Combining Unit-level Symbolic Execution and System-level Concrete Execution for Testing NASA Software

Corina S. Păsăreanu,
Peter C. Mehlitz,
David H. Bushnell,
Karen Gundy-Burlet,
Michael Lowry
NASA Ames Research Center
Moffett Field, CA 94035-1000
{corina.s.pasareanu,
peter.c.mehlitz,
david.h.bushnell,
karen.gundy-burlet,
michael.r.lowry}@nasa.gov

Suzette Person
Department of Computer
Science and Engineering
University of Nebraska
Lincoln, NE 68588-0115
sperson@cse.unl.edu

Mark Pape
NASA Johnson Space Center
2101 NASA Parkway
Houston, TX 77058
mark.w.pape@nasa.gov

ABSTRACT

We describe an approach to testing complex safety critical software that combines unit-level symbolic execution and system-level concrete execution for generating test cases that satisfy user-specified testing criteria. We have developed Symbolic Java PathFinder, a symbolic execution framework that implements a non-standard bytecode interpreter on top of the Java PathFinder model checking tool. The framework propagates the symbolic information via attributes associated with the program data. Furthermore, we use two techniques that leverage system-level concrete program executions to gather information about a unit's input to improve the precision of the unit-level test case generation.

We applied our approach to testing a prototype NASA flight software component. Our analysis helped discover a serious bug that resulted in design changes to the software. Although we give our presentation in the context of a NASA project, we believe that our work is relevant for other critical systems that require thorough testing.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Reliability, Verification

ISSTA'08, July 20–24, 2008, Seattle, Washington, USA.
Copyright 2008 ACM 978-1-59593-904-3/08/07 ...$5.00.

Keywords

Symbolic Execution, Software Model Checking, Unit Testing, System Testing

1. INTRODUCTION

Our work is motivated by an ongoing collaboration between NASA Ames and Johnson Space Center, whose goal is to develop *automated* techniques for error detection in complex, flight control software for manned space missions. Such software needs to be highly reliable. Techniques for checking software include model checking (e.g., [23, 26, 18, 7, 22]), static analysis (e.g., [29]), and testing. Model checking *exhaustively* analyzes all program executions in a systematic way, but it suffers from scalability issues. Static analysis is scalable and exhaustive, but it may give many warnings that are spurious (i.e., do not correspond to real errors). Testing, on the other hand, reports errors that are real, but it may miss errors since it can only analyze *some* of the program executions. Furthermore, testing is the most widely used method for error detection at NASA, as well as elsewhere. We aim to combine the strengths of automated exhaustive techniques to make testing more effective.

Towards this end we have developed Symbolic Java Path Finder (Symbolic JPF), a framework that integrates symbolic execution [28, 11] with model checking to perform *automated* generation of test cases and to check properties of code during test case generation. Symbolic JPF generates test cases that obtain high coverage for flexible, user-defined, coverage metrics. Programs are executed on symbolic inputs that represent all possible concrete inputs. Values of variables are represented as numeric (mixed integer and real) constraints, encoding the conditions from the code. These constraints are then solved to generate test inputs guaranteed to exercise the analyzed code. Pre-conditions, when available, may be leveraged to reduce the size of the input data domains and to only generate test inputs that satisfy the pre-conditions. The framework uses the analysis engine

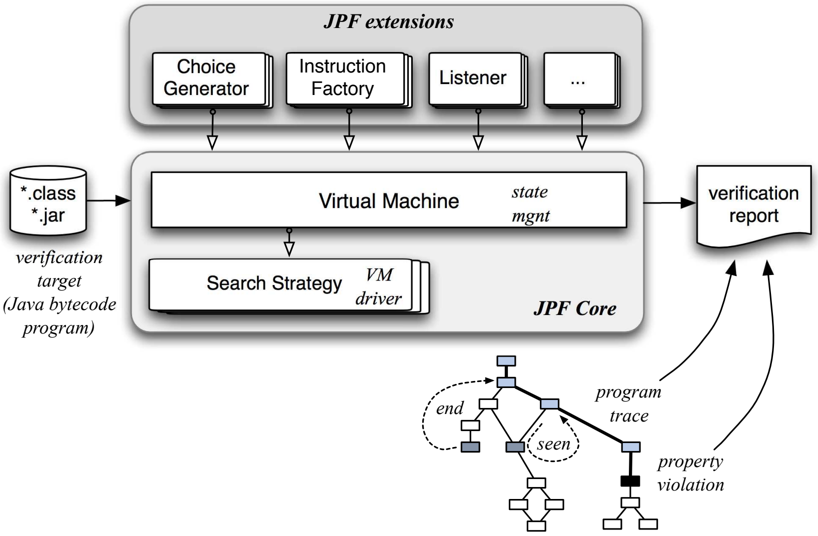

Figure 1: JPF high-level structure.

of the Java PathFinder (JPF) model checking tool and it is available at [26], the `symbc` JPF extension.

Unlike previous symbolic execution approaches (e.g., our own previous approach [27, 5] but also [19, 32]) that work by code instrumentation, this new framework does not require such instrumentation, but instead implements a "non-standard" interpreter of bytecodes. Moreover, the approach in [27] requires an (approximate) type-based static analysis [4] to determine if a program bytecode needs to be executed symbolically, and therefore needs to be instrumented. Symbolic JPF does not require such approximate analysis, since the symbolic information is stored in attributes associated with the program data and and is propagated dynamically during execution. As a result, our framework always maintains the most accurate information about the symbolic nature of the program data.

Symbolic JPF is quite general and it can be applied at different phases of software development, but it is most effective for unit (or sub-system) level testing. It is often the case that the input data to the unit is constrained by the environment, e.g., the calling context of a procedure representing the unit. To avoid generation of un-realistic test cases, such constraints would need to be encoded explicitly, which would require non-trivial additional manual effort from the software developers. We alleviate this problem in two ways.

- First, the new framework allows symbolic execution to be started at *any point* in the program and at *any time* during the concrete execution of a program. More specifically, one can let a program run in concrete execution mode within JPF's specialized Java virtual machine, and trigger symbolic execution based on some condition on the current 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 method/procedure symbolically, while some of the parameters and the calling context of the procedure are kept concrete.
- It is not always possible to run the whole program within the JPF's customized execution environment (due to sheer size, native libraries, hardware-software interaction, etc.). We therefore consider a second approach that uses *actual* system-level simulation runs to determine constraints on unit input data. Such constraints are then encoded as unit pre-conditions that help improve the precision of the unit-level symbolic analysis by avoiding generation of test cases that violate the constraints.

We describe the application of our techniques to a component of a NASA flight software system. Our symbolic framework generated in a few seconds a test suite that obtained full testing coverage, for a special coverage required by the developer. In contrast, random testing obtained only partial coverage while manual test case generation took approx. 20 hours to obtain not quite adequate coverage. During test generation, the framework also discovered errors that were later fixed by the developers. Furthermore, the generated test suite was applied to a new version of the code, where it helped uncover a subtle bug that led to the re-design of part of the flight software.

The rest of the paper is organized as follows. We begin with some background information on Java PathFinder (Section 2) and on symbolic execution – the enabling technique for test case generation (Section 3). We present our new framework for symbolic execution: we describe the extensions to Java PathFinder that were implemented to support this framework (Section 4) and we outline different mechanisms for combining system-level concrete program runs with unit-level symbolic execution (Section 5). We also discuss applications of our framework (Section 6), related work (Section 7) and conclusions (Section 8).

2. JAVA PATHFINDER (JPF)

JPF [26] is an open-source runtime environment for verifying Java bytecode, i.e., programs written in Java's intermediate representation. JPF consists of a backtrackable,

```
    int x, y;
[1] if (x > y)
[2]   result = x - y;
[3] else
[4]   result = y - x;
[5] assert (result > 0);
```

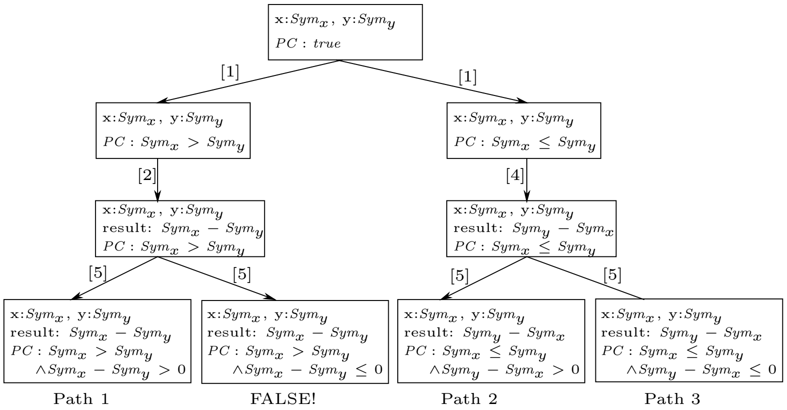

Figure 2: Example for symbolic execution (left) and corresponding execution tree (right).

state matching and state storing *virtual machine* (VM), and a configurable *search* strategy object that drives the execution in the VM (see Figure 1). The search and VM components together form the JPF core – a software model checker that can be directly applied to Java bytecode, to find property violations like unhandled exceptions, race conditions and deadlocks.

JPF employs a variety of mechanisms to reduce the number and storage costs of program states, such as on-the-fly partial order reduction and hash collapsing. Furthermore, JPF provides configurable *extensions* to define:

- operations that force the execution to branch (*Choice Generators*)
- code that should be executed outside JPF (*Native Peers*, e.g., for abstracting native libraries)
- code that allows changing the semantics of execution for bytecode instructions (*Instruction Factories*)
- code that non-intrusively monitors and controls JPF program execution (*Listeners*)
- properties to check for (such as no unhandled runtime exceptions)

In Section 4, we describe how we used JPF's configurable extensions to implement our symbolic execution framework.

3. SYMBOLIC EXECUTION

Symbolic execution [28] is a form of program analysis that uses symbolic values instead of actual data as 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 path condition is a boolean formula over the symbolic inputs, encoding the constraints which the inputs must satisfy in order for an execution to follow the particular associated path. The paths followed during the symbolic execution of a program are characterized by a *symbolic execution tree*.

To illustrate the difference between concrete and symbolic execution, consider the simple example in Figure 2 (left) that computes the absolute difference between two input integers `x` and `y`. Assume that the values of the input parameters are `x=2` and `y=1`.

Concrete execution will follow only one program path, corresponding to the *true* branch of the `if` statement at line `[1]`; this execution does not violate the assertion.

In contrast, symbolic execution starts with symbolic, rather than concrete, values, $\texttt{x} = Sym_x$, $\texttt{y} = Sym_y$, and the initial value of PC is *true*. The corresponding (simplified) execution tree is illustrated 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. If the path condition becomes "false", it means that the corresponding path is infeasible (and symbolic execution does not continue for that path).

For our example, symbolic execution explores three different feasible paths, it determines that a fourth path is infeasible and it reports an assertion violation (for Path 3). For test case 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.

3.1 Generalized Symbolic Execution

In previous work [27, 5], we have extended JPF to perform *generalized* symbolic execution for Java programs. The approach handles dynamically allocated data, arrays, and multi-threading. Programs are instrumented to enable JPF to perform symbolic execution; concrete types are replaced with corresponding symbolic types and concrete operations are replaced with calls to methods that implement corresponding operations on symbolic expressions. The model checker checks properties of the instrumented program using its usual state space exploration techniques. Several off-the-shelf decision procedures are used to check satisfiability of numeric path conditions.

While quite general, our previous approach may result in sub-optimal execution since for each instrumented bytecode the model checker needs to check a set of bytecodes representing the symbolic counterpart. To minimize the instrumentation effort, a specialized type based analysis is proposed in [4]. This analysis computes a conservative ap-

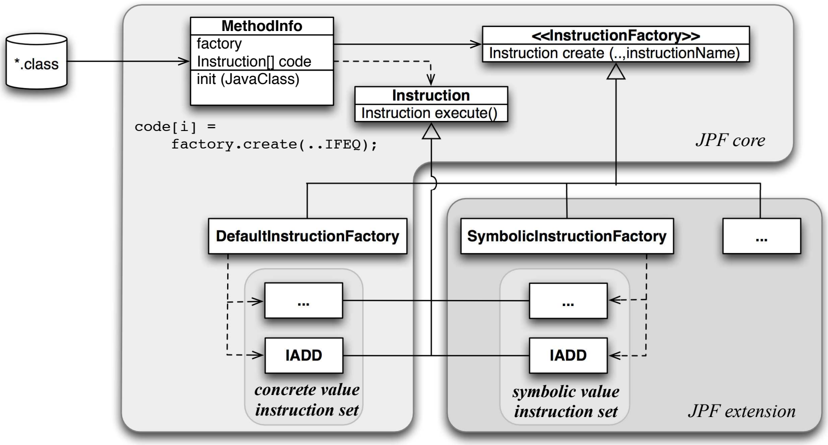

Figure 3: Bytecode factories for concrete vs. symbolic execution.

proximation of type-dependence information that is used to locate the parts of the code that depend on the input symbolic variables. Therefore only those parts need to be transformed into their symbolic counterparts (the rest of the code remaining unchanged).

To address the problems described above, we have developed a new framework that does not require the code instrumentation and the approximate type based analysis. This framework is described in detail in the next section.

4. SYMBOLIC JAVA PATHFINDER

The symbolic execution framework is built as an extension of JPF – the framework performs a non-standard bytecode interpretation and uses JPF to systematically generate and execute the symbolic execution tree of the code under analysis. The key mechanisms that we used were:

- JPF's bytecode instruction factory and
- attributes associated with the program state.

The instruction factory allows replacing or extending the standard, concrete execution semantics of bytecodes with a non-standard (symbolic) execution, as desired. The symbolic information is stored in *attributes* associated with program data (fields, stack operands and local variables) and it is propagated as needed, during symbolic execution.

These mechanisms together allow dynamic modification of execution semantics, i.e., changing mid-stream from a system-level, concrete execution semantics to a symbolic execution semantics, thus providing the integrated test generation capability described later in this paper. We note also that they enable other types of integrated program analyses that we plan to pursue in future work.

Furthermore, we used JPF's choice generators, for handling branching conditions during symbolic execution, and listeners, for printing the results of the symbolic analysis (i.e., method summaries) and for enabling dynamic change of execution semantics. We also used native peers, for modeling native libraries, e.g., to capture `java.lang.Math` library calls and to send them to the constraint solver.

We describe some of these features in more detail below.

4.1 An Instruction Factory for Symbolic Execution of Bytecodes

JPF analyzes an input Java program (class files) by interpreting the Java bytecodes in a custom-made Virtual Machine. JPF implements a "default", concrete execution semantics that is based on a stack machine model, according to the Java VM specification [30]. Furthermore, JPF allows replacing this standard execution semantics by using a configurable *InstructionFactory* (see Figure 3).

For each method that is executed, JPF maintains an object (`MethodInfo`) that internally stores the associated bytecode as an array of `Instruction` objects, which are created from the bytecodes read from the corresponding class file. JPF imposes few constraints on `Instruction` classes other than requiring an `execute()` method.

JPF uses the *abstract factory* design pattern [16] to instantiate its `Instruction` objects. We therefore created a `SymbolicInstructionFactory` containing instructions for the symbolic interpretation of Java bytecodes. The new `Instruction` classes are derived from the ones that come with the JPF core; they conditionally add new functionality and otherwise just delegate to their super classes. This enables simultaneous concrete-symbolic execution modes.

4.2 Attributes

JPF maintains program states very similar to a standard Java VM. Each *state* consists of a call stack per thread, the values of the fields (i.e., the heap) and the scheduling information. The call stack contains stack frames corresponding to the methods that are being executed. Each stack frame stores information about the locals and the operands.

Figure 4 illustrates the state representation in JPF core, including the heap and the stack frame for the currently executing method, i.e., the "callee", as well as the stack frame for the "caller" method. The values of the heap and the stack frames are manipulated via various bytecode operations, such as copy values between operand slots of Stack Frames (`dup`), between local variable slots and operand slots of Stack Frames (`istore`), and between operand slots and heap object fields (`putfield` and `getfield`).

Figure 4 also illustrates the *attributes* associated with the program values. We used a previous experimental extension

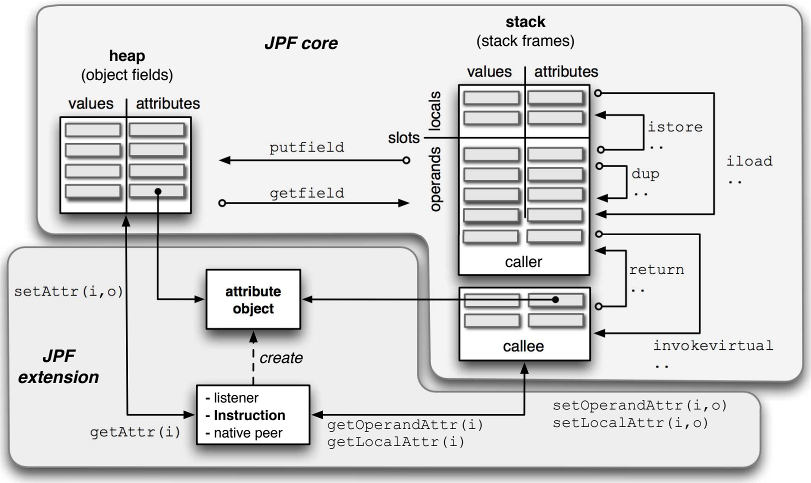

Figure 4: Attributes.

of JPF that associates so called *slot attributes* i.e., additional, state-stored information, with each of the locals and operands on the stack frame. This experimental extension was initially designed for verifying numeric properties.

We generalized this mechanism by providing support for field attributes (in addition to slot attributes). In our framework, the attributes are used to store the symbolic values and expressions that are created during symbolic execution.

Attributes are created or accessed by bytecode instructions, via accessors (e.g., methods `setAttr`, `getAttr`). They can also be created or accessed by listeners or native peers (see Figure 4). Attribute manipulation is mainly done inside of the JPF core, within the various operations that modify and store the program states (such as `dup`, `istore`, `putfield`, and `getfield`).

Therefore, we only needed to override instruction classes that create or modify symbolic information, like numeric, compare-and-branch and type conversion operations. Other bytecode instructions, that only retrieve or store the symbolic information, remained un-changed.

Note that while this mechanism was developed in the context of our symbolic execution mode, it is now generalized sufficiently to allow arbitrary value and variable attributes, and therefore it can be useful for implementing other analyses (e.g., to keep track of physical dimensions and numeric error bounds or to perform concolic execution [19]).

4.3 Handling Branching Conditions

The symbolic execution of branching conditions involves creating a non-deterministic choice in JPF's search and adding the condition (or its negation) to the corresponding path condition. We achieved this by creating 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 constraint solver. If the path condition becomes un-satisfiable, JPF is instructed to backtrack.

Our framework currently uses the `choco` constraint solver [10] for integer and real constraints, and `IASolver` [25], an interval arithmetic solver that can handle complex `Math` functions. The two constraint solvers are integrated in Symbolic JPF via a common, generic, interface; the user can specify which one to use. We also plan to incorporate other decision procedures/constraint solvers in our analysis (similar to [5]).

4.4 Multi-threading, State-matching, Loops

As mentioned, our framework uses JPF to explore the symbolic execution tree of the analyzed program. JPF is also used to analyze thread interleavings and other forms of non-determinism that might be present in the code. We do not require the model checker to perform state matching (this is in general undecidable when states represent path conditions on un-bounded data). To limit the possibly infinite (symbolic) search state space that results from analyzing programs with loops or recursion, we put a limit on the model checker's search depth or on the number of constraints encoded in the path condition.

4.5 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 5 (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]`), they are added and the result is pushed back on the stack (line `[3]`). JPF is then instructed to execute the next bytecode (line `[4]`).

Figure 5 (right) shows the (simplified) code that implements the "symbolic" counterpart. Class `IntegerExpression` implements symbolic integer expressions; a similar class, `RealExpression`, implements symbolic real expressions.

The symbolic information is propagated via attributes. Method `execute` first checks if the attributes associated with the two operands are `null`. If they are, then the two operands must be 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

```
   public class IADD extends Instruction { ...
    public Instruction execute (... ThreadInfo th){
[1]    int v1 = th.pop();
[2]    int v2 = th.pop();
[3]    th.push(v1 + v2, ...);
[4]    return getNext(th);
     }
    }
```

```
   public class IADD extends ....bytecode.IADD {
    public Instruction execute (... ThreadInfo th) {
[1]   IntegerExpression sym_v1, sym_v2;
[2]   sym_v1 = ... .getOperandAttr(0);
[3]   sym_v2 = ... .getOperandAttr(1);
[4]   if (sym_v1 == null && sym_v2 == null)
        // both values are concrete
[5]     return super.execute(ss, ks, th);
[6]   else {
[7]     int v1 = th.pop();
[8]     int v2 = th.pop();
        ...
[9]    th.push(0, ...); // don't care about concrete value
[10]    IntegerExpression result =
             IntegerExpression._plus(sym_v1,sym_v2);
[11]    ... .setOperandAttr(result);
[12]    return getNext(th);
     } } }
```

Figure 5: Concrete (left) and symbolic (right) execution for the IADD bytecode.

```
   public class IFGE extends Instruction {
    public Instruction execute (... ThreadInfo th) {
[1]    condition = (th.pop() >= 0);
[2]    if (condition)
[3]      next=getTarget();
[4]    else
[5]      next=getNext(th);
[6]    return next;
   }
}
```

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

Figure 6: Concrete (left) and symbolic (right) execution for the IFGE bytecode.

symbolic expression (line [11]). Since the result becomes symbolic, its concrete value does not matter, 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 6 (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 6 (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 non-deterministic 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]) or its negation (line [13]). Method isSatisfiable uses a decision procedure to check if the path condition is satisfiable or not, in which case JPF backtracks (line [16]).

5. USING SYSTEM-LEVEL CONCRETE EXECUTIONS

Symbolic JPF is quite general and it can be applied at different phases of software development, i.e., unit-level or system-level testing. However due to inherent limitations, such as availability of decision procedures for the application domains and number of constraints that can be handled by the constraint solver, our tool is most effective at unit (or sub-system) level. A unit is a Java method, or set of methods, but it can also be an arbitrary piece of code designated as such by the user. To simplify our discussion, we only consider here Java methods as units. It is often the case that the input data to the unit is constrained by the environment, e.g., the calling context of a procedure representing the unit. These environment constraints need to be encoded explicitly to avoid generation of un-realistic unit level test cases. We describe here two techniques that address this problem. Both techniques use system-level concrete execution to set-up the symbolic execution at the unit-level.

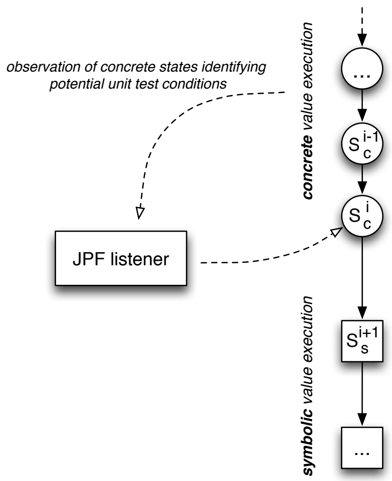

Figure 7: Combining concrete and symbolic execution.

5.1 "Any time" symbolic execution

In our framework, symbolic execution can start from any point in the program and it can perform symbolic execution of the unit, with mixed concrete and symbolic inputs. Furthermore, no special test driver needs to be created; it is sufficient to have an executable program that uses the unit.

To execute a method symbolically, the user needs to instruct JPF to use the `SymbolicInstructionFactory` and to specify the method name and the method inputs that are to be considered symbolic or concrete. These inputs include the method arguments and the globals (i.e., fields) that are accessed (i.e., read) by the method code.

The program begins execution using the concrete semantics, since all the symbolic bytecode `Instruction` classes simply delegate execution to the "concrete" super-class when there are no symbolic attributes associated with the data (see the examples in Figure 5 line `[5]` and Figure 6 line `[4]`). A listener monitors the concrete execution of the program within JPF's VM and it triggers symbolic execution the first time the method with the specified name is invoked, i.e., it "injects" new symbolic values in the attributes of the specified symbolic inputs (parameters and globals).

From that point on, the execution proceeds symbolically (e.g., the methods invoked by the designated method continue to process the symbolic information stored in the attributes). Once the method returns (or some user specified limit has been reached), JPF prints the method summary, i.e., a set of test cases characterizing the symbolic execution.

In general, one can trigger symbolic execution of a unit "any time" during concrete execution, by writing a specialized listener (see Figure 7). The listener is watching for certain concrete variable conditions and method sequences that identify program states which should be analyzed symbolically and it starts symbolic execution when the conditions are met. It is also possible to switch back to concrete execution, i.e., solve the constraints in the current path condition, compute the corresponding concrete values of the program variables and continue execution in concrete mode. We plan to investigate this feature for handling native library calls.

Since the unit can be analyzed with mixed concrete and symbolic inputs, one can use the concrete execution of a program to set up different concrete global contexts for the unit-level symbolic analysis. The use of concrete correlated input values also reduces the complexity of path conditions in the symbolic analysis (which in turn improves the performance and rate of success of the constraint solver when computing solutions for the path conditions).

Furthermore, our approach allows us to generate tests that exercise "deep" system executions. An example where this could be useful would be a complex server application that can run for a very long time. Instead of trying to symbolically analyze the whole state space (which might be impossible), such an application could be executed in concrete mode until a certain condition is recognized that warrants closer inspection. At this point, symbolic execution can take over and provide the data for test cases that specifically stress the suspicious program states.

Moreover, we can use our mixed concrete-symbolic execution to extend existing tests. For example, this could be useful when generating test sequences for Java containers [33]. A test in this case is a sequence of `add` and `remove` methods that add and remove elements to and from the container. One can use an existing test sequence to set-up the content of the heap for the container and subsequently execute `add` or `remove` symbolically to increase testing coverage. This would be much less expensive than executing the whole test sequence symbolically.

5.2 Using system-level simulation runs

We consider here a second technique that uses actual system-level simulation runs to improve the precision of unit-level analysis. The technique is illustrated in Figure 8. Multiple system runs are obtained via Monte Carlo simulations. The values of various system variables that form the unit's interface are monitored and the results are recorded in a log file that is then analyzed to determine "likely" correlations on unit input parameters. These correlations are then encoded as pre-conditions to the unit and they are asserted as initial path conditions for the symbolic execution of the unit. As a result, only test inputs that satisfy the input pre-conditions are generated.

As a simulation environment, we use ANTARES [1] (the Advanced NASA Technology ARchitecture for Exploration Studies), a trajectory simulation tool used by the JSC Guidance Navigation and Control (GN&C) community in support of NASA missions. ANTARES is a collection of spacecraft system related models and libraries that are assembled and executed by Trick, a C-based simulation environment.

Currently, ANTARES is used for spacecraft design assessment, performance analysis, requirements validation, Hardware in the Loop and Human in the Loop testing.

To execute a run in ANTARES, the user specifies an input file which contains all of the information that Trick needs to properly initialize the data in every model. There are several ANTARES input file templates that have been verified and validated, and the user typically needs only to modify one of these files for their specific task.

The input file also specifies the log file, which contains a list of the variables to be monitored during the run. The resulting data files can take up several gigabytes, depending on the number of variables and the frequency with which they are written to the log file.

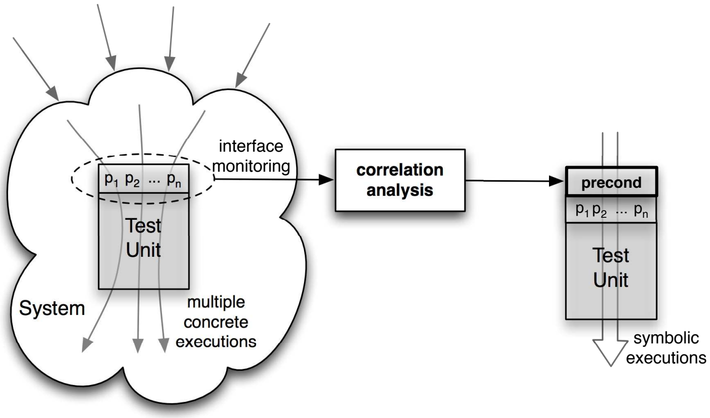

Figure 8: Using system simulations to determine unit pre-conditions.

One can also set up Monte Carlo simulations by writing an input file which specifies designated input variables, their probability distributions and how many cases one wishes to run while sampling from the probability distributions. This allowed us to obtain simulations with widely varying initial system data.

The correlation analysis uses machine learning techniques (e.g., [31, 12]) to determine unit input constraints. For the case study presented in the next section, we only needed a simple analysis of the log file to determine simple correlations in terms of range restriction on unit inputs. However, for other, more complex applications, we are experimenting with the treatment learner from [31] to determine more complex correlations.

We remark that this second technique can be combined with the first technique presented above in Section 5.1. Indeed, one can run a program within JPF and obtain input constraints for the unit, and then use these as pre-conditions during the symbolic execution analysis. In fact, an interface between JPF and the Daikon invariant detector tool [12] already exists, and it can be used for our purpose.

6. CASE STUDY

In this section we describe the application of our approach to testing NASA software.

6.1 On-board Abort Executive (OAE)

We applied our approach to a Java model of the Crew Exploration Vehicle's prototype ascent abort handling software, the Onboard Abort Executive (OAE). The OAE monitors the status of the vehicle during the ascent phase of flight. It decides the following:

- when an abort is required,
- which abort mode is currently safest for the astronauts,
- when to automatically initiate an ascent abort.

The high-level structure of the code is shown in Figure 9. The OAE receives its inputs (e.g., current altitude, launch vehicle internal pressures, etc.) from sensors and other software components. The inputs are analyzed to determine if any of the ascent flight rules have been broken, and to evaluate which ascent abort modes are currently possible. If multiple abort modes are currently possible, the OAE chooses the abort mode that is safest for the flight crew. The OAE also predicts what the vehicle's abort mode options will be in a short time. This gives the crew the opportunity to postpone abort initiation in favor of a safer abort mode in the near future. Once a flight rule is broken, and an abort mode is chosen, it is sent to the rest of the flight software for initiation. The analyzed code is approximately 600 lines of code, it has a large input space (approximately 65 input variables) and fairly complicated logic.

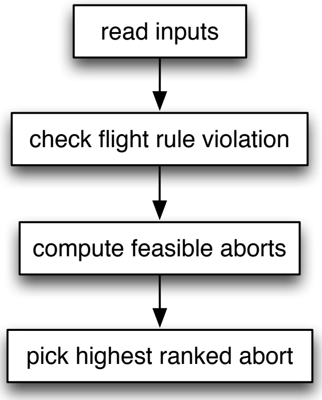

Figure 9: The OAE code structure.

There are many ascent flight rules and abort modes, and encoding them correctly in software is difficult. Since the OAE is directly concerned with human safety, it is imperative that this code be correctly implemented and carefully verified. Currently, test generation is done by hand by JSC engineers, so it is time-consuming to ensure that all necessary test cases are created. Our goal was therefore to demonstrate that our approach can quickly and automatically generate all required test cases for the OAE. During test case generation, our approach also checked known safety properties derived from the requirements.

6.2 Properties

Examples of safety properties that we checked for the OAE are:

- If a flight rule is violated, then an abort mode must be chosen.
- If no flight rules are violated, then an abort mode must not be chosen.
- If both performance and systems abort conditions exist, then the systems abort rules should apply.

We encoded these properties as assertions in the code.

6.3 Test Coverage

The OAE requires different kinds of code coverage for its test suite. These include abort coverage, flight rule coverage, combinations of abort/flight rules coverage, and branch coverage. Initially, we generated a maximal test suite that included test cases violating multiple flight rules. However, since the OAE was an early prototype, it was not designed to handle multiple simultaneous flight rule violations. We therefore generated a new set of test cases for single flight rule violations (by instructing JPF to backtrack when a second flight rule violation occurs). More generally, one can customize JPF's search to satisfy user-specified test coverage criteria (using JPF listeners).

6.4 Results

We used our symbolic execution framework to generate approximately 200 test cases to cover all aborts and flight rules. The total execution time was less than 1 minute. A typical test case required solving path conditions with between 70 and 85 clauses. During test case generation we also discovered an error (flight rules broken but no abort picked) which was later corrected by the OAE developers. Each test case includes the values of the input variables and the expected output abort mode.

Note that manual test case generation took more than 20 hours and did not cover all possible flight rule/abort combinations. We also experimented with random testing, which covered only a few flight rules and no aborts; this is not surprising, given the large input state space and the complicated conditions in the code.

The engineers at JSC used the test cases that we generated both as part of their test suite for this early version of OAE and as regression tests for later versions. This proved to be important as one of the generated test cases (run as a regression test) identified a significant design error in the next version of OAE and resulted in design changes that affected not just the flight rules and abort code, but also several other modules.

We also analyzed a later version of the OAE that implemented the predicted abort mode functionality. The analysis procedure was the same as for the original OAE model, and it generated approximately 300 test cases in less than 2 minutes.

6.5 Input Constraints

In our case study, we analyzed the OAE component in isolation, by writing a driver that invokes the component. However, the OAE operates in a vehicle subject to real-world constraints, so some combinations of input parameters that might cause the code to detect flight rule violations are not physically possible. For example, we initially generated a test suite that contained a case that set the inertial velocity of the vehicle to 24000 ft/s, though the altitude of the vehicle was only 0 feet (a physically impossible combination). We therefore needed to encode such constraints explicitly to avoid generation of un-realistic test cases. The constraints were expressed as range restrictions on input variables or as simple functions of input variables. These were asserted as pre-conditions to the OAE code and were thereby automatically included in the path conditions.

Some of the input constraints were directly provided by the domain experts, while other constraints, such as range restrictions, were determined from ANTARES simulation runs. For the OAE, we determined the minimum (min) and maximum (max) values of the pertinent OAE input variables.We used these values to encode the ranges $[min - \delta \ldots max + \delta]$ as pre-conditions in the OAE code; we used the extra quantity δ to increase the chances that symbolic execution would discover failure cases. As a result, we were able to generate test suites that did not contain physically impossible test cases, but still achieved the desired coverage.

In the future, as we will need to analyze new, more complex versions of the OAE, we plan to use the treatment learner [31] to correlate the ranges of the input variables with different phases of the abort logic.

In general, we should note that the analysis of the simulation data can only give "likely" correlations on the unit inputs. While these correlations help focus the symbolic execution, they need to be used with care, since they might mask unit behaviour. This is the reason we introduced the δ above. We need to investigate more how one can "expand" the input domains in the case of more complex correlations.

7. RELATED WORK

The work related to the topic of this paper is vast, and for brevity we only highlight here some of the closely related works.

Several approaches have been proposed that use model checking for test input generation [2, 17, 21, 24]. In these approaches, one specifies as a (temporal) property that a specific coverage cannot be achieved and a model checker is used to produce counterexample traces, if they exist, that then can be transformed into test inputs to achieve the stated coverage.

Two popular software model checkers, BLAST and SLAM, have also been used for generating test inputs with the goal of covering a specific predicate or a combination of predicates [8, 6]. Both these tools use over-approximation based predicate abstraction and use some form of symbolic evaluation for the analysis of (spurious) abstract counterexamples and refinement. We use a hybrid approach that combines model checking with symbolic execution and constraint solving for test case generation. Furthermore, we have shown how to use system-level concrete executions to improve unit-level test case generation.

Program analysis based on symbolic execution has received a lot of attention recently, e.g., [14, 34, 19, 32, 9].

The Extended Static Checker (ESC) [14] uses a static analysis to verify partial correctness of Java classes. Although our focus here is on test case generation, we can also use our symbolic execution framework to check light-weight properties in a way similar to ESC.

Symstra [34] uses a specialized symbolic execution over numeric data for generating test sequences for (sequential) Java containers. We provide here a general framework for the symbolic execution of arbitrary Java bytecode. Symclat is an experimental implementation of symbolic execution in JPF that was done in the more general context of an empirical case study [3]. Similar to Symbolic JPF, Symclat was done via changing the byte-code interpretation, but it did not use attributes or the instruction factory, and was limited to handling integer symbolic inputs. Bogor/Kiasan [13] is similar to JPF–SE [5], but uses a "lazier" approach.

We are working towards making our framework supersede in functionality our previous tool [27, 5], but without requiring the code instrumentation. Therefore, all the applications from our previous work, such as test sequence generation for Java containers [33] should be possible with our new approach. Furthermore, the ability to alternate dynamically between concrete and symbolic execution (as described in Section 5) opens up a new set of applications that need to be explored.

Concolic execution [19, 32, 9] is an analysis technique that performs a concrete execution on random inputs and it 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 concolic execution, which performs symbolic execution *along* a concrete execution, we use concrete execution to *set-up the environment* for symbolic execution. Furthermore, unlike our approach, the approaches described in [19, 32, 9] use code instrumentation and don't use model checking (that we use for analyzing multithreading systematically). We are not aware of any other techniques that combine symbolic and concrete execution in a way similar to our approach.

Our work is similar in spirit, but different methodologically to other hybrid approaches such as [35, 20]. These works combine abstraction techniques and theorem proving for program analysis and testing and do not address the problem of constraining the environment for unit analysis.

Finally, Symbolic JPF can be viewed as leveraging JPF to enable integration of mixed symbolic and concrete execution with the work on carving differential unit tests from system tests [15].

8. CONCLUSIONS AND FUTURE WORK

We presented a symbolic execution framework for test case generation that implements a non-standard interpreter on top of the JPF model checking tool. We proposed two techniques that use concrete system executions to improve the precision of the symbolic analysis. We described a case study that demonstrated the merits of our techniques.

In the future, we plan to work on a tighter integration of the symbolic execution with the system level simulations and to experiment with different machine learning techniques for correlation analysis. Furthermore, we want to investigate whether some of the constraints generated by the symbolic analysis can also be fed back to the simulator, to help it focus on complex, off-nominal scenarios.

We are actively working on extending the capabilities of our symbolic execution framework and applying it to testing new NASA software (e.g., a Java utility library for trajectory computation). We are also extending the tool's capabilities to generate test sequences for UML statecharts (using JPF's statechart extension).

9. ACKNOWLEDGMENTS

We thank Willem Visser for his early guidance in this project and Hamed Jafari for helping with the translation of the OAE code. We also thank the reviewers for their detailed comments.

10. REFERENCES

[1] A. Acevedo, J. Arnold, and W. Othon. ANTARES: Spacecraft simulation for multiple user communities and facilities. In *AIAA Modeling and Simulation Technologies Conference and Exhibit*, 2007.

[2] P. Ammann, P. E. Black, and W. Majurski. Using model checking to generate tests from specifications. In *Proc. of the 2nd IEEE International Conference on Formal Engineering Methods*, 1998.

[3] M. d'Amorim, C. Pacheco, T. Xie, D. Marinov, and M. D. Ernst. An Empirical Comparison of Automated Generation and Classification. In *Proc. of the 21st ASE*, 2006.

[4] S. Anand, A. Orso, and M. J. Harrold. Type-dependence analysis and program transformation for symbolic execution. In *Proc. of the 13th TACAS Conference*, 2007.

[5] S. Anand, C. S. Păsăreanu, and W. Visser. JPF-SE: A symbolic execution extension to Java PathFinder. In *Proc. of the 13th TACAS Conference*, 2007.

[6] T. Ball. A theory of predicate-complete test coverage and generation, 2004. Microsoft Research Technical Report MSR-TR-2004-28.

[7] T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Proc. 29th Annual ACM Symposium on the Principles of Programming Languages (POPL)*, pages 1–3, 2002.

[8] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. Generating tests from counterexamples. In *Proc. of the 26th International Conference on Software Engineering (ICSE)*, 2004.

[9] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: automatically generating inputs of death. In *Proc. ACM Conference on Computer and Communications Security*, 2006.

[10] The Choco Constraint Solver. `http://choco.sourceforge.net/`.

[11] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Trans. Software Eng*, 2(3):215–222, 1976.

[12] The Daikon invariant detector. `http://groups.csail.mit.edu/pag/daikon//`.

[13] X. Deng, J. Lee, and Robby. Bogor/Kiasan: A k-bounded Symbolic Execution for Checking Strong Heap Properties of Open Systems. In *Proc. of the 21st ASE*, 2006.

[14] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. Research Report 159, Compaq Systems Research Center, 1998.

[15] S. G. Elbaum, H. N. Chin, M. B. Dwyer, and J. Dokulil. Carving differential unit test cases from system test cases. In *Proc. of SIGSOFT FSE*, 2006.

[16] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[17] A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. In *Proc. of the 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 1999.

[18] P. Godefroid. Model checking for programming languages using VeriSoft. In *Proceedings of the 24th Annual ACM Symposium on the Principles of Programming Languages (POPL)*, pages 174–186, Paris, France, Jan. 1997.

[19] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *Proc. PLDI*, 2005.

[20] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. SYNERGY: a new algorithm for property checking. In *Proc. of SIGSOFT FSE*, 2006.

[21] M. P. E. Heimdahl, S. Rayadurgam, W. Visser, D. George, and J. Gao. Auto-generating test sequences using model checkers: A case study. In *Proc. 3rd International Workshop on Formal Approaches to Testing of Software (FATES)*, Montreal, Canada, Oct. 2003.

[22] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with blast. In *Proceedings of the Tenth International Workshop on Model Checking of Software*, volume 2648 of *Lecture Notes in Computer Science*, 2003.

[23] G. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.

[24] H. S. Hong, I. Lee, O. Sokolsky, and H. Ural. A temporal logic based theory of test coverage and generation. In *Proc. 8th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, Grenoble, France, April 2002.

[25] IASolver. `http://www.cs.brandeis.edu/~tim/Applets/IAsolver.html`.

[26] Java PathFinder. `http://javapathfinder.sourceforge.net`.

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

[28] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[29] T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. In *Proc. Static Analysis Symposium*, Santa Barbara, CA, June 2000.

[30] T. Lindholm and F. Yellin. *Java(TM) Virtual Machine Specification*. Prentice Hall, 1999.

[31] T. Menzies and Y. Hu. Data mining for very busy people. *Computer*, 36(11):22–29, 2003.

[32] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *Proc. ESEC/SIGSOFT FSE*, 2005.

[33] W. Visser, C. S. Pasareanu, and R. Pelanek. Test input generation for java containers using state matching. In *Proc. ISSTA*, 2006.

[34] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Proc. of the 11th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 2005.

[35] G. Yorsh, T. Ball, and M. Sagiv. Testing, abstraction, theorem proving: better together! In *Proc. ISSTA*, 2006.