A method for testing software systems based on state design pattern using symbolic execution

Cristina Tudose, Radu Oprişa
Department of Computer Science and Mathematics
University of Piteşti
Str. Targu din Vale 1, 110040 Pitesti, Romania
cristina_ferent@yahoo.com, oprisa_radu@yahoo.com

***Abstract*—The paper reports a new testing method working with state pattern designed software systems. The tests are performed in terms of symbolic execution aiming to identify conditions and values of some input parameters that violate assertions at runtime. The state based architecture of such systems allows a direct mapping of the methods to the transitions of the underlying finite state machine (FSM). In order to identify the methods that contain failing assertions, the Java PathFinder Symbolic Execution framework extension (JPF-SE) is used for an out of context execution of each method. We propose a new algorithm to compute a transition path from the initial state of the system to each faulty transition. The computation is carried out using a backward traversal scheme of the FSM support graph where the JPF-SE symbolically executes each transition of the path. The transition execution performed by JPF-SE yields to the backward propagation of the conditions imposed on the input parameters. The overall capabilities of the proposed algorithm are illustrated with an example.**

***Keywords*-finite state machine; Java PathFinder; software testing; state design pattern; symbolic execution.**

I. Introduction

Symbolic execution [1] is a well-known static program analysis technique and its application to software testing was proposed by J. C. King in [2]. A symbolic execution of a program is an execution where input parameters have symbolic values instead of concrete ones and the values of the program variables are represented by symbolic expressions over the input symbolic values. A symbolic state is represented by the values of the program variables, a path condition and a program counter. The path condition is a boolean formula defined on the symbolic inputs. The program counter defines the next statement to be executed.

If a path condition is unsatisfiable, meaning that there are no concrete values associated with the inputs that satisfy the path condition, then the associated symbolic state becomes unreachable. The outputs resulted from the symbolic execution of a program are conditions on the input variables corresponding to each execution path. The obtained conditions are used to generate concrete values to the input variables. Therefore the obtained test cases cover each execution path of the program.

Symbolic execution was initially used for checking sequential programs with a fixed number of integer variables. There are approaches, like [3], that implement dedicated tools that performs program analysis based on symbolic execution. The Java PathFinder model checker [4] does not require a dedicated tool but instead uses a model checker [5] to explore the symbolic execution tree and other forms of nondeterminism, taking advantage of the model checker's built-in space-exploration capabilities: partial order reduction, symmetry reductions, different search strategies such as depth-first, breadth-first, random search. The approach used by this tool handles complex inputs, such as recursive data structures or arrays of unspecified length via ”lazy initialization” as well as concurrency. A similar tool is the Bogor software model checking framework [6] that uses the same initialization strategy.

The JPF-SE framework [7] does not require the code transformation, as Java PathFinder does. It performs symbolic execution of Java bytecodes.

Some of the main limitations of the symbolic execution such as handling native code or availability of decision procedures could be overcome by combining the symbolic with concrete execution. A tool example that uses this technique is PEX [8]. Scalability is another problem due to the infinite execution tree or to the large number and the large size of the paths that need to be explored. Some techniques mentioned in [9] that try to overcome this problem are: abstraction, compositional reasoning, path merging.

Symbolic execution has many applications in areas like: test case generation, test sequence generation, proving program properties, static detection of runtime errors. This paper uses symbolic execution for generating test sequences that lead to a fault.

Related work regarding the generation of method call sequences belonging to a class includes [10], [11]. Test sequences are generated by enumerating all the possible method sequences, up to some specified size. In this paper we consider a particular set of software systems: the state pattern designed software systems [12], [13]. We try to generate a sequence of method calls that yields to an assert violation by starting from the method in which a fault is

978-0-7695-4153-2/10 $26.00 © 2010 IEEE
DOI 10.1109/SEFM.2010.20

detected and then using a backward traversal scheme of the FSM support graph, until the initial state is reached. We use JPF-SE framework for this purpose.

The paper is structured as follows. Section II presents our algorithm for test cases generation, an example is given in Section III and the conclusions are drawn in Section IV.

II. A NEW METHOD FOR GENERATING TEST SEQUENCES

The state design pattern has the main advantage due to its architecture that the internal state of the system is known at each moment of the execution. An other advantage is that the software system implementation must contain a different method for each transition from the associated FSM. The state of the system is changed only after the execution of a method will be completed. Due to these considerations, the execution of a software system designed with state pattern is a sequence of method calls and the order is determined by the associated FSM support graph. In the execution flow, some of the output values of each method are used as input values for the following method. Several execution paths can be identified for a system. Each method can have preconditions and postconditions implemented in the source code. Further we consider preconditions and postconditions implemented as assert instructions at the beginning and at the end of the method body.

By using the symbolic execution, we have the opportunity to analyze each method of the system by out of context symbolic executing it. The out of context execution of a method has the advantage of testing it in all the possible use cases and all the possible input values which is more general than executing the method in the context using the restricted set of values provided only by the previous methods output values from existing sequence method calls. The out of context execution of a method with input values that will never be provided during usual system execution could be considered useless, but there are some advantages on doing this. One of them is the opportunity to check if the implemented preconditions are strong enough for the method. If they are not strong enough, then the symbolic execution will reveal input values that are not handled correctly inside the method and will produce errors. An other advantage is a long term one and it concerns the future extension of the software system. By adding new states and new methods in the system, we observe that some of the old methods can deal with new input values, produced by the new methods. Avoiding further errors is done by implementing the right precondition in the source code from the early stages of the system development. In the next section, we search the methods containing errors by using out of context symbolic execution. Using an algorithm, we select only those methods capable of prodicing errors when the system is executed in the usual way starting from the initial state. For each error, input test values for the system and execution path are revealed.

The testing method starts by executing symbolic each method from the software system using JPF-SE. For some methods, JPF-SE will identify the values of the input parameters that generate assert violations. All these methods are stored in a collection in order to be analyzed.

Let M_T be a method in which a fault is detected corresponding to transition T in the underlying FSM and let P_T be the precondition of M_T, if defined. By default $P_T = True$. We try to find an execution path from the initial state of the system and a set of values for the input parameters that will generate at runtime the expected assert violation in M_T. The approach is to use a backtracking algorithm starting from M_T to traverse the FSM support graph G in reverse transition orientation order, looking to reach the node corresponding to the initial state of the FSM. The selection of the edges is conditioned by the result of the symbolic execution on the corresponding method in the following way. After M_T is symbolic executed, a condition on input values C_T is determined in order to violate an assertion in M_T. The preceding transition method M_{T^2} is determined together with a condition C_{T^2} on the values of the input parameters values such as, at the end of its execution, C_T will be satisfied. In this way, when C_{T^2} is satisfied and path $\{M_{T^2}; M_T\}$ is executed, we enforce that the intermediate C_T is satisfied, which will lead to assert violation in M_T. In practice, in order to find C_{T^2} we instrument the source code by adding at the end of M_{T^2} an assert with the negation of C^T. The instrumented M_{T^2} is executed using the JPF-SE and C_{T^2} is determined so that the assertion $!C^T$ is violated. If the condition C_{T^2} violates the assertion $!C^T$, then C_{T^2} enforces C_T. The preconditions of M_T are appended in the condition C_T. The path search is continued until the start state is reached or no path is found as the following algorithm presents. During the search, all possible paths are practiced, including cycles, but no more than one time. Symbolic execution requires only one traversal of the code in order to explore all the execution paths and compute the path conditions.

```
1: $T^1 \leftarrow T$, $Path \leftarrow \{T^1\}$, $k \leftarrow 2$
2: Execute symbolic $M_{T^1}$ to find condition $C_{T^1}$ that
   violates assertion in $M_{T^1}$
3: **while** $k > 1$ **do**
4:     **while** $(\exists)T_i^k$ preceding $T^{k-1}$ in $FSM$ **do**
5:         $A_{T_i^k} \leftarrow !(C_{T^{k-1}} \wedge P_{T^{k-1}})$
6:         Add assertion $A_{T_i^k}$ at the end of $M_{T_i^k}$
7:         Execute symbolic $M_{T_i^k}$ to find condition $C_{T_i^k}$ that
           violates $A_{T_i^k}$
8:         **if** $C_{T_i^k} \neq FALSE$ and $\{T_i^k; Path\}$ do not con-
           tains any cycle more than one time **then**
9:             $T^k = T_i^k$
10:            $Path \leftarrow \{T^k; Path\}$
11:            **if** $Path$ contains $Start$ state **then**
12:                $Solution \leftarrow C_{T^k}$, STOP
```

```
13:         else
14:             $k \leftarrow k + 1$
15:         end if
16:     end if
17:   end while
18:   Remove $T_k$ from $Path$
19:   $k \leftarrow k - 1$
20: end while
```

If a solution is found, the algorithm generates a condition on the values of the input parameters and an execution path that reveals the expected assertion failure in the system.

III. Example

We consider a simplified class Calculator that contains the following methods: *Clear(), EnterTerm (int T), DoubleIt(), EnterMinus(), EnterPlus(), Compute()*. A Calculator object can be in one of the several different states. The associated finite state machine is given in Figure 1. For simplicity, the $Error$ state and all the incoming and outgoing transitions, that make the FSM completely specified, are not displayed in the figure. When a Calculator object receives the method invocation calls, it responds differently depending on its current state. For example, a *Compute* request depends on whether the object is in its *Acc2Minus* state or *Acc2Plus* state. The State pattern describes how the Calculator can exhibit different behavior in each state.

Implementing the state pattern requires:

- **Context** (*Calculator*) - Defines the interface of the interest to the clients and maintains an instance of a ConcreteState subclass that defines the current state.
- **State** (*CalculatorState*) - Defines all the methods that depend on the state of the object.

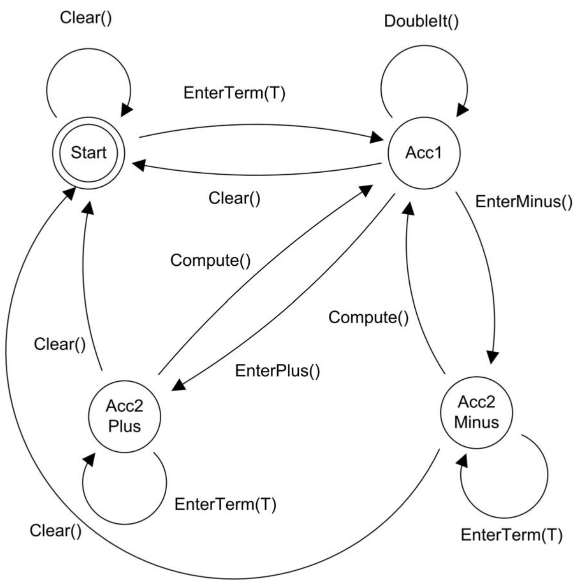

Figure 1. Calculator Finite State Machine

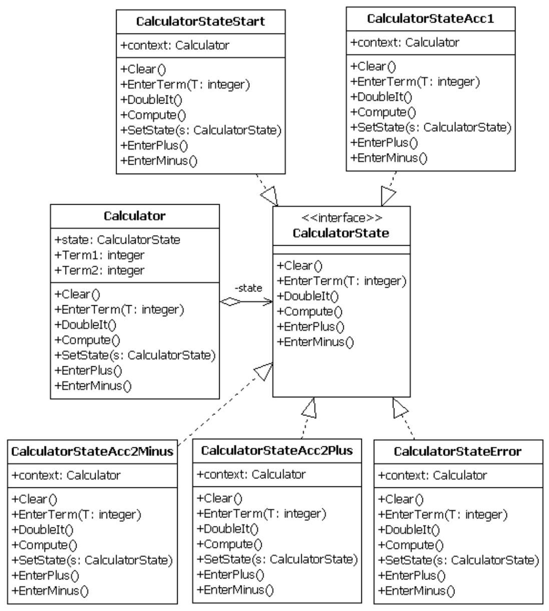

Figure 2. Calculator class diagram

- **ConcreteState** subclasses:
  - *CalculatorStateStart*
  - *CalculatorStateAcc1*
  - *CalculatorStateAcc2Plus*
  - *CalculatorStateAcc2Minus*
  - *CalculatorStateError*

  Each subclass implements a behavior associated with a state of the Context.

The state based architecture of such systems allows a direct mapping of the methods to the transitions of the underlying FSM.

The class diagram for our state pattern implementation is given in Figure 2. We use StarUML [14] tool to draw it.

In our example, the method in which a fault is detected is M_{T^1} = *DoubleIt()* from *CalculatorStateAcc1* class. Figure 4 illustrates its implementation. It is easy to see that *DoubleIt()* computes the double value of $Term1$ but the result is mathematically correct only for the positive integers input values. This is easily revealed by the JPF-SE when the method is executed symbolically and assertion is violated. The algorithm is applied for M_{T^1} and an execution path, from the initial state, is found together with a set of input values which reveal assert violation at runtime. The corresponding method call sequence is: *EnterTerm()*; *EnterMinus()*; *EnterTerm()*; *Compute(); DoubleIt()*.

The methods, the obtained conditions and the preconditions (where they are specified) at each execution step are given in Figure 3.

The generated concrete input values for $T1$ and $T2$ that produce the assert violation in M_T are $T1 = 0$ and $T2 = 1$.

M_{T^1}	***DoubleIt()*** from *CalculatorStateAcc1* class
$assert(A_{T^1})$	$assert(context.Term1 == T1 + T1);$
JPF	$T1\_1\_SYMINT[-10000]! = (T1\_1\_SYMINT[-10000] + T1\_1\_SYMINT[-10000])\&\&$
	$T1\_1\_SYMINT[-10000] <= CONST\_0$
C_{T^1}	$T1 \neq (T1 + T1) \quad \&\& \quad T1 \leq 0$
M_{T^2}	***Compute()*** from *CalculatorStateAcc2Minus* class.
$assert(A_{T^2})$	$assert(!((context.Term1! = (context.Term1 + context.Term1))\&\&(context.Term1 <= 0)));$
JPF	$(T1\_1\_SYMINT[-10000] - T2\_2\_SYMINT[-9999]) <= CONST\_0\&\&$
	$(T1\_1\_SYMINT[-10000] - T2\_2\_SYMINT[-9999])! =$
	$((T1\_1\_SYMINT[-10000] - T2\_2\_SYMINT[-9999])+$
	$(T1\_1\_SYMINT[-10000] - T2\_2\_SYMINT[-9999]))$
C_{T^2}	$(T1 - T2) \leq 0 \quad \&\& \quad (T1 - T2) \neq ((T1 - T2) + (T1 - T2))$
M_{T^3}	***EnterTerm(T2)*** from *CalculatorStateAcc2Minus* class.
P_{T^3}	$T2 \geq 0$
$assert(A_{T^3})$	$assert(!((context.Term1 - context.Term2 <= 0)\&\&((context.Term1 - context.Term2)! =$
	$(context.Term1 - context.Term2) + (context.Term1 - context.Term2))));$
JPF	$(T1\_1\_SYMINT[-10000] - T2\_2\_SYMINT[-9999])! =$
	$((T1\_1\_SYMINT[-10000] - T2\_2\_SYMINT[-9999])+$
	$(T1\_1\_SYMINT[-10000] - T2\_2\_SYMINT[-9999]))\&\&$
	$(T1\_1\_SYMINT[-10000] - T2\_2\_SYMINT[-9999]) <= CONST\_0$
C_{T^3}	$(T1 - T2) \neq ((T1 - T2) + (T1 - T2)) \quad \&\& \quad (T1 - T2) \leq 0$
M_{T^4}	***EnterMinus()*** from *CalculatorStateAcc1* class.
$assert(A_{T^4})$	$assert(!((((context.Term1 - context.Term2)! =$
	$((context.Term1 - context.Term2) + (context.Term1 - context.Term2)))\&\&$
	$(context.Term1 - context.Term2 <= 0))\&\&(context.Term2 >= 0)));$
JPF	$T2\_2\_SYMINT[0] >= CONST\_0\&\&(T1\_1\_SYMINT[-10000] - T2\_2\_SYMINT[0]) <=$
	$CONST\_0\&\&$
	$(T1\_1\_SYMINT[-10000] - T2\_2\_SYMINT[0])! =$
	$((T1\_1\_SYMINT[-10000] - T2\_2\_SYMINT[0])+$
	$(T1\_1\_SYMINT[-10000] - T2\_2\_SYMINT[0]))$
C_{T^4}	$T2 \geq 0 \quad \&\& \quad (T1 - T2) \leq 0 \quad \&\& \quad (T1 - T2) \neq ((T1 - T2) + (T1 - T2))$
M_{T^5}	***EnterTerm(T1)*** from *CalculatorStateStart* class.
P_{T^5}	$T1 \geq 0$
$assert(A_{T^5})$	$assert(!((context.Term2 >= 0)\&\&(context.Term1 - context.Term2 <= 0)\&\&$
	$((context.Term1 - context.Term2)! = ((context.Term1 - context.Term2)+$
	$(context.Term1 - context.Term2)))));$
JPF	$(T1\_1\_SYMINT[0] - T2\_2\_SYMINT[1])! =$
	$((T1\_1\_SYMINT[0] - T2\_2\_SYMINT[1]) + (T1\_1\_SYMINT[0] - T2\_2\_SYMINT[1]))\&\&$
	$(T1\_1\_SYMINT[0] - T2\_2\_SYMINT[1]) <= CONST\_0\&\&$
	$T2\_2\_SYMINT[1] >= CONST\_0\&\&T1\_1\_SYMINT[0] >= CONST\_0$
C_{T^5}	$(T1 - T2) \neq ((T1 - T2) + (T1 - T2)) \quad \&\& \quad (T1 - T2) \leq 0 \quad \&\& \quad T2 \geq 0$
$Solution$	$C_{T^5}; (T1 = 0, T2 = 1)$

Figure 3. Step by step symbolic execution

```
public void DoubleIt()
{
int T1 = context.Term1;
int T2 = context.Term2;
if (context.Term1 > 0)
{
context.Term1 = context.Term1 + context.Term1;
}
assert(context.Term1 == T1 + T1);
context.SetState( new CalculatorStateAcc1(context));
}
```

Figure 4. *DoubleIt()* method source code

By following the generated method call sequence, an unit test can be produced:

```
Calculator c = new Calculator();
c.EnterTerm(0);
c.EnterMinus();
c.EnterTerm(1);
c.Compute();
c.DoubleIt();
```

A consideration for using the proposed method is the well known issue of the big amount of computing resources, especially the computer memory used by the symbolic execution process. Symbolic execution needs to store for further investigation all the branches from the execution path, together with the corresponding path conditions. For medium and large systems, the symbolic execution of the whole source code might become impossible. The current approach uses the symbolic execution only at the method level fact that requires less resources for the symbolic execution process.

IV. Conclusions and future work

The paper presents a new method for generating test cases using symbolic execution for the state pattern designed software systems. In contrast with other approaches, where the error search is started from the initial state of the system, this new method for generating test sequences identifies the methods in which a fault is detected from the system and computes, using a backtracking algorithm, a path back to the initial state of the system. The problem is reduced to a graph traversal assisted by the results of the symbolic execution at each step. JPF-SE is used to execute the methods and to determine the values of the input parameters that are able to violate the assertions in the source code at runtime. If an execution path is found, JPF-SE generates conditions on the input parameters and concrete values for them, that can be used to define unit tests.

For future work, we plan to generalize the study over software systems implementations, other than those based on the state design pattern. For this purpose, we will try to use call graphs as support instead of the FSM support graph and to instrument the methods source code with proper assertions before each method call.

References

[1] C. S. Pasareanu and W. Visser, "A survey of new trends in symbolic execution for software testing and analysis," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 11, no. 4, pp. 339–353, october 2009.

[2] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[3] A. Coen-Porisini, G. Denaro, C. Ghezzi, and M. Pezze, "Using symbolic execution for verifying safety critical systems," in *Proceedings of ESEC/FSE*, 2001.

[4] "Java PathFinder." [Online]. Available: http://babelfish.arc.nasa.gov/trac/jpf

[5] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model checking*. Cambridge, MA, USA: MIT Press, 1999.

[6] "Bogor." [Online]. Available: http://bogor.projects.cis.ksu.edu/

[7] S. Anand, C. S. Pasareanu, and W. Visser, "JPF-SE: A symbolic execution extension to Java PathFinder," in *Proceedings of TACAS*, 2007.

[8] "PEX." [Online]. Available: http://research.microsoft.com/en-us/projects/pex/

[9] C. S. Pasareanu and W. Visser, "A survey of new trends in symbolic execution for software testing and analysis," *Int. J. Softw. Tools Technol. Transfer*, vol. 11, no. 4, pp. 339–353, 2009.

[10] W. Visser, C. S. Pasareanu, and R. Pelanek, "Test input generation for java containers using state matching." in *Proceedings of ISSTA*, 2006.

[11] T. Xie, D. Marinov, W. Schulte, and D. Notkin, "Symstra: A framework for generating object-oriented unit tests using symbolic execution." in *Proceedings of TACAS*, 2005.

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

[13] S. Stelting and O. Maassen, *Applied Java Patterns*. Upper Saddle River, New Jersey: Prentice Hall, 2002.

[14] "StarUML." [Online]. Available: http://staruml.sourceforge.net