Symbolic PathFinder: Integrating Symbolic Execution with Model Checking for Java Bytecode Analysis

Corina S. Păsăreanu · Willem Visser · David Bushnell · Jaco Geldenhuys · Peter Mehlitz · Neha Rungta

Received: date / Accepted: date

Abstract Symbolic Pathfinder (SPF) is a software analysis tool that combines symbolic execution with model checking for automated test case generation and error detection in Java bytecode programs. In SPF, programs are executed on symbolic inputs representing multiple concrete inputs and the values of program variables are represented by expressions over those symbolic inputs. Constraints over these expressions are generated from the analysis of different paths through the program. The constraints are solved with off-the-shelf solvers to determine path feasibility and to generate test inputs. SPF incorporates techniques for handling input data structures, strings, and native calls to external libraries, as well as for solving complex mathematical constraints. Model checking is used to explore different symbolic program executions, to systematically handle aliasing in the input data structures, and to analyze the multithreading present in the code. We describe the tool and its application at NASA, in academia, and in industry.

Keywords Symbolic execution \cdot Model checking \cdot Testing \cdot Java

1 Introduction

Symbolic execution [15, 36] is a popular analysis technique that performs execution of a program on symbolic values rather than concrete data inputs. The technique systematically collects input constraints along the executed program paths while computing the program effects as algebraic expressions in terms of the symbolic values. Symbolic execution was introduced in the 70s, but only recently it has found wider applicability in practice. This is due to recent algorithmic advances and to the increased availability of powerful constraint solving technology and computational resources [12]. We present Symbolic Pathfinder (SPF) – a tool for

Willem Visser · Jaco Geldenhuys University of Stellenbosh, South Africa E-mail: {wvisser, jaco}@cs.sun.ac.za

Corina S. Păsăreanu · David Bushnell · Peter Mehlitz · Neha Rungta NASA Ames Research Center Moffett Field, CA 94035, USA E-mail: {corina.s.pasareanu, david.h.bushnell, peter.c.mehlitz, neha.s.rungta}@nasa.gov

performing symbolic execution of Java bytecode. The tool is typically used for systematic generation of test cases that achieve high testing coverage and for checking safety violations, such as assert and concurrency errors, in programs with unspecified inputs. SPF handles inputs and operations on booleans, integers, reals, strings, and complex data structures [33]. SPF is tightly integrated with the Java PathFinder (JPF) model checker [32, 70]. The JPF framework provides SPF the ability to systematically explore symbolic program paths, different thread interleavings, as well as other forms of nondeterminism present in the code. Furthermore, SPF incorporates specialized techniques for handling external library calls [44] and for solving complex mathematical constraints [7]. SPF is a freely available open-source project [64]. A parallel version also exists [65].

SPF is quite general and it can be applied at different phases of software development. However, as it is not always possible to run the whole program within the SPF's customized execution environment (due to sheer size, native libraries, hardware-software interaction, etc.), SPF is most effective for unit or sub-system level testing. It is often the case that the inputs to the unit are constrained by the environment, e.g., the calling context of a method or set of methods representing the unit. To avoid exploration of unrealistic program paths, such constraints would need to be incorporated in the analysis. SPF addresses this problem in several ways. First, SPF allows symbolic execution to be started at any point in the program and at any time during the concrete execution of a program. In other words, one can run the program in "concrete execution mode" within SPF's specialized Java virtual machine and start symbolic execution based on some condition on the concrete program state. Thus, the concrete execution of the system can be effectively used to set up the environment for the symbolic execution of a unit in the system. Furthermore, one can analyze a unit symbolically, while some of the parameters and the calling context of the method are kept concrete. SPF also provides a mechanism for writing preconditions on the inputs, via annotations in the code. Such unit preconditions help improve the precision of the unit-level symbolic analysis by avoiding exploration of program paths that violate the constraints.

SPF has a diverse user base, both at NASA, where SPF helped uncover subtle bugs in flight software [44], in academia, where SPF is being used and enhanced for various research projects, and in industry; most recently SPF was used at Fujitsu to test web applications with over 60,000 source lines of code [39, 48, 60].

There are quite a few tools available that perform symbolic execution for programs written in modern programming languages (see Section 8). What distinguishes SPF from these tools is its ability to handle complex symbolic inputs and multithreading, and its *extensibility* as demonstrated by the many applications built on top of SPF and discussed in Section 7.

1.1 Tool Overview

A high-level overview of the tool is shown in Figure 1. SPF requires as input: (a) the class files of an executable program, (b) a configuration file specifying which methods in the program should be executed symbolically, and (c) properties being verified or any other coverage criteria. The configuration can contain other options to instruct SPF to use a specific constraint solver, treat complex data as symbolic, etc. The output of SPF is a test suite (test inputs or test sequences) and/or the counterexamples for the failed properties.

SPF relies on the Java PathFinder model checker (jpf-core) to systematically explore the different symbolic execution paths, as well as different thread interleavings. Furthermore, SPF uses JPF's built-in strategies for state space exploration, such as depth-first search or breadth-first search. To limit the possibly infinite search space that results from symbolically executing



Fig. 1 Symbolic Pathfinder Overview

programs with loops or recursion, a user-specified depth needs to be provided. The constraints created during symbolic execution are solved with off-the-shelf solvers to determine path feasibility and to generate test inputs. A generic interface facilitates the integration of new solvers and users can easily configure the symbolic execution to use any one of the supported solvers. Preconditions specified with user annotations on methods' inputs are leveraged to reduce the input data domains and to only generate test inputs that satisfy the preconditions.

1.2 Outline

The rest of this article is organized as follows. In the next section we give some background on symbolic execution. In Section 3 we describe the overall JPF framework and in Section 4 we describe in detail elements of SPF. Sections 5 and 6 provide an in-depth description of two recently developed techniques that further distinguish SPF from other symbolic execution tools: handling of native calls (through mixed concrete-symbolic solving) and symbolic string analysis. Section 7 describes some novel analyses that have been built by using SPF, demonstrating its usability and extensibility. A comparison with related tools is provided in Section 8 and Section 9 gives conclusions and future work.

2 Symbolic Execution

Symbolic execution [15, 36] is a program analysis technique that uses symbolic values as program inputs and symbolic expressions to represent the values of program variables. As a result, the outputs computed by a program are expressed as a function of the symbolic inputs. The state of a symbolically executed program includes the (symbolic) values of program variables, a path condition (PC), and a program counter. The PC is a conjunction of constraints over symbolic expressions that encodes the conditions on the input to follow a path through the program. The path associated with a PC can be executed concretely using input values that satisfy the constraints in the PC. The paths generated during the symbolic execution of a program are characterized by a symbolic execution tree.



Fig. 2 Example for symbolic execution (top left) and corresponding execution tree (right).

As an illustrative example [44], consider the code in Figure 2 (top left) that computes the absolute difference between two input integers x and y. The standard, concrete execution of the program will follow only one path through the code, based on the values of x and y. Given the values x=2 and y=1, the *true* branch of the *if* statement at line [1] is executed, and in turn the assertion is not violated.

Symbolic execution starts with symbolic, rather than concrete, input values, $\mathbf{x} = Sym_x$, $\mathbf{y} = Sym_y$, and sets the initial value of *PC* as *true*. The corresponding (simplified) execution tree is shown in Figure 2 (right). At each branch point, *PC* is updated with constraints on the inputs in order to choose between alternative paths. For example, after executing line [1] in the code, both alternatives of the **if** statement are possible, and *PC* is updated accordingly with the two different possibilities. When the path condition is unsatisfiable it becomes "false", which means that the corresponding path is infeasible. Symbolic execution does not continue further along an infeasible path.

For our example, symbolic execution explores three different feasible paths, with the following path conditions (see Figure 2 (right)):

- $-PC_1: Sym_x > Sym_y \wedge Sym_x Sym_y > 0$ for Path 1,
- $PC_2: Sym_x \leq Sym_y \wedge Sym_y Sym_x > 0$ for Path 2,
- $-PC_3: Sym_x \leq Sym_y \wedge Sym_y Sym_x \leq 0$ for Path 3.

Furthermore, PC_3 characterizes the concrete program executions that violate the assertion. This demonstrates that the code has an unstated precondition, namely that $x \neq y$. For test input generation, the obtained path conditions are solved using off-the-shelf decision procedures and the solutions are used as test inputs that are guaranteed to exercise all the paths through this code.

3 Java PathFinder

Java Pathfinder (JPF or JPF-core) [32] is an extensible run-time environment for verification of Java bytecode, i.e., compiled Java programs. JPF-core is available as the jpf-core project



Fig. 3 JPF's high-level structure.

from the JPF distribution [32]. JPF has been under development at the NASA Ames Research Center since 1999 and was open-sourced in 2005. Currently, JPF has an active user and developer base in academia, industry, and government agencies and labs. JPF was originally developed as a concrete-value, explicit-state software model checker for concurrent programs. Over the past few years, however, JPF has evolved into a extensible Java analysis framework for developing and exploring different verification techniques and application domains.

The inputs to JPF are: the class files (Java bytecode) for a system under test and a set of configuration text files which specify the desired JPF execution mode, program properties to verify, and artifacts to generate. The verification artifacts produced are usually reports in various formats.

An important feature of JPF is its extensibility that allows implementation of new execution modes, program property checks, report formats and user interfaces as run-timeconfigurable plugin modules (see Figure 3). Due to its origins as a software model checker, the core of JPF implements a Java Virtual Machine (JVM) that supports backtracking, state matching, and nondeterminism in both data and scheduling decisions. JPF constructs the program state space on-the-fly during the execution of the program in the special virtual machine. A transition in the state space is a sequence of bytecode instructions executed by a single thread, where the first instruction in the sequence represents a nondeterministic choice corresponding to a thread context switch. At every transition boundary, JPF saves the current JVM state (the program state) in a serialized form for the purpose of backtracking and state matching. Changes of the JVM state are performed inside the interpreter of bytecode instructions, which is also a part of JPF.

The following extension mechanisms of JPF are particularly relevant for SPF:

- Choice Generators state space branch points used to generate choices,
- Instruction Factories instruction set semantics,
- Attribute Objects metadata associated with concrete values and objects,
- Native Peers library abstractions,
- Listeners execution monitoring.

Some of these mechanisms had been extended specifically to support SPF. The attribute objects are used to store and propagate symbolic information while the instruction factories provide the ability to replace the standard execution semantics with a symbolic semantics. These mechanisms have now become sufficiently general to facilitate other applications as well, e.g. to keep track of physical dimensions and numeric error bounds or to perform taint analysis. The remainder of this section contains general descriptions for each of these constructs and mechanisms. Section 4 explains their concrete use within our SPF tool.

3.1 Choice Generators

JPF uses a generic model of the program state space consisting of *States*, *Choices* and *Tran*sitions. States are restorable snapshots of a program execution along a particular path. The snapshot contains the heap of the program, its current program location, current state of the various threads (running, waiting, blocked etc.), and the operand stacks of the corresponding threads. Note that in a sequential program there is only a single thread and operand stack. *Choices* are the discriminating execution contexts for possible executions leading out of a state, such as the choice of which thread to run next. *Transitions* are the sequence of executed instructions between two states, starting with a specific choice value and ending in an operation that constitutes the next choice.

JPF captures state space branch points in dedicated *ChoiceGenerator* objects, which keep track of the choice set for backtracking purposes. Typical examples of choices are threads representing different scheduling sequences and numeric values representing random numbers, which are both handled in core JPF. However, extensions can introduce their own choice types such as user input, or - as in SPF - branch condition values.

3.2 Instruction Factories

The standard Java Virtual Machine specification defines a set of bytecode instructions for operations such as invoking methods, accessing fields and pushing or popping values from the operand stack. JPF represents each of these bytecodes by a dedicated *Instruction* class that provides a specific *execute()* method defining its respective execution semantics. Each method that is loaded by JPF internally stores the associated code as an array of Instruction objects, which are created from the bytecodes read from the corresponding class file.

JPF instantiates such Instruction objects by means of an abstract *InstructionFactory*, i.e., it does not assume a particular implementation and imposes very few constraints about such execution semantics. The concrete InstructionFactory that implements the Java execution semantics can be overridden by JPF extensions. The JPF core distribution contains an implementation of the standard Java stack machine instruction set that operates on concrete values of operands, local variables and object fields. Alternative instruction sets can provide different execution semantics by means of overriding the execute() methods in their implementation of Instruction classes.

3.3 Attribute Objects

In addition to storing concrete values of operands, local variables, and fields according to the JVM specification, JPF provides a mechanism to attach extra information (metadata) to program data by means of *Attribute Objects* (see Figure 4).

Whenever a bytecode instruction transfers a concrete value between operands, local variables and fields, the associated attribute object reference is automatically copied by JPF, i.e. the attribute follows the value. If JPF backtracks to a previous state, attribute references are



Fig. 4 JPF Attributes.

restored. Because of these features, attributes are suitable for analyzing data flow and keeping track of information such as symbolic values – as in the case of SPF.

In addition, attribute objects can also be set and queried for JPF-specific constructs such as ChoiceGenerators and heap objects. All constructs that can have attributes provide a consistent API to set, add and query attributes based on their type, enabling coexistence of extensions that use this mechanism for different purposes.

3.4 Native Peers and the Model Java Interface

Native Peers are companion classes that can be used to execute certain methods of the system under test on the host Java Virtual Machine (JVM) instead of JPF. Each time JPF loads a system under test class, it uses the package and class name to locate a corresponding NativePeer. If such a class is found, JPF identifies potential peer methods by means of their modifiers and signatures.

In analogy to the standard Java Native Interface (JNI), the related JPF mechanism is called the *Model Java Interface* (MJI). In contrast to JNI, MJI allows interception of all methods, including constructors and static initialization. MJI is therefore a suitable mechanism for library abstraction. NativePeer classes are also the preferred way to handle *native*, platform specific Java methods such as file I/O, which cannot be directly executed by JPF and hence would result in UnsatisfiedLinkErrors at run-time.

When a respective method is called during system under test execution, JPF uses standard Java reflection to invoke the associated peer method, passing in an additional *MJIEnv* argument that allows the peer method code to access JPF internal data and functions.

3.5 Listeners

Listeners are JPF plugins that can monitor and control JPF's internal operations. Both the JVM and the Search object maintain their own sets of listeners and notify each listener instance of events such as instruction execution, thread context switches, object allocation, state backtracking, and many more. The JPF listener mechanism is an instance of the Publisher/Subscriber design pattern.

Listeners can query the program state and control JPF accordingly, for instance by registering ChoiceGenerators or detecting property violations. Consequently, listeners are often used to implement high level property checks or to set object attributes.

4 Symbolic PathFinder

Symbolic PathFinder (SPF) is an extension project in JPF and it is available as the project jpf-symbc from the JPF distribution [32]. SPF replaces the standard concrete bytecode interpretation of JPF-core with a non-standard symbolic interpretation using an instruction factory. The symbolic information is stored in *attributes* associated with program data and is propagated as needed during symbolic execution (see Section 3.3). Furthermore, SPF uses JPF to systematically generate and execute the symbolic execution tree of the code under analysis and also to handle multithreading. SPF uses a variety of off-the-shelf decision procedures and constraint solvers to solve the constraints generated by the symbolic execution of bytecode programs.

Choice generators are used to implement nondeterministic choices when symbolically executing branching conditions, and listeners are used for printing the results of the symbolic analysis. SPF also uses native peers for modeling native libraries. Recent work, [45], implements an alternative way for executing native libraries, without modeling them. We describe this technique in detail in Section 5. In this section, we describe the basic features of SPF and present several illustrative examples.

4.1 An Instruction Factory for Symbolic Execution of Bytecodes

As described in Section 3.2, JPF analyzes an input Java program (i.e. a set of class files) by interpreting the Java bytecodes in a customized Virtual Machine. It also provides a mechanism to extend execution semantics of the bytecode (recall Figure 3). JPF uses an *InstructionFactory* to instantiate its Instruction objects. In order to perform symbolic execution, SPF implements a SymbolicInstructionFactory that creates instances of instructions for the symbolic interpretation of Java bytecodes (see Figure 5). The new symbolic InstructionFactory. The symbolic instructionFactory inherit from JPF-core's DefaultInstructionFactory. The symbolic instructions classes conditionally add new functionality required for symbolic execution (e.g., in case the operands are symbolic), otherwise (e.g., in case the operands are concrete) they just delegate execution to their concrete super classes. This enables simultaneous concrete and symbolic execution modes.

As an example, JPF's standard execution of the *ifeq* bytecode pops a condition value from the operand stack and branches (or not) based on that value. The SPF implementation of the same bytecode ignores the condition value on the operand stack when it is symbolic, and instead uses a *ChoiceGenerator* to produce condition values that explore both branches. See Section 4.3 for more details and Section 4.9 for some longer examples.



Fig. 5 Symbolic Instruction Factory replacing Default Instruction Factory.

4.2 Symbolic Data Attributes

As described in Section 3.3, JPF can associate data attribute objects to storage entities such as values of operands, values of local variables, and values of fields. SPF uses these attributes to store the symbolic expressions computed during symbolic execution. The attributes are created or accessed in bytecode instructions via accessors (e.g., methods setAttr, getAttr). Attribute manipulation is mainly done inside of the JPF-core, within the various operations that modify and store the program states. Therefore, to implement symbolic execution, SPF only overrides instruction classes that create or modify symbolic information, such as instructions that perform numeric operations, compare-and-branch, and type conversion operations. Other bytecode instructions that simply retrieve or store values remained unchanged, since the attributes are propagated by JPF-core.

4.3 Handling Branching Conditions

The symbolic execution of conditional branch statements involves creating two choices in JPF. The two choices consist of adding the decision predicate in the branch statement and its negation respectively to the path condition in each choice. In order to generate these two choices, we implemented a new choice generator (PCChoiceGenerator) that branches the execution inside JPF. A path condition is associated with each choice generated by PCChoiceGenerator; the path condition is checked for satisfiability using a decision procedure or a constraint solver. If the path condition becomes un-satisfiable, JPF is automatically instructed to backtrack.

4.4 Multithreading, State matching, Loops and Recursion

Our framework performs an *inter-procedural* symbolic analysis by using JPF to systematically generate and explore the symbolic execution tree of the analyzed program. JPF is also used to systematically analyze all the possible thread interleavings and other forms of nondeterminism that might be present in the code. SPF takes advantage of JPF's built-in mechanisms for

combating state explosion, such as partial order and symmetry reductions. By default SPF is run with state matching turned "off". To limit the possibly infinite (symbolic) search state space that results from analyzing programs with loops or recursion, the user needs to provide a limit on the model checker's search depth or on the number of constraints encoded in the path condition.

When run with state matching "on", SPF explores an underapproximation of the program behavior, since, by default, the attributes used to store symbolic values do not participate in the matching. SPF also provides some support for abstract state matching, where only the shape of the program heap is used for matching. Note that in this case the symbolic state space explored by SPF forms a graph rather than a tree. Other kinds of abstract state matching can be written by over-riding JPF's default serialization mechanism. Extending SPF with full support for state matching would require subsumption checking between symbolic states [2]; this and more sophisticated handling of loops [43] is planned for future work.

4.5 Decision Procedures and Constraint Solvers

SPF uses multiple decision procedures and constraint solvers through a generic interface, which provides generic operations for building symbolic expressions and constraints and for solving them. Currently, SPF supports: the CVC3 [5] and Yices [75] Satisfiability Modulo Theories solvers, the Choco [66] and CORAL [63] solver for handling mixed integer-real constraints, and the interval arithmetic solver IASolver [30]. Adding support for additional constraint solvers such as HAMPI [35] and Z3 [42] is work in progress. SPF also implements constraint simplification and caching via a new recent extension (described in Section 7). While some of the existing decision procedures support incremental solving, SPF does not currently implement this feature.

4.6 Handling Input Data Structures

SPF uses lazy initialization [33] to handle unbounded input data structures. The execution starts on data structures with uninitialized fields and it initializes them lazily, when the fields are first accessed. A field of class T is initialized nondeterministically to (1) null, (2) a reference to a new instance of class T with uninitialized fields, or (3) a reference to an object of type T created during a prior field initialization; this systematically treats aliasing. The HeapChoiceGenerator is used to generate the choices. We have recently extended SPF to provide support for polymorphism. Step (2) above is replaced with nondeterministically assigning new instances of class T and of all the classes that inherit from T. Similarly, step (3) is replaced with assigning previously created objects to class T and objects from classes that inherit from T. Once the field has been initialized, the execution proceeds according to the concrete (non-symbolic) execution semantics. The model checker systematically handles the nondeterminism introduced when creating different heap configurations and when updating path conditions. SPF also offers support for fixed-size input arrays of primitive types; extended support for reference types is work in progress.

4.7 Handling Math Functions

SPF uses the MJI mechanism to model native libraries and other program parts that cannot be analyzed directly with symbolic execution. For an alternative technique see Section 5. SPF

```
public class IADD extends ....bytecode.IADD{
                                                public Instruction
                                                        execute (... ThreadInfo th){
                                             [1]
                                                   IntegerExpression svm v1. svm v2:
 public class IADD extends Instruction{
                                                   sym_v1 = ... .getOperandAttr(0);
                                             [2]
                                             [3]
                                                   sym_v2 = ... .getOperandAttr(1);
  public Instruction
                                             [4]
                                                   if (sym_v1 == null && sym_v2 == null)
          execute (... ThreadInfo th) {
                                                     // both values are concrete
[1]
       int v1 = th.pop();
                                             [5]
                                                      return super.execute(ss, ks, th);
       int v2 = th.pop();
[2]
                                             [6]
                                                   else {
[3]
       th.push(v1 + v2, ...);
                                             [7]
                                                     int v1 = th.pop();
[4]
                                                     int v2 = th.pop();
       return getNext(th);
                                             [8]
  }
 }
                                             [9]
                                                    th.push(0, ...); // ignore concrete val
                                             [10]
                                                     IntegerExpression result =
                                                       IntegerExpression._plus(sym_v1,sym_v2);
                                             [11]
                                                     ... .setOperandAttr(result);
                                             [12]
                                                     return getNext(th);
                                                  1 1 1
```

Fig. 6 Concrete (left) and symbolic (right) execution for the IADD bytecode.

also reuses JPF's large base of models for core libraries. Furthermore, SPF incorporates native peers to capture the calls to the java.lang.Math libraries and uses these calls to build complex mathematical constraints that can be handled by an appropriate constraint solver, such as CORAL. For example when Math.sin is called with a symbolic argument X, the method is not actually executed inside SPF, but rather a symbolic expression sin(X) is created, which can later appear in the symbolic expressions and path conditions built by the analysis. The same mechanism is also used for capturing String operations (see Section 6).

4.8 Symbolic Listeners

The listeners gather and display information about the path conditions generated during the symbolic execution. They generate test cases and test sequences in various user-defined formats. There are two listeners that display information about the execution:

- the SymbolicListener shows the computed path conditions and also the computed test cases test inputs and expected return (in text and HTML format), while
- the SymbolicSequenceListener shows test sequences (i.e. sequences of method calls), for each path condition encountered.

4.9 Examples

We illustrate symbolic execution of bytecodes with two examples. Let us first consider the IADD bytecode that performs addition of two integers. The code in Figure 6 (left) shows the default JPF class that implements the concrete interpretation of the bytecode: the first two values on the operand stack are popped (lines [1] and [2]), the two values are added, and the result of the addition is pushed back on the stack (line [3]). JPF is then instructed to execute the next bytecode (line [4]). Figure 6 (right) shows a simplified version of the code that implements semantics for the "symbolic" counterpart. The class IntegerExpression implements symbolic

```
public class IFGE extends ....bytecode.IFGE{
                                                 public Instruction
                                                         execute (... ThreadInfo th) {
                                                   IntegerExpression sym_v;
                                                  PCChoiceGenerator cg;
                                             [1]
                                                  sym_v = ... .getOperandAttr();
                                             [2]
                                                  if(sym_v == null)
                                             [3]
                                                     // the condition is concrete
                                             [4]
                                                    return super.execute(... th):
 public class IFGE extends Instruction{
                                             [5]
                                                  else {
                                                     // the condition is symbolic
  public Instruction
                                                    cg = new PCChoiceGenerator(2);
          execute (... ThreadInfo th) {
                                             [6]
[1]
       condition = (th.pop() >= 0);
                                                     . . .
[2]
       if (condition)
                                             [7]
                                                     condition=cg.getNextChoice()==0?false:true;
[3]
         next=getTarget();
                                             [8]
                                                     th.pop();
[4]
       else
                                             [9]
                                                    if (condition) {
[5]
         next=getNext(th);
                                             [10]
                                                        pc._add_GE(sym_v, 0);
                                                        next = getTarget();
[6]
                                             [11]
       return next;
  }
                                                     }
 }
                                             [12]
                                                     else {
                                             [13]
                                                        pc._add_LT(sym_v, 0);
                                             [14]
                                                        next = getNext(th);
                                             [15]
                                                     if(!pc.isSatisfiable())
                                             [16]
                                                        ... // instruct JPF to backtrack
                                             [17]
                                                     else
                                                        cg.setCurrentPC(pc);
                                             [18]
                                             [19]
                                                     return next;
                                               } } }
```

Fig. 7 Concrete (left) and symbolic (right) execution for the IFGE bytecode.

integer expressions; a similar class, RealExpression, implements symbolic real expressions. SPF path conditions are built from these expression classes (among others).

Recall that the symbolic information is propagated via attributes. The *execute* method first checks if the attributes associated with the two operands are null. Null values for the operands indicate that the operands are concrete and the execution follows according to standard execution semantics (line [5]). Otherwise, if at least one of the operands is symbolic, then the result also becomes symbolic, and this is recorded in the result attribute that is pushed on the stack. Method _plus builds a new symbolic expression that represents the addition of its parameters. The attribute of the result is set to this new symbolic expression (line [11]). Since the result becomes symbolic, its concrete value does not matter as by default we use no state matching, so we set it to 0 (line [9]).

We illustrate the use of choice generators in the symbolic execution of branching conditions with the IFGE bytecode. The code in Figure 7 (left) shows the concrete interpretation of the bytecode: the first popped value from the stack is compared with 0 to compute the associated condition. This condition determines the next instruction to be executed.

In symbolic execution (Figure 7 (right)), the concrete condition is no longer used to exclusively choose between program branches. Instead we create a choice generator (line [6]) that introduces a nondeterministic choice (line [7]) that allows both execution branches to be considered. For each branch, the path condition is updated with the symbolic condition (line [10]) and its negation (line [13]) respectively. The isSatisfiable method uses a decision procedure to check if the path condition is satisfiable or not. In the latter case, JPF backtracks (line [16]).



Fig. 8 Screenshot of an application file (.jpf) in SPF used to execute symbolic execution on a Java program within the Eclipse IDE.

4.10 Experience

The Java Pathfinder toolkit sources are Mercurial repositories within the

http://babelfish.arc.nasa.gov/hg/jpf directory that provides anonymous public read access. Running SPF requires getting the jpf-core and jpf-symbc projects from the repository at the above address. The project jpf-core is a pure Java 6 application with no specific platform requirements, while jpf-symbc is written in Java but needs different off-the-shelf constraint solvers, that may have different platform requirements. Where possible we provide binaries of the constraint solvers for some of the commonly used platforms.

JPF-core and SPF can be run from within an IDE (integrated development environment), such as Eclipse or NetBeans. Figure 8 shows a screenshot of a JPF Eclipse plugin being used to run an application configuration file (.jpf). For the example in Figure 8, the configuration file contains the class name of the Java program which is being analyzed (target), the path to the directory that contains the compiled class files of the Java program (classpath), the path that contains the sources of the Java program (sourcepath), the name of the method to be symbolically executed (symbolic.method), the parameters in the symbolic method that are to be executed symbolically (symbolic.method(sym#sym)), the listener to gather infor-

| | Error Type | LOC | States | Time (sec) | Memory (MB) |
|--------------|------------|------|--------|------------|-------------|
| VecDeadlock0 | Deadlock | 7267 | 1370 | 4.56 | 66 |
| VecDeadlock1 | Deadlock | 7169 | 2948 | 6.89 | 69 |
| VecRace | Race | 7151 | 3120 | 7.98 | 65 |

Table 1 SPF Results for Finding Concurrency Errors

mation about the path conditions and test cases generated (listener), and finally turning off state matching (vm.storage.class). The application configuration file can also be run from the command line or using a NetBean plugin.

SPF has been applied at NASA in various projects including test case generation for the Orion control software [44], fault tolerant protocols, aviation software, and robot executives. SPF has been extended by Fujitsu, where it is being used for testing web applications. MIT's tool JFuzz [31], a concolic white-box fuzzer for Java, is built on top of SPF and is freely available from the JPF web site. See Section 7 for a description of other analyses and projects that are built on top of SPF.

SPF can analyze both Java bytecode and statechart models, e.g., Simulink/Stateflow, Standard UML, and Rhapsody UML. The statechart models are automatically translated into Java bytecode using the Polyglot tool [3]. We are currently using SPF for the analysis of modelbased NASA software composed of multiple interactive components; SPF has generated test sequences that uncovered problems in the interaction between Ares and Orion models [4]. We have also used the tool for the analysis and testing of the JPL MER Arbiter [3].

We present here some recent results on running SPF for finding concurrency errors with guided heuristics [52] and for generating test sequences for the examples from [65]. In addition we show some results for running SPF with increasing analysis depth, which is a typical scenario for using the tool.

Table 1 shows the number of states explored and the resources consumed (time and memory) for detecting two deadlocks and a race-condition in the **Vector** class in the JDK 1.4 library, using a total of three threads. These results demonstrate SPF's capability of analyzing small multithreaded programs with respect to deadlocks and data races. Other properties, such as the absence of assert violations or program specifications written as automata monitors and temporal logic (supported in some experimental JPF extensions), are also handled.

Table 2 gives the results of running SPF for six examples [65]. The first two systems, an Altitude Switch (ASW) and a Wheel Brake system (WBS) are two synchronous reactive systems from the avionics and automotive domains, respectively. They were both developed in Simulink and later translated to Java. The other four Java data structures are freely available with the JPF distribution and they are commonly used in conjunction with JPF. In our experiments, we performed analysis and test generation for these data structures by exploring all possible sequences of input operations up to a finite length, as specified in the Java test driver that we built for each example. We put a limit on sequence length as shown in the table; this number was chosen to allow SPF to run for a reasonably long but still feasible analysis time, and to complete using 1 GB of memory. Random depth first search was used as a search strategy. The results (as shown in the table) indicate that the time for performing symbolic execution increases quickly with the sequence size. A parallel version of SPF [65] attains close to linear speedup for the same set of examples.

Finally, Table 3 displays the results for running SPF with iterative deepening for three examples, including two larger ones. Mer Arbiter models a component of the flight software for NASA JPLs Mars Exploration Rovers (MER). The analyzed software consists of a Resource Arbiter and several user components. Each user serves one specific application, such

| | System | Classes | LOC | Seq. Length | Time (min:sec) |
|---|---------|---------|-----|-------------|----------------|
| ĺ | ASW | 15 | 425 | 2 | 7:57 |
| ĺ | WBS | 1 | 231 | 5 | 10:22 |
| ĺ | BinHeap | 2 | 268 | 6 | 9:26 |
| Ì | BinTree | 2 | 115 | 6 | 117:34 |
| Í | FibHeap | 2 | 258 | 7 | 47:16 |
| ĺ | TreeMap | 2 | 447 | 7 | 47:33 |

Table 2 SPF Results for Test Sequence Generation

| System | Classes | LOC | Depth | States | Time (sec) | Memory (MB) |
|-------------|---------|-------|-------|--------|------------|-------------|
| WBS | 1 | 231 | 25 | 349272 | 20 | 242 |
| | | | 30 | 644184 | 37 | 214 |
| Mer Arbiter | 268 | 4.6 K | 25 | 17103 | 47 | 413 |
| | | | 30 | 33273 | 92 | 413 |
| | | | 35 | 35359 | 102 | 292 |
| Apollo | 54 | 2.6 K | 10 | 674 | 195 | 421 |
| | | | 12 | 2243 | 1033 | 390 |

Table 3 SPF with Iterative Deepening

as imaging, controlling the robot arm, communicating with earth, and driving. The arbiter module moderates access to several shared resources. It prevents potential conflicts between resource requests coming from different users and it enforces priorities. Mer Arbiter has been modeled in Simulink/Stateflow and it was automatically translated into Java using the Polyglot framework [3]. The configuration for our analysis involved two users and five resources. The example has 268 classes, 553 methods, 4697 lines of code (including the Java Polyglot execution framework).

The Apollo Lunar Autopilot is a Simulink model that was automatically translated to Java, resulting in 2.6 KLOC in 54 classes. The model is available from MathWorks. It contains both Simulink blocks and Stateflow diagrams and makes use of complex mathematical functions (e.g. Math.sqrt). The code has been analyzed using Symbolic PathFinder together with the Coral solver for solving complex, non-linear constraints (see Section 7).

In the experiments reported here SPF was run with two different search depths, as specified in the JPF configuration file. The experiments mimic a typical use of symbolic execution tools such as SPF, which are usually run in an iterative deepening mode, to achieve the desired code coverage or to discover the errors that might be present in the code. We note that the results from previous iterations could in principle be reused at the current depth (we do not report on it here). For some recent work on reusing the information during iterative deepening and other incremental analyses in the context of SPF see [74].

Table 3 shows the time and memory consumed when running SPF. We note that, as expected, the time cost is larger for increased depths. However the memory cost for symbolic execution at the larger depth can be smaller. To understand this observation we ran SPF several times with the same configuration and we found that the reported memory cost varies a lot. We conjectured that this depends on how the underlying garbage collection works and that comparison of the memory cost (as shown in the table) is not very meaningful. On the other hand, in terms of space cost, a comparison based on number of states seems to make more sense. Indeed a larger number of states is explored with increasing depth, as expected.

4.11 Discussion

In this section we have provided an overview of Symbolic PathFinder. The tool uses a combination of symbolic execution, model checking and constraint solving to perform a systematic analysis of Java programs with unspecified inputs, up to a prescribed search depth. The tool handles inputs of reference and primitive types. Furthermore, the tool relies on JPF for the efficient analysis of multithreading. SPF uses models and also more sophisticated techniques (see next sections) for handling external and complex mathematical libraries. The tool can be used for automated test case generation and for the detection of software errors (assert violations, run-time errors, races, deadlocks, etc.). The tool currently provides only partial support for input arrays and a rather simple treatment of loops. Addressing these limitations together with other on-going challenges to symbolic execution, such as path explosion in the presence of concurrency and adding even more support for complex mathematical calculations and external function calls are the subject of current and future work.

The most important engineering decision was to build SPF tightly integrated with JPFcore. This enabled us to take advantage of the large code base in the core for the efficient systematic analysis of multithreaded Java bytecode. The symbolic information is stored in attributes associated with, but separated from, the program data, enabling a mixed symbolicconcrete execution mode. We note that in our previous tool, JPF–SE [1], symbolic execution was performed by program instrumentation, which incurred significant interpretation overhead, as compared with the attribute based implementation, and was not fully automated. Finally, we also mentioned that we have a generic interface for integrating different constraint solvers. This interface is designed to take advantage of the plethora of solvers that are becoming available, and it facilitates easy extension with new solvers.

5 Mixed Concrete-Symbolic Solving

Symbolic execution may fail due to inherent incompleteness in decision procedures and its inability to handle external library calls. So far, we have discussed how SPF addresses theses limitations by using the MJI mechanism to model external libraries or to capture the calls to Mathematical functions to build complex constraints that can be solved with Choco or CORAL (see Section 7 for a description of heuristic solving with CORAL). In this section we describe an alternative approach that does not rely on MJI and addresses limitations arising from the incomplete nature of decision procedures and native calls; for additional details see [45].

The approach performs satisfiability checking with mixed concrete-symbolic solving and it applies to "pure" (side-effect free) external function calls. Similar to dynamic symbolic execution techniques, such as DART [25], we use concrete values to simplify constraints that can not be handled by the decision procedure directly. However, unlike DART, we do not use the run-time values of program variables, but instead we use the concrete solutions of the solvable constraints in the current path condition. As a result our technique can be more powerful than DART, i.e. it can obtain full path coverage where DART fails (e.g. on the example in the next section). We note however that, in fact the two techniques are incomparable in power, and provide more discussion on DART and related tools in Section 8. The mixed concrete-symbolic solving consists of two main ingredients:

- use of *uninterpreted functions* to represent calls to unknown or complex functions during symbolic execution and
- the *delayed execution* of these functions, based on the values obtained by solving the "simple", solvable constraints in the path condition (*simplePC*).



Fig. 9 Example illustrating mixed concrete-symbolic solving.

The results returned from the delayed execution of the external functions are used to build a new set of simplified constraints that are solved again, using an off-the-shelf solver, hence the name "mixed concrete-symbolic solving". If the new constraints are satisfiable, then the original PC is also satisfiable, and the symbolic execution continues on that path; otherwise it backtracks. Note that in the latter case, it may happen that the original constraints are indeed satisfiable, but our method failed to detect that. In other words, our method is incomplete, similar to related approaches [11, 25].

We address the incompleteness through three heuristics that attempt to "force" the solver to generate more solutions from simplePC and increase the chances of finding a solution for the simplified complex part of PC.

- The first heuristic uses incremental solving to generate multiple solutions for *simplePC*. These solutions are then used to execute the external functions and perform iterative mixed concrete-symbolic solving (up to a user specified limit).
- The second heuristic leverages user annotations that partition the domains of the uninterpreted functions into subsets that are deemed interesting by the user. Such partitions can be simple abstractions (e.g. partition the inputs between positive and negative values) or they can come from some form of black-box analysis of the external functions. The extra constraints are systematically added to the current PC and mixed concrete-symbolic solving is called for each newly obtained PC.
- The third heuristic simply uses random values for all the inputs that are unconstrained in the path condition.

5.1 Example

Figure 9 (left) shows a simple program taken from [45] using Java-like syntax. We analyze method test that invokes another method hash. Assume that hash is a complex mathematical function that our off-the-shelf constraint solver can not handle or a function whose code is simply unavailable for analysis (e.g. *native* method in Java). For illustrative purposes, let us assume that hash(x) = 10 * x, for $0 \le x \le 10$ and 0 otherwise. "S0", "S1", "S3" and "S4" denote statements that we wish to cover with our automated testing techniques (their exact content does not matter here).

```
[1] void test(int x, int y) {
[2] if(x>=0 && x>y && y==x*x)
[3] S0;
[4] else
[5] S1;
[6] }
```

Fig. 10 Example illustrating potential for unsoundness.

The path condition PC to reach the execution of "S0" at line #4 has the value X>0 & Y=hash(X), where X and Y denote the two input symbolic values for method test. We use upper case letters to denote the symbolic representation of the equivalent variables defined in lower case letters. Normally symbolic execution would get stuck given our assumption that the constraint solver cannot handle hash directly so it cannot generate two values for the inputs x and y that drive the execution of test through the then branch of the conditional at line #3. For example, in Figure 9(right), in order to execute the path corresponding to "S0, S3", we need to decide if PC: X>0 & Y=hash(X) is satisfiable.

With our technique, we first split *PC* into *simplePC*: X>0 and *complexPC*: Y=hash(X) and solve *simplePC*. The obtained solution X=1 is used to compute hash(1)=10 and to simplify *complexPC* into Y=10. This constraint is conjoined back with *simplePC* and the result is *newPC*: X>0 & Y=10 which is satisfiable (more constraints are added to ensure soundness as discussed in the next section).

Similarly, the second *PC* along the path "S0, S3" is *PC*: X>0 & X>3 & Y>10 & Y=hash(X), which is equivalent to *PC*: X>3 & Y>10 & Y=hash(X). Our mixed concrete-symbolic solving technique will first solve *simplePC*: X>3 & Y>10, it will use solution X=4 to compute hash(4)=40 and will then solve again the simplified constraints *newPC*: X>3 & Y>10 & Y=40. Our mixed solving approach can cover all the paths through the code.

Assume now that in the code of Figure 9(left), we replace line #6 with line #7. Then it becomes harder to cover the paths through test. Our technique will not be able to cover that path, since it does not have enough information in *simplePC* to decide the satisfiability of the overall *PC*. To overcome the problem, we developed some simple heuristics that allow us to cover "S0, S3" as well. The first heuristic uses iterative solving to generate multiple solutions for the *simplePC*: X>0 & Y>10 and then uses these solutions to repeatedly concretize hash and to perform mixed concrete-symbolic solving. Thus, after running for two iterations, solution X=2 is found and this is good enough for making X>0 & Y>10 & Y=hash(X) true, since Y=hash(2)=20 and X>0 & Y>10 & Y=20 is satisfiable.

The second heuristic uses extra constraints provided by the user via a @Partition annotation to help finding solutions (see Figure 9(left) that defines two partitions on the input: X>3 and X<=3). For the example, applying the heuristic results in adding the constraints describing the partitions to *PC*. As a result, we obtain two new path conditions: *PC* & X>3 and *PC* & X<=3 and we apply mixed concrete-symbolic solving for each resulting path condition. The new constraint *PC* & X>3 is equivalent to X>0 & Y=hash(X) & Y>10 & X>3, for which we can find a solution so we stop.

5.2 Potential for Unsoundness

The example in Figure 10 illustrates the need for additional equality constraints to ensure the soundness of our method. Soundness here means that path conditions reported to be satisfiable should indeed have at least one solution. The path condition corresponding to the execution

18

of "S0" is X>=0 & X>Y & Y=X*X. Note however that "S0" is unreachable, i.e. the path condition is not satisfiable. If we split this path condition into the part we can solve, namely *simplePC*: X>=0 & X>Y, and the part we cannot solve¹, namely *complexPC*: Y=X*X, then we can obtain a result for the first part that suggests we should use X=0 in the non-linear part. This will simplify the non-linear side to Y=0. In turn this will lead to a simplified combined constraint of X>=0 & X>Y & Y=0 which is satisfiable with X=1 and Y=0; which could lead us to believe that S0 is reachable. The problem is that we introduced unsoundness when we ignored the solutions obtained from the solvable side when using the simplified result from the non-linear side.

We therefore always add extra equality constraints on the solutions that we use to simplify complexPC. These extra constraints are stored in extraPC. For the example, extraPC is X=0 and the final constraint becomes X>=0 & X>Y & Y=0 & X=0 which is not satisfiable. The constraint on the solution for X, namely X=0, that we used to simplify Y=X*X is thus added back into the final constraint.

5.3 Implementation

We have implemented the mixed concrete-symbolic solving as additional procedures for checking satisfiability of path conditions within SFP. The input required from the user is in the form of annotations (@Concrete("true")), that specify which methods to leave uninterpreted. Another annotation (@Partition(..)) provides a list of conditions that is used in the partitioning heuristic (as discussed in the example above).

We use Java reflection to perform the actual invocation of the methods during the delayed execution and simplification of the generated PCs. Using reflection we are able to concretely execute the external functions on the host VM, without providing a model class using MJI. Note that our approach works only for the external methods that are pure, side-effect free, or for methods whose side effects are deemed uninteresting for the analysis (e.g. printing statements). All the other methods still need models. Using this technique we were able to analyze software that could not be analyzed before with "classical symbolic execution", such as components from Tactical Separation Assisted Flight Environment and Apollo Lunar autopilot (see [45] for details on the heuristics and the experiments).

6 Symbolic String Analysis

The widespread use of web-based interfaces have resulted in a greater need for the validation of various string inputs to ensure better security. However, classic testing approaches, such as guided black-box and random testing are not capable of reliably detecting malicious inputs, simply because the domain of string inputs is too large. For this reason, SPF has been extended to enable symbolic reasoning over strings.

Consider function site_exec in Figure 11: its purpose is to receive and execute remote commands and if the command extracted from the input contains the substring "n", a run-time exception is thrown. This example, although very simple, illustrates a typical code injection scenario; the example is taken from [29] and is based on a real error [13]. An input string s1 triggers the run-time exception if it satisfies the following constraints (s2 and i2 are auxiliary variables):

¹ Assuming here our solver cannot deal with non-linear integer arithmetic

```
public static void site_exec(String cmd) {
1
        String result, path = "/home/ftp/bin";
\mathbf{2}
3
        int j, sp = cmd.indexOf(' ');
4
        if (sp == -1) {
\mathbf{5}
           j = cmd.lastIndexOf('/');
6
           result = cmd.substring(j);
7
        } else {
8
           j = cmd.lastIndexOf('/', sp);
9
           result = cmd.substring(j);
10
        3
11
        if (result.length() + path.length() > 32) {
12
            // buffer overflow
13
           return:
14
        3
        String buf = path + result;
15
        if (buf.contains("%n")) {
16
            throw new RuntimeException("THREAT");
17
18
        }
19
     }
```

Fig. 11 Example of code injection

s1.indexOf(' ') = i1 ∧ s1.lastIndexOf('/') ≥ 0 ∧ s1.lastIndexOf('/') = i2 ∧ s1.substring(i2) = s2 ∧ s2.length() < 19 ∧ s2.contains("%n")

We refer to the last constraint as a (pure) string constraint, because it involves only string variables and constants. The second to last constraint, on the other hand, is a (pure) integer constraint, since s2.length is in essence an integer variable and 19 is an integer constant (19 comes from 32 - path.length()). The first three constraints are mixed (integer and string) constraints. This classification is important, because typically different off-the-shelf decision procedures and constraint solvers are required to handle string vs. numeric constraints. In fact many of the current solutions to symbolic execution for strings either consider a limited set of string-integer interactions [14, 56, 60, 67] or none at all [35, 28].

6.1 String Graphs

To handle constraints from mixed numeric and string domains, we construct a constraint hypergraph [51, pp. 211–212] which we refer to as a *string graph*. This usually requires the introduction of auxiliary variables (such as s2 above). Specifically, an integer variable is introduced to represent the length of each of the string variables in the string graph.

Figure 12 shows the string graph for the six constraints discussed above. Variables are shown as round vertices, constant values as square vertices, and hyperedges as lines that meet at a black dot. Because the string graph is directed, the vertices connected by a hyperedge are numbered. For example, the hyperedge in the center of the figure is labeled "substring" and connects vertices "s1", "i2", and "s2", in that order; it corresponds to the constraint s1.substring(i2) = s2. The dashed lines are not edges: they merely connect a string variable with the integer variable that represents its length. In other string graphs, the lengths may, however, participate directly in constraints.

Note that the constraint s2.length() < 19 does not appear in the string graph. Instead, n2 < 19 is added to the set of numeric constraints. Other numeric constraints are also gen-



Fig. 12 Example of a string graph

```
SOLVE(PathCondition pc)
1 StringGraph sg
 2 (pc, sg) := BUILDSTRINGGRAPH(pc)
3 boolean sat := false
 4 while \neg sat \land \neg timeout:
5
      (sat, pc, sg) := \text{NUMERICSOLVER}(pc, sg) \{ \text{Phase I} \}
6
      if sat: (sat, pc, sg) := \text{STRINGSOLVER}(pc, sg)  { Phase II }
 7 return sat
BUILDSTRINGGRAPH(PathCondition pc)
8 StringGraph sg := \emptyset
9 for string or mixed constraint c \in pc:
10
      sg := sg \cup \text{HYPEREDGE}(c)
11 return PREPROCESS(pc, sg)
```

Fig. 13 String constraint solving algorithm

erated by the string graph. For example, s2.contains("%n") triggers the introduction of the constraint $n2 \ge 2$. Simple heuristics are applied to the string graph during a preprocessing stage. The preprocessor can often lead to the early (and cheap) conclusion that the constraints are unsatisfiable. For example, given the constraints s1.equals(s2), the preprocessor would merge the vertices for s1 and s2, and merge all their edges accordingly. Or, given constraints such as s3.equals(s4), s4.equals(s5), and !s5.equals(s3), the preprocessor can conclude immediately that the constraints are unsatisfiable.

The string graph allows us to easily split the PC into: (a) purely numeric constraints, and (b) string and mixed string-numeric constraints. We then use an iterative two-phase approach: In phase I the numeric constraints are solved and the candidate solutions are temporarily placed in the string graph. In phase II we use one of two techniques (based on either bitvectors or automata; discussed below) to solve the string and mixed constraints. If they are satisfiable, solutions are generated for the string and numeric variables and symbolic execution can then proceed to the next instruction. Otherwise, new numeric constraints are generated based on the unsatisfiable constraints and we return to phase I. The algorithm is outlined in Figure 13.

6.2 The Bitvector-based Approach

After the completion of phase I, candidate lengths for all of the string variables are known. This makes a straightforward translation to bitvectors possible, where each character is represented by 8 (in the case of ASCII) or 16–32 (in the case of Unicode) bits. Verbosity is still a problem: a constraint such as s1.contains(s2) and the candidate integer solutions n1 = 5 and n2 = 2, produce the following bitvector constraints:

| $s1[0]=s2[0] \land s1[1]=s2[1]$ |
|---------------------------------|
| $s1[1]=s2[0] \land s1[2]=s2[1]$ |
| $s1[2]=s2[0] \land s1[3]=s2[1]$ |
| $s1[3]=s2[0] \land s1[4]=s2[1]$ |
| |

We use Z3 [42] for satisfiability checking, but it is a simple step to adjust the translation for other SMT solvers that are capable of deciding bitvector constraints.

6.3 The Automata-based Approach

In the automata approach, we leverage the automata package of the Java String Analyzer (JSA) [14]. Each string variable is represented by a separate finite state automaton, which initially accepts the universal language (the regular expression " \cdot *"). Once again, iteration is used: each of the string and mixed constraints is taken in turn, and the automata are adjusted to comply with the constraint. For instance, if $M_1 = (\cdot*)$ and $M_2 = (\cdot*)$ are the automata for string variables s1 and s2, respectively, the constraint s1.startsWith("abc") will effect the change $M_1 = (abc \cdot *)$, and the constraint s2.endsWith(s1) will effect the change $M_2 = (\cdot*abc \cdot *)$. After one pass over the constraints is completed, each automaton accepts the same language or a smaller language. This guarantees that, after a finite number of such passes, the automata converge to a fixed value. At this point, all of the automata satisfy all of the constraints, and unless one or more automata are empty, the automata describe the solutions to the set of constraints.

There is one important caveat to this seemingly simple approach: it is impossible to encode inequalities such as !s1.equals(s2) in this way. They are therefore postponed until the automata have converged, when they are handled by a special post-processing phase, that operates much like a SAT solver. More details about this can be found in Gideon Redelinghuys's thesis [49].

6.4 Implementation and Evaluation

We extended SPF to accommodate string operations as part of path conditions, added code to manipulate string graphs, added translations from string constraints to Z3, and implemented our automata-based solver, which is based on Java String Analyzer (JSA) [14]. Although this represents a significant amount of work, the new code fits smoothly into the existing architecture of SPF, and the new functionality is transparently available to users.

The largest example we analyzed with the system is a piece of industrial code from the mobile communications domain which actually caused a system with millions of users to go down for hours. The problem occurs when sanitizing input strings in HTML format. Essentially, a very unlikely sequence of characters forces the program to go into an infinite loop that allocates new objects on every iteration, thus exhausting available memory and crashing the JVM. In the field this error crashed one server after another as a user issued the same service

request to the system, over and over. We took the code "as-is" without any modifications and just inserted a check for when the infinite loop was entered. In a matter of seconds the tool found an input sequence that causes the error. The code we analyzed was 311 lines and it required 109 iterations between integer and string solving (using bitvectors) to find the error.

We also did an extensive analysis on using both back-ends, and interestingly found that although there are differences, in the end other factors play a much bigger role than the back-end engine. For example, better pre-processing or more precise constraints being derived during string analysis to narrow the search in the integer space, have a bigger influence than the back-end system being used (again see [49] for more details).

7 Extensions and Applications

As a testament to the usability and extensibility of SPF we describe here several recent interesting analyses that have been integrated with SPF. Other approaches that build upon SPF, such as parallel symbolic execution [65], load testing [78], memoised symbolic execution [74] and concolic testing [31] are described elsewhere (see references).

7.1 Extensions

7.1.1 CORAL: Meta-heuristic Solving for Complex Mathematical Constraints

As we already described, the application of symbolic execution to automatic test generation involves two steps: first, generating the constraints on the input variables that must hold for particular paths to be executed (the path conditions); second, solving the constraints to give particular test vectors involving concrete values for the input variables.

For constraints on integer and boolean variables there are many implementations of capable constraint solvers. But in some domains the path constraints often include floating point variables and complex mathematical functions (for example, trigonometric and other transcendental functions). These types of constraints are especially common in cyber-physical systems, such as those that arise in aerospace, automotive, and medical systems. The choices for solvers (especially solvers still under active development) are generally more limited and the solvers themselves are unable to solve many of the more complex constraints that occur in the domains' software.

For these reasons SPF has recently been integrated with the CORAL meta-heuristic solver [63]. CORAL is a "meta" constraint solver – it sits on top of one or more traditional constraint solvers and applies search heuristics in order to solve more complex constraints than the underlying constraint solvers can solve by themselves. In its current version, CORAL uses particle swarm optimization to guide the underlying constraint solvers in generating successive families of solution candidates. Experience has shown that these candidates will often converge to constraint solutions even when the underlying solvers have difficulty. See [63] and [7] for more details on CORAL's operation.

CORAL has been integrated with SPF through SPF's generic decision procedure interface. SPF has been designed to make this a relatively simple operation – it only requires the creation of wrapper classes that translate between SPF's constraint and solution representations and CORAL's. SPF then allows CORAL to be specified as the desired constraint solver through a run-time configuration file.

7.2 Architecture For Reuse

Calls to a decision procedure or constraint solver are expensive, and it makes sense to avoid them whenever possible. One obvious solution is to reuse previously calculated results, and this is commonly done within one run of a symbolic analysis [10]. But this basic idea can be generalized to great effect. Reuse can occur between different analysis of the same program, different analysis of different programs, different runs of different systems, and between different users. For this idea to work effectively, it is important that queries and their results are standardized. Moreover, reuse occurs at two levels: not only are previous operations stored and reused, but the sometimes complicated code to manipulate the queries is centralized to be shared by systems.

At the core of this approach is the Green tool [72]. It is independent of JPF/SPF to make it available for integration in many different systems. It allows clients to register components, such as decision procedures, constraint solvers, and model counters. After this configuration, clients construct problem instances by building expressions akin to SPF's path conditions. Internally, Green tool slices the expressions to remove redundant information, canonizes the expressions to obtain a normal form (when possible), and manages a database that caches the result of queries. The components that transform the expressions and communicate with the database are also configurable.

At the moment Green provides a default slicer and canonizer that are targeted to linear integer arithmetic constraints. A default store is provided for caching results in a network of Redis servers [50, 53], and there is support for CVC3 [5] and Choco [66] as both decision procedures and constraint solvers, and for LattE [24, 40, 37] as a model counter. Work is underway to support a wider range of decision procedures and to extend the slicer and canonizer for more general constraints. It is easy to see how this framework can be extended to support mixed constraints (as described in Section 5), string constraints (Section 6), and even features such as concurrent distributed constraint solving.

If a system does not need a feature, the component can be omitted. For symbolic execution using CVC3 for example, only the translator from SPF path conditions to Green expressions needed to be custom written. Modifying SPF to use Green instead of its traditional decision procedure infrastructure, took 16 hours of work. Similarly, to integrate it into a Dynamic Symbolic Execution system took 1.5 hours. In both cases the integration was done by people other than the main developers of the respective tools. We believe this illustrates that the integration is quite manageable and for tool developers it might be even easier.

7.3 Applications

7.3.1 Directed Incremental Symbolic Execution

The goal of Directed Incremental Symbolic Execution (DiSE) is to leverage symbolic execution and static analyses in synergy to enable more efficient symbolic execution of programs as they evolve [46]. The path conditions computed by DiSE characterize the differences in program behaviors between two related program versions. The program instructions whose execution may lead to the generation of affected path conditions are termed as "affected locations" or "affected instructions". Standard static analysis techniques using control and data dependencies are used to identify program instructions in the source code that define program variables relevant to *changes* in the program. These instructions are marked as affected. Conditional branch instructions that use those variables, or are themselves affected by the changes, are also identified as affected. The information generated by the static analysis is used to *direct* symbolic execution to explore only the parts of the programs affected by the changes, potentially avoiding a large number of unaffected execution paths. DiSE generates, as output, path conditions only on conditional branch instructions that use variables affected by the change or are otherwise affected by the changes.

DiSE is implemented on top of SPF. For the examples used in our case-study, DiSE consistently explores fewer states and takes less time to generate fewer path conditions compared to standard symbolic execution when the changes affect only a subset of the program execution paths. This demonstrates the effectiveness of DiSE in terms of reducing the cost of symbolic execution of evolving software. Furthermore, we apply the results of DiSE to test case selection and augmentation to demonstrate the utility of such an analysis. The DiSE extension is available through a mercurial repository: http://babelfish.arc.nasa.gov/hg/jpf/jpf-regression

7.3.2 JPF Continuity

JPF Continuity [9] is another application of SPF to cyber-physical systems. In this extension, SPF is applied to the problem of analyzing continuity and robustness properties of floating point software. This is an area which has begun to receive more attention as high-reliability cyber-physical systems become more common and their software becomes more complex.

The question addressed here is, "how smoothly does the software's behavior change when its inputs vary?" Many of these cyber-physical systems either require their behavior to be smooth or to have their discontinuities be well understood. The JPF Continuity extension to SPF seeks to identify those regions of the input space which are discontinuous because of the software's control structure.

JPF Continuity builds on SPF by analyzing both pairs of path conditions (instead of single path conditions) and their associated pairs of computations (called "path functions"). It first identifies so-called "boundary constraints" on the inputs which cause the software to switch from one execution path to another. It then analyzes the functions computed along each path to determine which boundary inputs give rise to discontinuous or non-robust behavior.

7.3.3 Probabilistic Symbolic Execution

Symbolic execution typically only uses the fact that a path is either feasible or infeasible. However, one can also consider that a path condition will be satisfied with a certain probability, i.e. add meaning to the values between 0 (infeasible) and 1 (all probability values besides 0indicate feasible behavior). The probabilistic symbolic execution (probsym) extension to SPF allows the calculation of path probabilities [24]. The approach involves counting the number of solutions to a path condition using LattE [37] under the assumption that the input values are uniformly distributed in given finite input domains. The extension is implemented as a Listener that is triggered whenever a branch is found to be feasible during symbolic execution. When triggered it conceptually calculates the probability by dividing the number of solutions to the current path condition over the size of the input domain of all variables. In practice however the complete path condition can be very large and thus one first *slices* the path condition to only obtain the part that is used to determine if the current condition is feasible. However, that then also means one can only calculate *conditional* probabilities that just state what the odds are of taking the current branch (without considering the previous branches). One must calculate the complete path probability by multiplying all conditional probabilities along the path. Slicing the path condition also reduces the cost of symbolic execution (since it means a smaller path condition needs to be checked for feasibility) and therefore is also part of the Solver architecture discussed in Section 7.2.

In [24] the probabilities are used to show how errors can be found by using the notion of the *least likely* paths through the code, how the chances of obtaining coverage can sometimes decrease and sometimes increase when input ranges are varied and lastly, how one can use the probabilities for fault localization.

We are currently investigating a related approach that performs reliability analysis for a Java program [23], based on a *usage profile* which specifies a probability distribution on the program inputs. The approach handles both concurrency and input data structures and it further reports a confidence measure for the results. Furthermore, while the approach in [24] computes probabilities "on-the-fly", the reliability analysis is decoupled from the symbolic execution.

8 Related Work

Our work is related to the large body of work on white-box test-case generation and static analysis, but we focus here on the more closely related works. Symbolic execution was introduced in the 70s [15, 36] and it has since been explored in many ways [8, 16, 20, 21, 22, 33, 54, 44, 62, 68, 71, 73, 61, 55, 38] in the context of test case generation and software error detection.

The Extended Static Checker (ESC) [22] uses a static analysis to verify partial correctness of Java classes. Although our focus here was on test case generation and error detection, we can also use SPF to check light-weight properties in a way similar to ESC.

Symstra [73] uses a specialized symbolic execution over numeric data to generate test sequences for (sequential) Java containers. We have provided here a general framework for the symbolic execution of arbitrary Java bytecode. Symclat is an experimental implementation of symbolic execution in JPF that was developed in the context of an empirical case study [18]. Similar to SPF, Symclat overrode the bytecode interpretation in JPF, but it does not use attributes or the instruction factory, and is limited to handling integer symbolic inputs. Our previous tool, JPF–SE [1] performed symbolic execution by program instrumentation, which incurred significant interpretation over-head and was not fully automated. Bogor/Kiasan [19, 21] is similar to JPF–SE [1], and proposes a more efficient "lazier" approach to handling symbolic data structures (with respect to *null* objects). A formal treatment of lazy initialization is also provided in [21].

Dynamic symbolic execution or concolic testing [11, 25, 57] is an analysis technique that performs a concrete execution on random inputs and collects the path constraints along the executed path. These path constraints are then used to compute new inputs that drive the program along alternative paths. Unlike SPF, the approaches described in [11, 25, 57] use code instrumentation and don't use model checking, which we use for analyzing multithreading systematically. We remark that one could implement concolic execution in our framework by performing the concrete operations along with the symbolic ones; in fact some JPF extension projects already do that, e.g., [31].

We have already mentioned the close relationship between our work on mixed concretesymbolic solving and dynamic symbolic execution as implemented in e.g. DART [25] or EXE [11] and many other tools that implement the DART algorithm, such as CUTE [57, 58], PEX [67], and SAGE [26]. All these tools have the ability to fall back on concrete run-time values when "classical" symbolic execution would fail, i.e. when the decision procedure can not handle the complex mathematical constraints that are generated or when analyzing code that uses native or external libraries. In [45] we provide a direct comparison between SPF with mixed concrete-symbolic execution approach and DART and EXE tools on the example in Figure 9. It turns out that DART and SPF cover all the program statements while EXE does not; the example also shows that DART and EXE do not cover all the paths through the code, while SPF does.

Using the example in Figure 9, DART starts execution by generating random values for inputs. If the concrete value v of x satisfies X>0, then DART can easily generate a value for y that is equal to hash(v) (known at run-time). If v does not satisfy X>0, then DART performs an extra iteration where it first solves X>0 and sets the value of x to the solution. DART then re-executes the program and finds a value for y that is equal to the run-time value of hash(x). By first picking randomly and then fixing the value of x, DART can drive the execution of test through different program paths.

EXE takes a much simpler approach. To check PC: X>0 & Y=hash(X), EXE first generates a solution for the "simple", solvable part of the PC, X>0, and uses the solution to compute hash(1)=10. From that point on, the value of input "x" is fixed to be 1, and it will not change for the rest of the execution. The final PC generated is X>0 & Y=10 which is satisfiable. While this approach may work very well in practice, it fixes some symbolic inputs to concrete values, which may be overly restrictive. EXE therefore may miss covering some large parts of the code under analysis. DART solves this problem by re-execution. Note however that some paths (e.g., "S0, S4") remain uncovered, due to divergence [25].

In contrast, SPF with mixed concrete-symbolic solving uses uninterpreted functions to represent calls to unknown or complex functions during symbolic execution. Whenever symbolic execution needs to decide feasibility of alternate paths, we first solve the simple part of the PC, then use those values to concretize hash, and we finally solve again the path condition that was thus simplified. Note that unlike DART, SPF does not use the run-time values of program variables but instead it uses the solutions of the collected constraints. As a result SPF may use different concrete values along the same symbolic path; this would correspond to multiple concrete paths in DART. Note however that the two techniques are incomparable in power [45]. Let us remark that while the rather detailed discussion here shows some of the benefits of our technique, it is not intended to serve as a thorough comparison between different approaches. Rather it illustrates the limitations of existing techniques and indicates the need for future heuristics that can better deal with the challenges of handling external calls or complex mathematical computations.

The KLEE [10] symbolic execution engine is the follow-on tool to EXE; it no longer uses concrete values when invoking external libraries, but instead it uses models of external (unknown or unanalyzable) functions. This allows the analysis to stay completely symbolic, rather than concretizing inputs for functions that cannot be analyzed. Note that such a modeling approach is standard practice in software model checking. The approach however requires considerable manual effort.

Extensibility is a key difference between KLEE and our approach. The JPF framework provides different extensibility mechanisms through listeners, instruction factories, search strategies, etc. Using these extensibility mechanisms, SPF can be easily extended to support new algorithms. KLEE in its current form is not engineered to support extensibility in an unobtrusive manner. Furthermore, adding new constraint solvers is fairly straightforward in the SPF framework, while in KLEE there is no well-defined interface through which new constraint solvers or decision procedures can be added. Other tools that build on top of KLEE tend to take a snapshot of KLEE and make changes as needed. Examples are KLOVER [38], which provides support for symbolically analyzing C++ code and KLEE-FP [17] which reasons about equivalence between floating point operations in different C programs.

Symbolic string analysis is an active area of research. There are three main axes of comparison with related work: a graph-like representation of constraints, interaction between integer, string, and mixed constraints, and the use of automata or bitvectors. Several tools make use of some form of graph representation, but in most cases it is only peripheral. The approach of Hooimeijer and Weimer is closest to ours, but it does not include integer variables in the graph representation [27, 28]. Most other tools offer no (or very limited) support for the interaction between integer and string or mixed constraints; one exception is Kudzu [56] (for JavaScript) which uses an iterative strategy somewhat similar to our own. Lastly, the field is quite evenly divided between tools that use automata [14, 27, 28, 59, 60, 76, 77] and those that use bitvectors [34, 35, 56, 6, 67, 69]. As far as we know, we are unique in using both.

9 Conclusions

We have described Symbolic PathFinder, a tool that combines symbolic execution with model checking and constraint solving for the automated error detection and test case generation for Java bytecode programs. We have highlighted some of the techniques that are built into SPF to handle complex mathematical constraints, external library calls and symbolic string analysis. To demonstrate SPF's usability and extensibility, we further described a diversity of applications that have been recently integrated in SPF. The tool is open-source and it is being used and extended in many projects in industry and academia. SPF is under active development and plans for the near future include: extending support for input arrays with unspecified size and with non-primitive data, more scalable analysis of concurrent programs via partial evaluation and compositional techniques, and more sophisticated handling of loops. We are also working on leveraging SPF for automated testing of Android applications [41] and for security [47] and reliability [23] software analysis.

References

- Anand S, Păsăreanu CS, Visser W (2007) JPF–SE: A symbolic execution extension to Java PathFinder. In: Proc 13th Intl Conf on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS 4424, pp 134-138
- 2. Anand S, Păsăreanu CS, Visser W (2009) Symbolic execution with abstraction. International Journal on Software Tools for Technology Transfer STTT 11(1):53–67
- Balasubramanian D, Păsăreanu CS, Whalen MW, Karsai G, Lowry MR (2011) Polyglot: Modeling and analysis for multiple statechart formalisms. In: Proc 2011 International Symposium on Software Testing and Analysis (ISSTA), pp 45–55
- Balasubramanian D, Păsăreanu CS, Biatek J, Pressburger T, Karsai G, Lowry MR, and Whalen MW (2012) Integrating Statechart Components in Polyglot. In: Proc NASA Formal Methods 2012: 267-272
- Barrett C, Tinelli C (2007) CVC3. In: Proc 19th Intl Conf on Computer Aided Verification (CAV), Springer, LNCS, vol 4590, pp 298–302
- 6. Bjørner N, Tillmann N, Voronkov A (2009) Path feasibility analysis for stringmanipulating programs. In: Proc 15th Intl Conf on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Springer, LNCS, vol 5505, pp 307–321
- Borges M, D'Amorim M, Anand S, Bushnell D, Păsăreanu CS (2012) Symbolic execution with interval solving and meta-heuristic search. In: Proc 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST), pp 111-120

- Bush W, Pincus J, Sielaff D (2000) A Static Analyzer for Finding Dynamic Programming Errors. Software: Practice and Experience 30(7):775–802
- 9. Bushnell D (2011) Continuity analysis for floating point software. In: Numerical Software Verification Workshop, NSV-2011
- Cadar C, Dunbar D, Engler D (2008) KLEE: Unassisted and automatic generation of highcoverage tests for complex systems programs. In: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, USENIX Association, Berkeley, CA, USA, OSDI'08, pp 209–224
- Cadar C, Ganesh V, Pawlowski P, Dill D, Engler D (2008) EXE: Automatically Generating Inputs of Death. ACM Transactions on Information and System Security 12(2):1–38
- Cadar C, Godefroid P, Khurshid S, Păsăreanu CS, Sen K, Tillmann N, and Visser W (2011) Symbolic execution for software testing in practice: preliminary assessment. In: Proc 33rd International Conference on Software Engineering (ICSE): 1066-1071
- CERT/CC (2001) Cert Advisory: Multiple vulnerabilities in WU-FTPD. Tech. Rep. CA-2001–33
- Christensen AS, Møller A, Schwartzbach MI (2003) Precise analysis of string expressions. In: Proc 10th Intl Symposium on Static Analysis (SAS), Springer, LNCS, vol 2694, pp 1–18
- Clarke LA (1976) A system to generate test data and symbolically execute programs. IEEE Trans Softw Eng 2:215-222, DOI 10.1109/TSE.1976.233817, URL http://dl.acm. org/citation.cfm?id=1313320.1313532
- Coen-Porisini A, Denaro G, Ghezzi C, Pezzé M (2001) Using Symbolic Execution for Verifying Safety-Critical Systems. In: Proc ESEC/SIGSOFT FSE, ACM, p 151
- Collingbourne P, Cadar C, Kelly PH (2011) Symbolic crosschecking of floating-point and simd code. In: Proc of the 6th Conference on Computer systems, ACM, New York, NY, USA, EuroSys '11, pp 315–328, DOI 10.1145/1966445.1966475, URL http://doi.acm.org/ 10.1145/1966445.1966475
- d'Amorim M, Pacheco C, Xie T, Marinov D, Ernst MD (2006) An empirical comparison of automated generation and classification techniques for object-oriented unit testing. In: Proc 21st IEEE/ACM Intl Conf on Automated Software Engineering (ASE), IEEE Computer Society, Washington, DC, USA, pp 59–68
- 19. Deng X, Lee J, Robby (2006) Bogor/Kiasan: A k-bounded symbolic execution for checking strong heap properties of open systems. In: Proc 21st IEEE/ACM Intl Conf on Automated Software Engineering (ASE), IEEE Computer Society, Washington, DC, USA, pp 157–166
- Deng X, Robby, Hatcliff J (2007) Kiasan/KUnit: Automatic test case generation and analysis feedback for open object-oriented systems. In: TAIC PART– Mutation Analysis, 3rd International Workshop, pp 3–12
- Deng X, Lee J, Robby (2012) Efficient and formal generalized symbolic execution. Automated Software Engineering 19:233-301, URL http://dx.doi.org/10.1007/s10515-011-0089-9, 10.1007/s10515-011-0089-9
- Detlefs DL, Leino KRM, Nelson G, Saxe JB (1998) Extended static checking. In: SRC Research Report 159, COMPAQ Systems Research Center
- Filieri A, Păsăreanu CS, and Visser W (2013) Reliability Analysis in Symbolic PathFinder. In: Proc 35th International Conference on Software Engineering (ICSE), 2013.
- Geldenhuys J, Dwyer MB, Visser W (2012) Probabilistic Symbolic Execution. In: Proc International Symposium on Software Testing and Analysis (ISSTA), pp 166–176
- Godefroid P, Klarlund N, Sen K (2005) Dart: Directed automated random testing. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), ACM, New York, NY, USA, pp 213–223

- 26. Godefroid P, de Halleux P, Nori A, Rajamani S, Schulte W, Tillmann N, Levin M (2008) Automating Software Testing using Program Analysis. Software, IEEE 25(5):30–37
- Hooimeijer P, Weimer W (2009) A decision procedure for subset constraints over regular languages. In: Proc 2009 ACM SIGPLAN Conf on Programming Language Design and Implementation (PLDI), ACM, pp 188–198
- Hooimeijer P, Weimer W (2010) Solving string constraints lazily. In: Pecheur C, Andrews J, Nitto ED (eds) Proc 25th IEEE/ACM Intl Conf on Automated Software Engineering (ASE), ACM, pp 377–386
- 29. Hooimeijer P, Molnar D, Saxena P, Veanes M (2010) Modeling imperative string operations with transducers. Tech. Rep. MSR-TR-2010-96, Microsoft
- 30. IASolver (2010) IASolver page. http://www.cs.brandeis.edu/~tim/Applets/IAsolver. html
- Jayaraman K, Harvison D, Ganesh V, Kiezun A (2009) jFuzz: A concolic whitebox fuzzer for Java. In: NASA Formal Methods Symposium, NASA Technical Memorandum
- 32. JPF (2012) JPF project. http://babelfish.arc.nasa.gov/trac/jpf
- Khurshid S, Păsăreanu CS, Visser W (2003) Generalized symbolic execution for model checking and testing. In: Proc 9th Intl Conf on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), pp 553–568
- Kieżun A (2009) Effective software testing with a string-constraint solver. PhD thesis, Massachusetts Institute of Technology, USA
- Kieżun A, Ganesh V, Guo PJ, Hooimeijer P, Ernst MD (2009) HAMPI: A solver for string constraints. In: Rothermel G, Dillon LK (eds) Proc 2009 International Symposium on Software Testing and Analysis (ISSTA), ACM, pp 105–116
- King JC (1976) Symbolic execution and program testing. Commun ACM 19:385– 394, DOI http://doi.acm.org/10.1145/360248.360252, URL http://doi.acm.org/10.1145/ 360248.360252
- 37. LattE (2012) LattE Integrale. URL http://www.math.ucdavis.edu/~latte, UC Davis, Mathematics
- 38. Li G, Ghosh I, Rajan SP (2011) Klover: a symbolic execution and automatic test generation tool for C++ programs. In: Proceedings of the 23rd International Conference on Computer aided verification, Springer-Verlag, Berlin, Heidelberg, CAV'11, pp 609–615, URL http://dl.acm.org/citation.cfm?id=2032305.2032354
- Li X, Shannon D, Ghosh I, Ogawa M, Rajan S, Khurshid S (2008) Context-sensitive relevancy analysis for efficient symbolic execution. In: Asian Symposium on Programming Languages and Systems (APLAS)
- 40. Loera JAD, Dutra B, Köppe M, Moreinis S, Pinto G, Wu J (2011) Software for exact integration of polynomials over polyhedra, arXiv:1108.0117v2[math.MG]
- 41. Mirzaei N, Malek S, Păsăreanu C, Esfahani N, Mahmood R (2012) Testing Android apps through symbolic execution. In: JPF Workshop
- 42. de Moura L, Bjørner N (2008) Z3: An efficient SMT solver. In: Proc 14th Intl Conf on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Springer, LNCS, vol 4963, pp 337–340
- Păsăreanu CS, Visser W (2004) Verification of Java programs using symbolic execution and invariant generation. In: Proc of 11th International SPIN Workshop (SPIN), LNCS 2989 Springer pp 164–181
- 44. Păsăreanu CS, Mehlitz PC, Bushnell DH, Gundy-Burlet K, Lowry M, Person S, Pape M (2008) Combining unit-level symbolic execution and system-level concrete execution for testing nasa software. In: Proc 2008 International Symposium on Software Testing and Analysis (ISSTA), pp 15–26

- 45. Păsăreanu CS, Rungta N, Visser W (2011) Symbolic execution with mixed concretesymbolic solving. In: Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA), ACM, New York, NY, USA, pp 34–44, DOI 10.1145/2001420. 2001425
- Person S, Yang G, Rungta N, Khurshid S (2011) Directed incremental symbolic execution. In: Proc 2011 ACM SIGPLAN Conf on Programming Language Design and Implementation (PLDI), pp 504–515
- Phan Q-S, Malacaria P, Tkachuk O, and Păsăreanu CS (2012) Symbolic quantitative information flow. In: ACM SIGSOFT Software Engineering Notes 37(6): 1-5 (2012)
- Rajan S, Tkachuk O, Prasad M, Ghosh I, Goel N, Uehara T (2009) WEAVE: WEb Applications Validation Environment. In: Proc 31st International Conference on Software Engineering (ICSE Companion)
- 49. Redelinghuys G (2012) Symbolic string execution. Master's thesis, Stellenbosch University
- 50. Redis (2012) Redis NoSQL database. URL http://redis.io
- 51. Rossi F, van Beek P, Walsh T (2006) Handbook of Constraint Programming. Elsevier
- Rungta N, Mercer EG, Visser W (2009) Efficient testing of concurrent programs with abstraction-guided symbolic execution. In: Proc of 16th International SPIN Workshop (SPIN) pp 174-191
- 53. Sanfilippo S, Noordhuis P (2012) Redis: The Definitive Guide. O'Reilly Media
- Santelices R, Harrold MJ (2010) Exploiting program dependencies for scalable multiplepath symbolic execution. In: Proc 2010 International Symposium on Software Testing and Analysis (ISSTA), pp 195–206
- 55. Sasnauskas R, Dustmann OS, Kaminski BL, Wehrle K, Weise C, Kowalewski S (2011) Scalable symbolic execution of distributed systems. In: Proceedings of the 2011 31st International Conference on Distributed Computing Systems, IEEE Computer Society, Washington, DC, USA, ICDCS '11, pp 333–342, DOI 10.1109/ICDCS.2011.28, URL http://dx.doi.org/10.1109/ICDCS.2011.28
- Saxena P, Akhawe D, Hanna S, Mao F, McCamant S, Song D (2010) A symbolic execution framework for JavaScript. In: Proc 31st IEEE Symposium on Security and Privacy, IEEE Computer Society, pp 513–528
- Sen K, Agha G (2006) CUTE and jCUTE : Concolic unit testing and explicit path modelchecking tools. In: Proc. 18th International Conference on Computer Aided Verification (CAV), pp 419–423
- Sen K, Agha G (2007) A race-detection and flipping algorithm for automated testing of multithreaded programs. In: Proc. Haifa Verification Conference (HVC), Springer, LNCS, vol 4383, pp 166–182
- Shannon D, Hajra S, Lee A, Zhan D, Khurshid S (2007) Abstracting symbolic execution with string analysis. In: Proc Testing: Academic and Industrial Conf, Practice and Research Techniques, IEEE Computer Society, pp 13–22
- Shannon D, Ghosh I, Rajan SP, Khurshid S (2009) Efficient symbolic execution of strings for validating web applications. In: Proc 2nd Intl Workshop on Defects in Large Software Systems, ACM, pp 22–26
- Siegel S, Zirkel T (2011) Tass: The toolkit for accurate scientific software. Mathematics in Computer Science 5:395–426, URL http://dx.doi.org/10.1007/s11786-011-0100-7, 10.1007/s11786-011-0100-7
- Siegel S, Mironova A, Avrunin G, Clarke L (2006) Using Model Checking with Symbolic Execution to Verify Parallel Numerical Programs. In: Proc 2006 International Symposium on Software Testing and Analysis (ISSTA), ACM, pp 157–168

- 63. Souza M, Borges M, d'Amorim M, Păsăreanu CS (2011) CORAL: Solving complex constraints for Symbolic PathFinder. In: NASA Formal Methods, pp 359–374
- 64. SPF (2012) Symbolic Pathfinder (jpf-symbc). http://babelfish.arc.nasa.gov/trac/jpf
- Staats M, Păsăreanu C (2010) Parallel symbolic execution for structural test generation. In: Proc 2010 International Symposium on Software Testing and Analysis (ISSTA), ACM, New York, NY, USA, pp 183–194, DOI 10.1145/1831708.1831732
- 66. choco: (2012) Java constraint solver. URL http://choco.emn.fr
- 67. Tillmann N, de Halleux J (2008) Pex–white box test generation for .NET. In: Beckert B, Hähnle R (eds) Proc 2nd Intl Conf on Tests and Proofs, Springer, LNCS, vol 4966, pp 134–153
- Tomb A, Brat G, Visser W (2007) Variably interprocedural program analysis for run-time error detection. In: Proc 2007 International Symposium on Software Testing and Analysis (ISSTA), ACM Press, New York, NY, USA, pp 97–107
- Veanes M, de Halleux P, Tillmann N (2010) Rex: Symbolic regular expression explorer. In: Proc 3rd Intl Conf on Software Testing, Verification and Validation, IEEE Computer Society, pp 498–507
- 70. Visser W, Havelund K, Brat GP, Park S, Lerda F (2003) Model checking programs. Automated Software Engineering 10(2):203–232
- Visser W, Păsăreanu CS, Pelánek R (2006) Test input generation for Java containers using state matching. In: Proc 2006 International Symposium on Software Testing and Analysis (ISSTA), pp 37–48
- 72. Visser W, Geldenhuys J, Dwyer MB (2012) Green: Reducing, Reusing and Recycling Constraints in Program Analysis. In: International Symposium on the Foundations of Software Engineering (FSE), Cary, North Carolina, USA
- 73. Xie T, Marinov D, Schulte W, Notkin D (2005) Symstra: A framework for generating object-oriented unit tests using symbolic execution. In: Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Springer-Verlag, Berlin, Heidelberg, pp 365–381
- 74. Yang G, Păsăreanu CS, and Khurshid S (2012) Memoized symbolic execution. In: Proc International Symposium on Software Testing and Analysis (ISSTA), pp 144–154
- 75. Yices (2012) Yices SMT Solver. URL http://yices.csl.sri.com/
- 76. Yu F, Bultan T, Cova M, Ibarra OH (2008) Symbolic string verification: An automatabased approach. In: Proc 15th Intl SPIN Workshop on Model Checking Software, Springer, LNCS, vol 5156, pp 306–324
- 77. Yu F, Alkhalaf M, Bultan T (2010) Stranger: An automata-based string analysis tool for PHP. In: Proc 16th Intl Conf on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Springer, LNCS, vol 6015, pp 154–157
- 78. Zhang P, Elbaum SG, Dwyer MB (2011) Automatic generation of load tests. In: Alexander P, Păsăreanu CS, Hosking JG (eds) Proc 26th IEEE/ACM Intl Conf on Automated Software Engineering, IEEE, pp 43–52