int the first step. This method also capitalize on turning symbols int to concrete values, like the approach we present in this paper.

## 6 Summary and Future Work

The basic principles of symbolic execution has been known for decades, and several tools were made that utilizes the possibilities offered by this technique. SymbolicChecker which we developed at the Software Engineering Department of University of Szeged is differing from the most of these tools because it does not aim to generate test inputs, but to detect execution paths that lead to runtime errors and dangerous code parts as accurately as possible. In order to reach this goal we developed a constraint building mechanism and integrated it into SymbolicChecker, which differs from the ones which are used in other systems. The presented approach builds a constraint system for each execution path, which includes constraints over the variables too that depends on the inputs handled as symbolic variables and in case of unambiguity the concreted values are used in the later stages of the analysis. As a result, runtime errors can be detected that would not be possible using a conventional symbolic execution tool. For example the demonstrated ArrayIndexOutOfBoundsException in Section 4 cannot be detected nor by the JPF Checker, nor by SymbolicChecker without constraint building. By concreting symbolic variables the size of the symbolic execution tree can be reduced as well, which implies improvements in performance also. The ability to eliminate false positive results is achieved by ignoring those paths that carries contradictory constraints.

The results so far are promising and we continue the development of our tool. First, the review and evaluation of the results we get on large systems will take place, which will determine the future tasks. We plan to optimize the whole symbolic execution engine included the constraint building mechanism, as well as develop new methods and techniques that make the detection of runtime errors even more accurate.

## Acknowledgment

The publication is partially supported by the European Union FP7 project "REPARA – Reengineering and Enabling Performance And poweR of Applications", project number: 609666.

## References

1. King, J.C.: Symbolic Execution and Program Testing. Communications of the ACM **19**(7) (July 1976) 385–394

2. Gecode Tool-set. http://http://www.gecode.org/
3. Kádár, I., Hegedűs, P., Ferenc, R.: Runtime exception detection in java programs using symbolic execution. Acta Cybernetica **21**(3) (2014) 331–352
4. Ferenc, R., Beszédes, Á., Tarkiainen, M., Gyimóthy, T.: Columbus – Reverse Engineering Tool and Schema for C++. In: Proceedings of the 18th International Conference on Software Maintenance (ICSM'02), IEEE Computer Society, IEEE Computer Society (oct 2002) 172–181
5. Allen, F.E.: Control flow analysis. SIGPLAN Not. **5**(7) (July 1970) 1–19
6. Păsăreanu, C.S., Rungta, N.: Symbolic PathFinder: Symbolic Execution of Java Bytecode. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering. ASE '10, New York, NY, USA, ACM (2010) 179–180
7. Weimer, W., Necula, G.C.: Finding and Preventing Run-time Error Handling Mistakes. In: Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications. OOPSLA '04, New York, NY, USA, ACM (2004) 419–431
8. Csallner, C., Smaragdakis, Y.: JCrasher: an Automatic Robustness Tester for Java. Software Practice and Experience **34**(11) (September 2004) 1025–1050
9. Godefroid, P., Klarlund, N., Sen, K.: DART: Directed Automated Random Testing. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '05, New York, NY, USA, ACM (2005) 213–223
10. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: EXE: Automatically Generating Inputs of Death. In: Proceedings of the 13th ACM Conference on Computer and Communications Security. CCS '06, New York, NY, USA, ACM (2006) 322–335
11. Coward, P.D.: Symbolic Execution Systems – a Review. Software Engineering Journal **3**(6) (November 1988) 229–239
12. Cadar, C., Godefroid, P., Khurshid, S., Păsăreanu, C.S., Sen, K., Tillmann, N., Visser, W.: Symbolic Execution for Software Testing in Practice: Preliminary Assessment. In: Proceedings of the 33rd International Conference on Software Engineering. ICSE '11, New York, NY, USA, ACM (2011) 1066–1071
13. Song, J., Ma, T., Cadar, C., Pietzuch, P.: Rule-Based Verification of Network Protocol Implementations Using Symbolic Execution. In: Proceedings of the 20th IEEE International Conference on Computer Communications and Networks (ICCCN'11). (2011) 1–8
14. Java PathFinder Tool-set. http://babelfish.arc.nasa.gov/trac/jpf
15. Bushnell, D., Giannakopoulou, D., Mehlitz, P., Paielli, R., Păsăreanu, C.S.: Verification and Validation of Air Traffic Systems: Tactical Separation Assurance. In: Aerospace Conference, 2009 IEEE. (2009) 1–10
16. Khoo, Y.P., Chang, B.Y.E., Foster, J.S.: Mixing type checking and symbolic execution. In Zorn, B.G., Aiken, A., eds.: PLDI, ACM (2010) 436–447
17. Shannon, D., Zhan, D., Hajra, S., Lee, A., Khurshid, S.: Abstracting symbolic execution with string analysis testing. In: Academic and Industrial Conference Practice and Research TechniquesMUTATION, 2007. TAICPART-MUTATION. (2007)
18. Păsăreanu, C.S., Rungta, N., Visser, W.: Symbolic execution with mixed concrete-symbolic solving. In: Proceedings of the 2011 International Symposium on Software Testing and Analysis. ISSTA '11, New York, NY, USA, ACM (2011) 34–44