Evaluation of Java PathFinder Symbolic Execution Extension

Kari Kähkönen

June 26, 2007

Abstract

Executions of a system can follow different execution paths depending on the inputs to the system. In symbolic execution constraints over input values are generated during execution and these constraints are used to guide the execution so that every distinct execution path will be covered. Symbolic execution has gained a renewed interest among researchers since the recent advancements in decision procedures that now allow many path constraints to be solved efficiently. In this paper we give an overview of the techniques used in symbolic execution extension of a model checking tool called Java PathFinder and show how it can be used to symbolically execute sequential and concurrent Java programs. We will describe what parts of the tool are currently functional and what parts are planned or under development. To give the reader a better understanding of the advantages and disadvantages of the tool, we also describe shortly two other tools using symbolic execution.

1 Introduction

Ensuring correctness and reliability of software systems is an important task. Faulty systems can have great financial implications and correct behaviour of software is especially important in safety critical systems. Testing is a widely used method for finding errors but it can be costly and time consuming as it often requires manual generation of test cases. Testing also cannot prove the absence of errors and does not work well with concurrent systems. Model checking on the other hand is good at analysing concurrent systems but the state explosion problems restricts model checking to small systems that usually have to have bounds on their input sizes.

In this paper we give an evaluation of the current state of the symbolic execution extension of Java PathFinder (JPF-SE) that tries to address the problems stated above. The extension uses symbolic execution [10] and it can be applied for automatic generation of test cases and it allows model checking of systems with unbounded inputs that can be complex dynamic structures.

Java PathFinder (JPF, http://javapathFinder.sourceforge.net/) is an explicit state software model checker that can be used for verifying Java byte-code programs. It uses a custom-made Java Virtual Machine that supports all features
int example(int x, int y) {
1: int temp = x;
2: if (y > 0)
3: x = x + y;
4: y = temp;
5: if (x < y)
6: assert (false);
7: return x + y;
}

Figure 1: Simple example

of Java and in addition nondeterministic choice. JPF explores all execution paths of the given program taking thread interleavings into account. It backtracks when a previously explored state is visited and the user can also guide the search by defining different search heuristics or specifying conditions when the search backtracks. In its default settings JPF can be used to check for uncaught exceptions, assertion violations and deadlocks. JPF previously contained support for checking LTL properties but the use of this feature is now discouraged in current versions. In this work we are using JPF version 4.1 (revision 221).

The general plan of this work is as follows. In Section 2 the concept of symbolic execution is introduced and the details of how JPF-SE uses it are given. Also the topic of code instrumentation for symbolic execution is discussed. Section 3 gives an overview of the experiments done with the extension and tells how it can be used for model checking and test input generation. In Section 4 we go through some hands on experiments to demonstrate the use of the extension. Section 5 contains a short overview of CUTE/jCUTE and EXE, tools that are also using symbolic execution. Section 6 concludes the paper.

2 Symbolic Execution

In symbolic execution [10] the program variables that normally contain concrete values are replaced with their symbolic counterparts that express a range of possible values using symbolic expressions. The states of a single threaded system now contain the symbolic values of the expressions, a path condition that is a set of constraints that the values have to satisfy for the path being considered and a program counter indicating the next statement to be executed.

When executing a program the path condition is updated on branching points (e.g. when encountering an if statement). The branching condition restricts the range of values the variables can have if the branch is taken. Knowing the branching condition and the symbolic values of the variables, the path condition can be updated and it can be checked if the new condition is still satisfiable. If it is not, the branch cannot be taken and can be safely ignored. Otherwise the original program can take the branch and belongs to the possible behaviours of the system.
Figure 2: Symbolic execution tree
**Example 1.** To illustrate the idea of symbolic execution let us look how it can be applied to check a simple program presented in Figure 1. The program takes two integers as input and does some calculation with them. After this calculation, if $x$ is less than $y$, the assertion on line 6 will break. The symbolic execution on this example can be seen as forming a symbolic execution tree shown in Figure 2.

In the initial state the variables $x$ and $y$ are given symbolic values $X$ and $Y$ respectively. Line 1 in the code causes the variable $temp$ to have the same symbolic value as $x$. After line 2 the symbolic execution branches. In the first case the path condition is updated to $Y > 0$, as $y$ contains the symbolic value $Y$ when the if-statement is encountered. The second case will contain those symbolic values that do not satisfy the path condition on the first branch, i.e. the path condition becomes $Y \leq 0$.

If we take the left branch on the symbolic execution tree, variables $x$ and $y$ will be updated on lines 3 and 4. After this a new branching point is encountered. Now because $x$ has a symbolic value of $X + Y$ and $y$ a symbolic value of $X$, the path condition that would lead to the assertion statement is $Y > 0 \land X + Y < X$. However, this condition is unsatisifiable meaning that this branch can be ignored. The algorithm now backtracks and continues on another branch that has not yet been checked. When all branches have been checked, the algorithm terminates. In this particular example the program does not execute the assert statement no matter what input values the program is given.

### 2.1 Implementation in JPF-SE

In JPF-SE the idea of symbolic execution is extended to allow execution without initialisation of the input variables and to add support for complex data structures. The main task of the extension is to guide the underlying model checker, in this case JPF, to inspect the different paths of a symbolic execution tree. This allows that the normal state exploration techniques of JPF are available to be used, enabling for example symbolic execution of multi-threaded systems as JPF is capable of generating different thread schedules.

To handle uninitialised inputs to the system under verification, JPF-SE uses so called lazy initialisation [10]. The basic idea is that uninitialised variables will be initialised when they are first met during the execution of the program. The initialisation is done as follows. If the variable is of a primitive type, it will be initialised to a new symbolic value corresponding to the primitive type. If the uninitialised variable is a reference variable, it will be nondeterministically initialised to `null`, to a reference to a new object with uninitialised variables or to a reference to an object created earlier in the process. This way all the possibilities will be taken into account whether the variable points to a new object or is an alias to some existing one. This naturally allows the use of data structures like linked lists and trees to be used as inputs.

For input arrays the lazy initialisation proceeds in a similar way [3]. The array has a symbolic value representing its length and a set of array cells. These cells contain symbolic values for the index of the cell and for the actual data in the cell. When an uninitialised cell is encountered during the symbolic execution, it is initialised nondeterministically to new cell or to a cell that was initialised at some earlier time.
The path condition needs to be also updated so that it can be checked that
the index stays within the size of the array.

If the uninitialised reference variables are known to have some restrictions to their
values, it is possible to use method preconditions to prevent these variables to be
initialised in a way that conflicts with the restriction. After the initialisation it will
be checked if the precondition is broken and if it is, JPF is forced to backtrack and thus
the branches with infeasible variable values will not be checked. It is also important to
notice that if preconditions are used to limit the possible inputs to the system, these
preconditions have to be conservative. This means that when an input structure has
uninitialised fields after the lazy initialisation, it is not always possible to say if an
input structure will satisfy the condition after the field is initialised. In these cases
the inputs have to be seen as valid or otherwise some valid inputs could be excluded
from the symbolic execution.

Example 2. To get a better understanding of lazy initialisation, imagine that we
are in the process of symbolically executing a program that has gotten a linked list
as its input. If the current statement accesses a third element of the list where the
previous two elements have already been initialised, the lazy initialisation algorithm
can choose nondeterministically from four different possibilities. The reference to the
third element can be null, making the list two elements long. The reference to the third
element can also point to either the first or the second element, making the list cyclic.
The last possibility is that the reference points to a new element of the appropriate
type. By using preconditions it is possible for example to restrict the input structure
be an acyclic list, in which case JPF would backtrack when a reference creating a
cycle was generated.

When a branching condition is encountered during the symbolic execution, two
possible paths are generated. One with the branching condition added to the path
condition and one with the negation of the branch condition added. JPF-SE uses off-
the-shelf decision procedures to check if the updated path condition is satisfiable and
backtracks if it is not. If the satisfiability of the path condition cannot be determined,
JPF will backtrack also in this case. This way spurious counterexamples can be
avoided but some possible execution paths might be ignored.

JPF-SE supports Omega library [12], CVC-lite [4], YICES [6] and STP [3] as
decision procedures. In [2] it is mentioned that RealPaver [8] support is also added
but in the revision evaluated in this paper, the use of this constraint solver was not
yet available. In JPF-SE there are three ways of communicating with the decision
procedures [2]. In the file interface all queries and their results are sent using files
and the decision procedure is started every time a query is sent. This makes this
interface simple but slow in practice. In the pipe interface the decision procedure is
kept running together with JPF and the communication is done through a pipe. In
the native interface the communication to the decision procedure is done directly by
using Java Native Interface. This method is usually the fastest of the three.

2.1.1 Limiting the state space

Even if a symbolic state can represent potentially an unbounded number of concrete
states, a system may still have an unbounded number of symbolic states. Therefore
we need some way of limiting the space of possible symbolic states.

One simple way is to use bounded model checking by limiting the input sizes to the system and setting a maximum search depth for the model checker. Of course, this way only a subset of possible behaviours of the system will be checked.

In [3] a technique for checking subsumption of two symbolic states was presented. A symbolic state $s_1$ is said to subsume another symbolic state $s_2$ if the set of concrete states represented by $s_1$ contain all the concrete states represented by $s_2$. Now if a state that is found to be subsumed by some other state is encountered, it is safe to backtrack. For example imagine a case where a part of a program has been found free of errors when it is given a symbolic list with one element followed by an uninitialised element. Now if this part is given a longer symbolic list followed by an uninitialised element and the symbolic states are otherwise similar, it is not necessary to check the part of the program again, as all concrete states represented by the symbolic state have already been covered.

To check for subsumption, in JPF-SE symbolic states contain a so called symbolic heap configuration. This heap configuration is a graph that has a set of nodes consisting of reference variables and dynamically allocated objects in a particular state. In addition to these the graph has nodes representing `null' and uninitialised objects. The edges of the graph tell to what object each reference variable or object points to, or if it is `null' or not yet initialised. A simple example of a heap configuration representing a binary tree with two initialised nodes and one uninitialised subtree is given in Figure 3.

A precondition for symbolic state subsumption is that if a state subsumes another state, its heap configuration must also subsume the heap configuration of the other state. In JPF-SE a simple algorithm for testing this has been developed. The algorithm takes two heap configurations, $H_1$ and $H_2$, as input and starts traversing the heap graphs from their root nodes in the same order. For each node in the graph $H_2$ the algorithm tries to find a matching node from the graph $H_1$. If a matching node is found, both nodes are given a same unique label. If a matching node cannot be found, the algorithm returns false. If an uninitialised node is encountered in $H_1$, it can be matched only with an uninitialised node in $H_2$ and if such a node is encountered in $H_2$ it can be matched with arbitrary subgraph in $H_1$. If all nodes in $H_2$ can be matched, heap configuration $H_2$ subsumes heap configuration $H_1$.

The complexity of this algorithm is $O(n)$ but although it guarantees that heap
configurations are subsumed if the algorithm returns value true, it does not return true for all inputs where the heap configurations are subsumed. For a precise definition of heap configurations and the algorithm for subsumption for heap configurations, the reader is referred to [3].

In addition to checking subsumption of heap configurations it is necessary to compare the values of the variables and path conditions in symbolic states. Recall our simple list example. If the first element of the list contains a symbolic integer value that represents values greater than zero, a state where the list has a subsumed heap configuration but a symbolic value that can be less than zero cannot be subsumed by the state containing the first list.

The labelling constructed by the subsumption algorithm of heap configurations can now be used to identify the objects whose symbolic values have to compared. Now if the heap configuration of state $s_2$ subsumes the heap configuration of state $s_1$ and if the path condition and the constraints on symbolic variables according to the labelling are more general in state $s_2$ than in state $s_1$, the state $s_1$ is subsumed by the state $s_2$.

2.1.2 Abstractions

Subsumption cannot guarantee by itself that the symbolic state space will be finite. To address this and to make better use of subsumption, methods for creating abstract symbolic states have been developed. The abstractions presented in [3] are used to compute under-approximations of the behaviours of the system under testing. This way no spurious counter-examples will be reported but it cannot be guaranteed that all behaviours of the system will be checked.

One possible way of abstracting linked lists is to introduce a notion of summary nodes. The idea is that a group of adjacent list elements can be summarised in a single abstract element. The valuation of this summary node is then set to be a disjunction of the valuations of the nodes that were summarised. Now we can summarise some nodes of a linked list and check if the state with this new abstracted linked list is subsumed by a state visited earlier. With this abstraction the number of possible heap configurations can be made finite, but when we take the constraints on the variables into account, the number of symbolic states may still be infinite.

With small changes the abstraction for linked lists can also be applied to arrays if we think of arrays as singly linked lists with nodes as array elements and keeping the nodes in the same order as the array elements. Another useful abstraction especially with tree-like structures is to discard the actual data and record only the shape of the structure in a state.

2.2 Instrumentation

To execute a program symbolically its source code needs to be instrumented first [10]. The concrete variables and the operations on them have to be changed into their symbolic counterparts. JPF-SE currently provides classes for manipulating symbolic integer expressions (Expression class) and string expressions (StringExpression class). JPF-SE supports also symbolic arrays that contain Boolean or integer type variables.
These classes contain the methods that operate on symbolic variables (e.g. different comparison operators, addition and multiplication for integers).

Statements accessing or updating variables need to be instrumented also. Accessing is done by using `get` methods that use lazy initialisation if the variable has not yet been initialised. For updates `set` methods are used. These methods also set a flag that tells that the variable has been initialised.

Simply instrumenting all the variables in a program does not necessary lead to a program that can be symbolically executed. There are two problems with symbolic execution that arise especially with real systems [1]. First, the decision procedure being used to check the satisfiability of the path conditions might not be able to solve all kind of constrains that will be generated during the symbolic execution. For example an encountered branch condition might use unsupported operations (e.g. use a modulo operator). Second, if a concrete variable is transformed into a symbolic variable, it has to be possible to instrument all the statements that will use the original variable. This is a problem if for example third party libraries that cannot be instrumented or Java native methods are used.

If these kind of problems are encountered, they cannot be solved automatically. The user might be able to write the problematic part differently or use some other methods instead of uninstrumented library calls. This however requires that the problematic points of the system have to be identified.

### 2.2.1 Type analysis

Type-dependence analysis has been proposed in [1] to tackle problems in instrumentation. For another approach, see the discussion about CUTE in Section 5. The idea of type-dependence analysis is to find all the variables and objects that are dependent on some given variable or object. Following the definition in [1] we can say that an entity $e_1$ is type depended on an entity $e_2$ if and only if the type of $e_1$ may need to be changed when the type of $e_2$ is changed in order to keep the program type-correct. If a variable is wanted to be symbolic, type-dependence analysis can be used to locate all the other entities that have to be transformed into their symbolic counterparts. This analysis also can be used to locate the parts of the code where the problems stated in the previous subsection can occur. Transforming only the parts that are shown to be necessary to transform according to type-dependence analysis, gives us two additional advantages compared to transforming the whole program. Using symbolic variables has always some computational overhead in comparison with using normal variables. This means that transforming only the necessary parts will result in better performance when the transformed program is verified. Second benefit is that some parts of the code where the decision procedure used could face problems can be avoided if those parts are not needed to be transformed. This way the user can concentrate only on a subset of the code that can potentially cause problems.

An outline of an algorithm that can be used to calculate a conservative approximation of the type-dependence of a given set of entities in relation to other entities in the program under verification is described in [1]. The algorithm works in two phases by constructing first a Type-Dependence Graph and after that performing Context-Free Language reachability on the constructed graph.
Type-Dependence Graph (TDG) is a directed graph where the set of nodes contain the entities (e.g. local variables and parameters of methods) in type-analysis and the edges \((u, v)\) describe that entity \(u\) is directly type dependent on entity \(v\). To construct a TDG, each program statement in a given program is analysed once and specific rules are used to determine if the statement causes an edge to be added to the graph.

Context-Free Language (CFL) reachability is used to find all the entities that are type-dependent on a given initial set of entities. These initial elements represent the variables that the user wants to be symbolic. The algorithm works by inspecting the TDG and calculating an extended set of entities that are type-dependent on the initial entities. Then the same procedure is done again with the initial set replaced with the extended set and the process is continued until no new entities are added to the set. Specific field- and context-sensitive rules are used to identify the type-dependent entities.

### 2.2.2 Transformation

After type-dependence analysis the program can be transformed into symbolically executable program. The algorithm presented in [1] adds new symbolic counterparts for each variable and method that have to be transformed without removing the originals. For each method created this way, the content of the original method is copied into the symbolic method and the original is transformed to work as a proxy that can be used with concrete values.

Allowing concrete values makes it possible that the whole code does not have to be instrumented as was discussed above. To retain type-correctness there has to be special operators that can change a concrete value to its symbolic counterpart and vice versa. Given these operators it is now possible to allow proxy methods to do the conversion and to use simple transformation rules to transform the statements one by one in the newly created methods.

### 2.2.3 Stinger

The currently available version of JPF-SE does not include a tool for automatic instrumentation of source code. We managed to get for testing an early prototype version of a tool called Stinger that is not yet publicly available. In Stinger the previously discussed algorithm for type-dependence analysis and program transformation is implemented.

To use the tool, the user has to first indicate the variables that are meant to be symbolic by instrumenting manually the parts of the code where these variables are initialised. This is done by using a special class provided by the tool. For examples, see the discussion in Section 4. This modified code is then compiled and given to Stinger. The tool uses SOOT (http://www.sable.mcgill.ca/soot/) a Java optimisation framework to translate the byte-code into a more suitable form for the transformation. Jeddl (http://www.sable.mcgill.ca/jeddl/) a Java extension for decision diagrams is used in the type-dependence analysis process (the Type-Dependence Graph is implemented as a Binary Decision Diagram). The analysis in Stinger is an approximation that can result in transformation of unnecessary variables or methods. During the analysis
those parts of the code that are problematic for symbolic execution are identified and
the tool is supposed to report the found problems to users. However, in the prototype
we tested, this feature was not yet fully functional.

After analysis Stinger instruments the given code and uses JODE to translate the
instrumented code back to normal Java source code. This source code can then be
compiled and given to JPF for model checking or testing.

Some additional information about the tool can be found in Section 4 where it is
used for experiments.

3 Applications

To get a better understanding of the applicability of the symbolic execution extension
of Java PathFinder, we give a short survey of different areas where the extension can
be used and present some experiments that have been done to evaluate the feasibility
of the approach.

3.1 Model checking and test input generation

Model checking programs with large input domains becomes quickly infeasible espe-
cially when concrete values for variables are used. Symbolic execution allows better
scalability as it explores only those execution paths that are distinct to each other
where as with concrete values the same execution path could be explored several times.
Even with symbolic execution real systems are often too complex to model check and
only a subset of systems behaviours can be verified making model checking a form of
testing in those cases. Although JPF-SE gives promising results on small programs,
few experiments have been done on applying JPF-SE on more complex or real systems.
More work is needed to evaluate how well the approach scales when the complexity
of the system under testing increases. Most research has been done on evaluating
JPF-SE on complex data structures such as different trees in Java containers. The
result of these experiments are discussed in the following subsection.

In addition to checking a system for errors directly with JPF-SE it is also possible
to use the extension to generate test inputs. In \cite{14, 15, 16} a framework is presented
to automatically do this test input generation. The framework allows test input gen-
eration for black box testing in cases where it is applicable and also input generation
during white box testing. In black box testing the source code of the system cannot
be used but if a description of the input preconditions is available, it is possible to
write this description as a program and use symbolic execution on this program to
generate inputs. As an example this method could be used to automatically generate
all different red-black trees up to a given size. Because the source code is not available,
this method does not allow generation of test inputs that would necessarily lead to
good code coverage. It is also important to notice that the program that describes
the input preconditions have to be written so that it does not evaluate the validity of
a given input only at the end. This could lead into generating all the possible input
structures and eliminating the non-valid candidates only one by one.

When the source code is available it is possible to generate test inputs that satisfy
a given testing criterion by using JPF-SE to find the symbolic execution paths where the criterion holds. This is done by writing the criterion as a property that JPF can check and so JPF reports the executions that satisfy the criterion as counterexamples. The path conditions in these symbolic execution paths are then solved to find concrete values for inputs and to change symbolic input structures into their concrete counterparts. Because the aim here is to generate inputs, it is necessary to be able to handle cases where the system does destructive updates on the input structure, i.e. the symbolic input will be modified during the execution. With a special mapping between the original object that can have uninitialised fields after lazy initialisation and the object after a destructive update, it is possible to reconstruct the right input structure for the system.

3.2 Test input generation for Java containers

In [16] different techniques for test input generation, including ones using symbolic execution, are evaluated using Java container classes as the systems for which the inputs are generated. More specifically these containers are implementations of a binary tree, a binomial heap, a Fibonacci heap and a tree map. For all techniques basic block coverage and predicate coverage is measured. Predicate coverage means here that a set of predetermined predicates are selected and it is measured how well the test input generation methods cover all the combinations of these predicates. The methods used in the evaluation are all implemented using JPF and are divided into exhaustive and lossy techniques depending if they have the ability to explore all the behaviours of the system.

The exhaustive methods include normal model checking with bounds on the input sizes, model checking with an exact abstraction of the system states which records the shape and data of the container allowing thus better use of heap symmetry reductions during model checking. The last exhaustive method is symbolic execution with symbolic subsumption.

Lossy techniques include bounded model checking with an abstraction that records only the shape of the container but not the actual data values, symbolic execution with this same shape abstraction and testing with random inputs.

The results of the evaluation shows, that lossy techniques obtain better block and predicate coverage than exhaustive techniques. In case of a Fibonacci heap none of the exhaustive methods got optimal block coverage where as lossy methods obtained optimal block coverage for all cases. From the exhaustive methods symbolic execution performed clearly the best and got good predicate coverage even compared to lossy methods with an exception of the case with Fibonacci heap.

From the lossy methods, bounded model checking with shape abstraction had the best performance showing that shape is a good representation of a state at least in case of the containers that were used. Symbolic execution with shape abstraction did not get as good coverage in all cases but was not far behind although the time and memory usage was considerably larger when compared to model checking with shape abstraction.
3.3 Test data generation for programming exercises

Using model checking to find program errors in industry is not the only area that can benefit from symbolic execution. The possibility of using JPF-SE for evaluating and giving feedback of programming exercises in computer science education is discussed in [9].

Symbolic execution of a student’s solution to a programming assignment against a specification can be used to give more comprehensive feedback to the student if errors are found in the solution. Model checking with concrete values would report that the solution does not work with some specific concrete values. With symbolic execution it is possible to inform the student that the solution does not work correctly for example with input values that are negative. This more abstract feedback gives the student better understanding of possible cause of the error.

4 Experiments

In this Section we demonstrate the use of the symbolic execution extension of Java PathFinder by using it to check errors in two simple Java programs. The first program is a single threaded insertion sort algorithm that sorts integer arrays and the second program is a slightly more complex multi-threaded program in which information is passed through multiple threads. We will use Stinger to instrument both of these examples.

4.1 Insertion sort example

Insertion sort is a simple but inefficient algorithm that can be used to sort arrays. Our version of the algorithm and a program that uses it is presented in Figure 4 on the left side and the instrumented version of the same program is shown on the right side. The program includes a method that takes an array as input and uses insertion sort to sort it in ascending order and a method that checks if a given array is sorted correctly, which can be seen as a postcondition for the algorithm. As the version of Stinger we had available for testing does not support unbounded arrays, we choose to instrument the program so that it takes as an input a fixed size array that contains symbolic integers.

Stinger provides a class with a set of methods that can be used to indicate which variables the user wants to be symbolic. In this example we created an array with five elements and set each element to be symbolic by using a method Symbolic.integer() as can be seen in the main method in Figure 4. No other changes to the original program code were needed. The program is then compiled and the compiled byte code given to Stinger which gives as output the translated Java code that is ready to be compiled and given to JPF.

We used JPF-SE with Omega library and native interface to check the instrumented code for errors. As can be expected, no errors was found. The method isSorted along with assert statement was used as a simple way to check that the sorting algorithm does give correctly sorted array as output. We also modified the code by inserting bugs in it. If the check that \(j > 0\) is removed from the while statement
Figure 4: Insertion sort

of the algorithm, JPF correctly reports an array index out of bounds exception and if the assignment statement after the above-mentioned while statement is removed, JPF finds assertion violation as the algorithm no longer works correctly.

We also used JPF to check the same program with different array lengths. For an array of length five JPF generates 719 states which is well less what the number of concrete states would be if every integer value for each element was inspected separately. Although this result looks promising, it is important to notice that the number of states JPF visits starts to grow factorially as the length of the array increases. It can be seen that the number of end states JPF visits is in fact $\theta(n!)$ where $n$ is the length of the array. This is because before sorting the elements can be in any permutation depending on the input and the symbolic execution paths are different for each permutation. This means that the method used here does not scale well for large arrays as the number of states starts growing too quickly. Subsumption checking and abstractions could be used here to tackle this problem, but as these areas of JPF-SE were not documented and not fully developed, such experiments were deemed to be outside the scope of this work.

4.2 Multi-threaded example

To test JPF-SE with multi-threaded programs we instrumented a pipeline example program from [7]. The program demonstrates a case where a number of threads communicate with each other by using blocking message queues. A thread tries to read a
class Pipeline {
    static int num = 3;

    public static void main (String argv[]) {
        BlockingQueue first, in, out;
        first = in = out = new BlockingQueue();
        for (int i = 0; i < num; i++) {
            out = new BlockingQueue();
            (new Stage(in,out)).start();
            in = out;
        }
        (new Listener(out)).start();
        first.add(7);
        first.add(-1);
    }
}

final class BlockingQueue {
    private int queue;

    public final synchronized int take() {
        int value;
        while (queue == 0)
            try { wait(); }
        catch (InterruptedException ex) {}
        value = queue;
        queue = 0;
        return value;
    }

    public final synchronized void add(int o) {
        queue = o;
        notifyAll();
    }
}

final class Listener extends Thread {
    protected BlockingQueue input;

    public Listener(BlockingQueue in) {
        input = in;
    }

    public void run() {
        int tmp;
        while (true) {
            tmp = input.take();
            if (tmp == -1) break;
            System.out.println(tmp);
        }
    }
}

final class Stage extends Thread {
    protected BlockingQueue input, output;

    public Stage(BlockingQueue in, BlockingQueue out) {
        input = in;
        output = out;
    }

    public void run() {
        int tmp;
        while (true) {
            tmp = input.take();
            if (tmp != -1) tmp = tmp + 1;
            output.add(tmp);
            if (tmp == -1) break;
        }
    }
}

Figure 5: Before instrumentation

message from its input queue and blocks if no message is yet available. When a thread reads a new message, it processes it, in our case simply increases the read integer by one, and adds the processed message to its output queue making the message available to another thread. On a predefined message, in our case on receiving message \(-1\), the thread terminates after forwarding the message. The program also has a listener thread that will print the message it receives but does not send it forward. The original program code where two integers are added to the input queue of the first thread is presented in Figure 5.

For comparison and initial testing we first checked the code without JPF-SE by making the program choose nondeterministically a value between \(-3\) and 3 using JPF’s Verify-class and give this value as the first input to the first thread. As the second message we sent the termination message to end the program. This way the system is checked separately with each of these inputs while JPF automatically checks different possible thread interleavings. For this test JPF reported no errors.

To instrument the program to be used with JPF-SE we changed the first input message to be a symbolic integer. With this modification the program was given to Stinger and the resulting instrumented code can be seen in Figure 6. Stinger generates a new Java source file for each class found in the input program and in Figure 6 the four generated files are shown grouped together. We used again the native Omega interface in JPF-SE to check the instrumented code for errors.

As the initial checking without JPF-SE found no errors, we were surprised that JPF reported a null pointer exception in BlockingQueue class while checking if the queue had the value zero, meaning in this context that the queue did not contain any messages. This kind of error cannot happen with the uninstrumented code because
the type of the variable queue is primitive integer that has zero as its default value and cannot be set to be null. In JPF-SE a special class is used to represent symbolic integer values and an object of this class can be null if not initialised. In the instrumented code JPF found a thread interleaving where the symbolic queue integer was accessed before the symbolic input from main method reached it and caused the symbolic integer to be null while the comparison was made. A natural way to fix this is to initialise the symbolic integer with a symbolic constant zero, which makes the instrumented code work like the original code. However, this modification caused an unknown error in JPF-SE while it was trying to compare two symbolic constants with each other. To circumvent this error we initialised the queue variable with a symbolic value. This increases the possible behaviours of the system as the queues in different threads may not be empty initially but contain any values. The original behaviour of the system is included in this modification so the change does not exclude any previously existing errors.

The value printing command in listener thread caused also an error in JPF-SE and we choose to remove this line from the program as it only gives information to the user and does not affect the run of the program in any other way.

After these modifications to the instrumented code no more errors preventing us from checking the code was found. We ran JPF on the instrumented code but it did not finish in a time limit of 40 minutes using a 2 GHz Linux PC with 2 GB of RAM. We were, however, able to use JPF to find errors we inserted into the program code. In the first case we removed the check for termination message before a thread processes a message. This causes the termination message to be transformed into another message which in turn can cause the system to deadlock. JPF correctly found a possible input leading to deadlock situation. In the second case we inserted a condition in BlockingQueue class that on a certain predefined value the program will try to do a division by zero. Finding this single input for example with random testing would be difficult. JPF correctly reports in this case an uncaught arithmetic exception.

5 Other Tools

In this Section we discuss shortly two other tools also using symbolic execution to find program errors and describe their main differences in comparison with JPF-SE. CUTE and jCUTE can be seen as an attempt to bring testing closer to model checking and EXE is a tool that generates automatically inputs that will cause a program to crash.

5.1 CUTE/jCUTE

CUTE and jCUTE [13] are testing tools for sequential C programs and concurrent Java programs respectively. The main idea in these tools is to combine symbolic and concrete execution by using a technique called concolic testing. In this method a program is executed by giving concrete values to the symbolic inputs of the system. The execution then follows the path that is taken by the concrete values. When state-
import edu.gatech.translate.Symbolic;

class Pipeline {
    static int num = 3;

    public static void main(String[] strings) {
        BlockingQueue blockingqueue = new BlockingQueue();
        BlockingQueue blockingqueue_0_ = blockingqueue;
        BlockingQueue blockingqueue_1_ = blockingqueue;
        for (int i = 0; i < num; i++) {
            BlockingQueue blockingqueue_2_ = new BlockingQueue();
            blockingqueue_0_ = blockingqueue_2_;
            new Stage(blockingqueue_1_, blockingqueue_2_).start();
            blockingqueue_1_ = blockingqueue_2_;
        }
        new Listener(blockingqueue_0_).start();
        blockingqueue.add(Symbolic.symbolicInteger());
        blockingqueue.add(-1);
    }
}

import gov.nasa.jpf.symbolic.integer.Expression;

class BlockingQueue {
    private int queue;
    private Expression queue_JPF_;
    public final synchronized int take() {
        return Symbolic.fromintConstant(take_JPF_());
    }
    public final synchronized void add(int i) {
        add_JPF_(Symbolic.intConstant(i));
    }
    public final synchronized void add_JPF_(Expression expression) {
        queue_JPF_ = expression;
        this.notifyAll();
    }
    public final synchronized Expression take_JPF_() {
        while (queue_JPF_._NE(Symbolic.intConstant(0)) != true) {
            try {
                this.wait();
            } catch (InterruptedException interruptedexception) {
                return;
            }
        }
        Expression expression = queue_JPF_; 
        queue_JPF_ = Symbolic.intConstant(0);
        return expression;
    }
}

import gov.nasa.jpf.symbolic.integer.Expression;

class Listener extends Thread {
    protected BlockingQueue input;
    public Listener(BlockingQueue blockingqueue) {
        input = blockingqueue;
    }
    public void run() {
        run_JPF_();
    }
    public void run_JPF_() {
        for (;;) {
            Expression expression = input.take_JPF_();
            if (expression._NE(Symbolic.intConstant(-1)) != true) {
                System.out.println(Symbolic.fromintConstant(expression));
            } else break;
        }
    }
}

import gov.nasa.jpf.symbolic.integer.Expression;

class Stage extends Thread {
    protected BlockingQueue input;
    protected BlockingQueue output;
    public Stage(BlockingQueue blockingqueue, BlockingQueue blockingqueue_0_)
        (input = blockingqueue; output = blockingqueue_0_;)
        { this.notifyAll();
    }
    public void run() {
        run_JPF_();
    }
    public void run_JPF_() {
        Expression expression;
        do {
            expression = input.take_JPF_();
            if (expression._NE(Symbolic.intConstant(-1)) != true) {
                expression = expression._plus(Symbolic.intConstant(1));
                output.add_JPF_(expression);
            } else while (expression._NE(Symbolic.intConstant(-1)) == true;
        } while (expression._NE(Symbolic.intConstant(-1)) == true;
    }"
ments using these values are encountered during the execution, path conditions are
generated similarly to JPF-SE. At the end of the execution a custom-made constraint
solver is used to calculate new concrete values from the path constraints such that
the values will cause the next execution to follow a different path. This process is
iterated until all execution paths have been covered. If the constraint solver cannot
solve the path condition, it is approximated by replacing the symbolic values with
concrete ones. This allows CUTE to inspect some execution paths following the state
where the approximation was made, but does not guarantee that all execution paths
will be covered.

To use concolic testing, the program to be tested is first instrumented so that it
can maintain a symbolic state and get the correct input values and calculate new
ones for the next iteration. The instrumented program is then, in case of jCUTE,
rung using a normal Java Virtual Machine. This means that concolic testing cannot
test different thread schedules of a concurrent program by itself. For this reason
jCUTE is extended with an algorithm for race-detection and thread flipping. The
algorithm detects events where different threads access a common memory location
without holding a common lock and computes different thread schedules and input
values that will cause all different executions to be covered.

For testing a program against a formal specification (e.g. a temporal safety prop-
erty), jCUTE is also extended with a technique called predictive monitoring. It is
a form of runtime monitoring where a program is instrumented to check if the spec-
ification is violated. Predictive monitoring can generate equivalent execution paths
from an observed path and thus predict specification violations in different equivalent
execution paths when only one of them has been explored.

One problem with CUTE and jCUTE is that if the tree of possible execution paths
is so large that CUTE does not terminate given a reasonable time limit, the tool might
search only a small local part of the tree depending on the first concrete execution.
In [11] a suggestion to alleviate this problem by combining concolic and random
testing is presented. This so called hybrid concolic testing is started with random
testing. When no new code coverage points are generated on the random path after
some predefined step limit, the testing is changed to concolic testing starting from the
state where random testing ended. After concolic testing finds a new coverage point,
random testing is continued. This kind of interleaving can be beneficial for certain
kind of programs and experiments show that it can result in two-fold increase in the
branch coverage when compared to normal concolic testing.

5.2 EXE

EXE [5] is a tool that is designed to find input values for programs that will cause the
program to crash. The basic idea of using path constraints is similar to the approaches
discussed before. However, EXE is limited to testing sequential programs only. Like
with JPF-SE and CUTE, the program to be tested is instrumented first. In EXE the
user marks the variables that are supposed to be initially symbolic. After this the rest
of the program is instrumented automatically to check every statement whether it uses
symbolic values. When the instrumented program is run, all statements except the
ones handling symbolic values are executed normally. If an assignment with symbolic
values is encountered, the path condition is updated and in case of a branching point, the execution is forked. The path condition of the first thread is updated according to the branching condition and the second thread created by the fork operation is updated with a negation of the branching condition. During instrumentation it is also checked if a statement with symbolic values could cause an error (e.g. out-of-bounds memory reference or division by zero) and the code is modified to report this if necessary.

EXE has been developed together with a constraint solver called STP that the tool uses. STP is a decision procedure for bit vectors and it can handle more types of constraints than for example the custom constraint solver in CUTE that can be used only with linear constraints. Naturally solving linear constraint is normally faster on decision procedures that have been specifically designed for such constraints than on more general decision procedures [1].

The results obtained with EXE have been promising. For example the tool has been used to generate disc images for different Linux file systems that will cause errors in a Linux kernel and to find errors in a real DHCP server implementation.

6 Conclusion

We have described the idea behind symbolic execution and how it is implemented in an extension of Java PathFinder. We also showed how the extension with an early version of an instrumentation tool can be used to check simple Java programs.

As the instrumentation tool and the extension itself are still under active development with unfinished parts and the documentation on how to use the tools is on most parts not yet written, we were not able to test all the features described in this paper and could not test the tools on more complex and real-like applications. More experiments are thus needed to evaluate how well the extension and symbolic execution in general scales for larger programs. On small scale the results obtained with the tool have been promising, showing that the approach is feasible, but it is good to keep in mind that symbolic execution has its limits when the size the symbolic execution tree becomes large as happened with the sorting experiment in Section 4.

Especially with concurrent systems the number of possible execution paths becomes quickly intractable and methods for limiting the number of paths to be explored are needed. One approach taken in JPF-SE is to use abstractions and subsumption of symbolic states. One other problem faced with JPF-SE is that the instrumentation of the program code cannot be always fully automated if the path condition is updated with operators that the decision procedure used does not support or if the constraint becomes too difficult to solve. This is a problem for other tools as well. The approach taken in CUTE results in approximating difficult constraints with concrete values which leads in random testing. In other words, much of the power of symbolic execution is depended on the underlying decision procedures. Further research is required to address these problems.
References


