An Eclipse-based environment for conformance testing by FSMs

Angelo Gargantini, Marco Guarnieri, and Eros Magri

Dip. di Ing. dell'Informazione e Metodi Matematici, Università di Bergamo, Italy
`angelo.gargantini@unibg.it,0guarnieri.marco0@gmail.com,erosmagri@gmail.com`

Abstract. Finite state machines (FSMs) represent a very simple yet widely used formalism. They are used to model simple protocols or even complex systems in an abstract way. Java classes often behave like FSMs. For these reasons, FSMs are often used in conformance testing, which consists in checking that a real implementation conforms with its specification given in terms of a FSM. In this paper, we show how Eclipse-related technologies, like GMF and PDE, can be used to build an editor and test generator for FSMs. We present a complete environment which contains: (1) a graphical editor based on a metamodel for FSMs, (2) an extensible framework for test generation from FSMs according to several coverage criteria, and (3) an API that can be used to test Java classes against their specifications given as FSMs.

1 Introduction

Finite State Machines (FSMs) are a simple notation which permits the description of systems with a finite number of states and simple transitions between them. Although they are quite simple, they are frequently used to model systems or system parts, also because they provide an appealing and intuitive graphical notation. Even the behavior of very complex systems is often represented in an abstract way by means of FSMs. FSMs have been used to model protocols, sequential circuits, and parts of programs. In particular, Java classes which have a simple behaviour leading to a finite number of states can often be modeled by FSMs, for instance by using the well known *state pattern* [9]. Lately, several authors proposed to extract finite state machines from source code [8].

For these reasons, FSMs are widely used in conformance testing [11]. In conformance testing, one assumes that the specification S is given in terms of a FSM and wants to check whether the implementation I behaves like its specification, i.e. I conforms to S. The specification is built by the designer, and a test sequence (or a set of test sequences) is derived form it in order to check if the implementation behaves as prescribed by the test sequences (called *checking sequences*).

In conformance testing with FSMs, we address the following issues. First, although there is standard mathematical definition of FSMs, there is no standard tool for designing them. Some tools which are used in practice, do not offer graphical editors (as the State Machine Compiler [3]). We investigate the use of the Eclipse framework, in particular GMF, to build a prototype editor for FSMs. The desired editor must feature a graphical notation similar to that already defined for the FSMs and possibly be integrated in the IDE used daily by the developers. Second, we want to introduce a framework for test generation. Since there are plenty of algorithms for test generation from FSMs [10,11], the framework must allow to add new algorithms in an incremental way. To this aim, we investigate the use of the Eclipse plugin framework in which test generation algorithms can be introduced as extensions of particular extension points. Third, we assume that the test sequences may be used to test Java code. We address the issue of using the test sequences generated to check the conformance of a Java class. In this paper, we assume that the user has a certain access to the source code and he/she may be able to slightly modify it in order to ease the conformance testing.

The paper is structured as follows. Section 2 presents the notation of finite state machines used in this paper, namely the Mealy FSMs. Section 3 gives an overview on how we created the editor using the GMF framework, while Section 4 presents how we introduced the test generation algorithms. Finally Section 5 presents the architecture of the tool. Section 6 presents related works and Section 7 presents our conclusions and ideas for future works.

2 Background

A Mealy machine M is a finite state machine whose outputs depend on the current state and on the input received. It is defined as the following tuple $M = \langle I, O, S, s_0, \delta, \lambda \rangle$, where, more in detail,
I is the set of input symbols,
O is the set of output symbols,
S is the set of states,
s_0 is the initial state and it belongs to S
δ **:** $\mathbf{S} \times \mathbf{I} \longrightarrow \mathbf{S}$ is the state transition function,
λ **:** $\mathbf{S} \times \mathbf{I} \longrightarrow \mathbf{O}$ is the output function,
and *I, O, S* are all finite non empty sets. The transition and output functions describe the FSM behavior, that is, when a the FSM M is currently in a state $s \in S$ and it receives an input $i \in I$, it moves to the new state $s' \in S$ where $s' = \delta(s, i)$, producing the output $o \in O$ where $o = \lambda(s, i)$. A FSM can be represented also as a state transition diagram, that is a graph in which vertexes represent states, edges represent transitions and each transition can be labeled with an input and output symbols. Fig. 1 depicts an instance of FSM (used in the following also for test generation) with an intuitive graphical notation.

Given two input or output sequences a and b, we use the notation $a.b$ to identify the result of the concatenation of a with b. We define an input sequence $i^* = i_1.i_2. \ldots .i_k$ of input symbols and an output sequence $o^* = o_1.o_2. \ldots .o_j$ of output symbols. A variation of the state transition function is $\delta^* : S \times I^* \longrightarrow S$ that takes as input the current state s_1 and i^* that moves M successively to $s_{i+1} = \delta(s_i, i_i)$ for each input into the sequence i^*. We can also define the

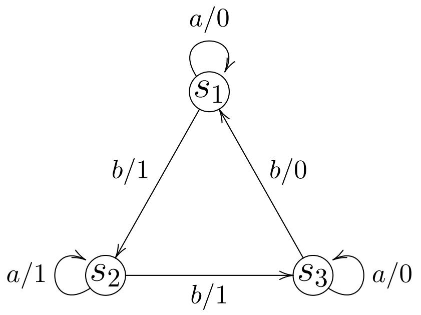

Fig. 1. An instance of FSM M (taken from [11])

function $\lambda^* : S \times I^* \longrightarrow O^*$ that takes as input the current state s_1 and i^* and returns the associated output sequence. Another way to represent a Mealy FSM is using a tuple $M = \langle I, O, S, T \rangle$ where I, O, S have the same meaning as above and T represents the set of transitions, where each transition is defined as the tuple $< s, i, o, s' >$ where s is the source state, *s'* is the target state, i and o are respectively the input and output symbols. We use this definition when introducing the metamodel for the FSMs.

A FSM M may also have a *status* message, which is a particular input which signals as output the current state of the machine and it does not move the machine from the current state. Moreover, a FSM may also have a *reset* message, which moves the machine to its initial state. Classical FSMs have no final states.

A classical use of FSMs is for conformance testing [12], which can be described as follows. Given two different FSMs, a first one M_s, that acts as specification and for which we known the state transition diagram, and a second one M_i, that acts as implementation of the first one and for which we can only observe its behavior, we want to test if M_i correctly implements M_s, that is we want to detect defects into the implementation M_i with respect to M_s. The two classical kinds of failures considered in this paper are:

transfert fault : a transition t in M_s that starts from s_s and ends on s_e is implemented in M_i with a transition t' that starts from s_s and ends on s'_e where $s_e \neq s'_e$,

output fault : a transition t in M_s that starts from s_s and ends on s_e with an output o is implemented in M_i with a transition t' that starts from s_s and ends on s_e with an output o', where $o \neq o'$.

The testing process is divided into three phases. In the first one we generate an input sequence i^* from M_s and compute the expected output sequence o^* by applying i^* to M_s, while in the second phase we test the implementation M_i, applying to it the input sequence i^* and obtaining the real output sequence o'^*. In the last phase, in order to check that the implementation M_i conforms to the specification M_s, we compare the real output sequence o'^* and the expected output sequence o^*; if o'^* is equal to o^*, then the implementation M_i conforms to the specification M_s.

In the following, we introduce some assumptions about M_s and M_i. First we assume that M_s is minimal, that is for every pair of states s and s' into M_s must exists a sequence of inputs i^* for which $\lambda(s, i^*) \neq \lambda(s', i^*)$. We also assume that M_s is strongly connected and completely specified, that is for each state $s \in S$ and for each input $i \in I$ both the function $\lambda(s,i)$ and $\delta(s,o)$ are specified. The last assumption is that M_i doesn't change during the testing and that the set of input I of M_i is equal to the one of M_s.

The classical test generation algorithms implemented into our tool are [10]:

State cover set: This method provides only state coverage and it generates the testing sequence simply visiting the FSM graph using the Dijkstra's algorithm and adding *status* message after each input.

Transition Tour: This method provides both state and transition coverage. It uses transition tour, that is a sequence of state transitions that begins and ends on the initial state. We assumed that the FSM is strongly connected and thus we can compute one transition tour that visits all the transitions in the machine, by simply solving the *Direct Chinese Postman Problem* [14] on the graph of the finite state machine. Once that we have computed the sequence of inputs that provides the transition tour we put a *status* message after each input to obtain the testing sequence.

W method: This method can be used when the FSM doesn't support the *status* message but it supports only the *reset* message. Instead of the *status* message we use a *characterizing sequence* extracted from a *characterizing set*, that is a set of input sequences such that for every pair of distinct states s and t there exists an input sequence x^* for which $\lambda(s, x^*) \neq \lambda(t, x^*)$. The method uses the characterizing set and the transition cover set, that is a set of input sequences such that for each state $s \in S$ and each input $i \in I$, there exist two input sequences x^* and y^* such that $x^* = y^*.a$ and $\delta(s_i, y^*) = s$ (where s_i is the initial state of the FSM), to build the testing sequence.

WP method: This method computes the testing sequence in two different phases. Into the first one the input for the FSM consists into the concatenation of a *state cover set* with a *characterizing set*, while for the second phase it use a *identification sets*. An *identification set* W_i for the state s_i is a set of input sequences such that for each state $s_j \in S$(where $i \neq j$) exists an input sequence x^* into W_i such that $\lambda(s_i, x^*) \neq \lambda(s_j, x^*)$ and no subset of W_i has this property. The main advantage of the *Wp method* is that it can build a shorter testing sequence than *W method*.

3 Building an Editor for FSMs

The first goal of our project is to build a prototype graphical editor for FSMs.

We base the construction of the Editor on the Graphical Modeling Framework (GMF) that is a framework within Eclipse, that provides tools for developing graphical editors based on the Eclipse Modeling Framework (EMF) and Graphical Editing Framework (GEF). Starting point is a model specification, called metamodel, from which Eclipse Modeling Framework generates a set of Java classes for the model and finally a basic editor.

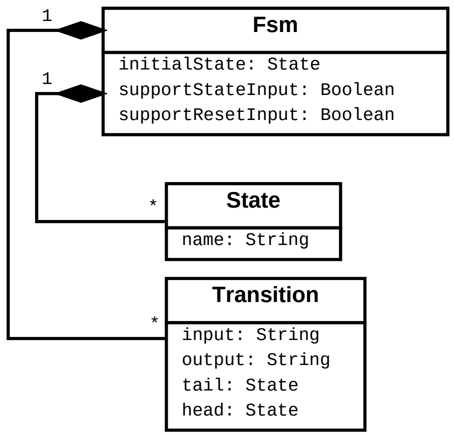

Fig. 2. FSM Metamodel

The metamodel represents the model for the language for which the editor is built. It describes the language in an abstract way by means of a class diagram (in a classical Object Oriented way). Figure 2 shows the metamodel for FSMs. To be more precise, the metamodel defines the following entities:

– `State:` represents the states of the FSM and has attribute:
  • `name` type *String* which is the unique name assigned to the state

– `Fsm:` it contains the main features of the finite state machine and owns (by composition) a set of States and Transitions. Its main attributes are:
  • `initialState` of type *State*, identifies the initial state of the machine
  • `supportStateInput` of type *Boolean*, contains the information that the machine supports or not the *state* input.
  • `supportResetInput` of type *Boolean*, contains the information that the machine supports or not the *reset* input.

– `Transition` represents a transition and its attributes are:
  • `input` type *String* containing the input value of the transition
  • `output` type *String* which is the output value of the transition
  • `head` type *State* identifies the state where the transaction starts
  • `tail` type *State* identifies the state where the transaction comes

The metamodel is also called *domain model* and after creating it, we define and build:

– Domain Generation Model, enabling the generation of all the editor code.
– Graphical Definition Model, which allows to establish what are the essential components of the diagram.
– Tooling Definition Model, defining the tools for drawing the diagram.
– Mapping model, that links the Domain Model, the Graphical Model and the Tooling Model.
– Diagram Editor Generation Model, that generates the GMF graphical editor, in addition to the code generated with the Domain Model.

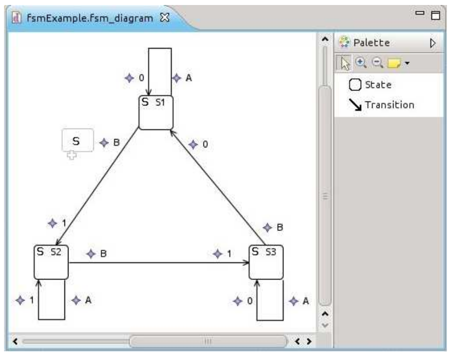

Fig. 3. Editor's screenshot

The graphical editor obtained allows the user to easily draw FSM charts, using its two tools:

- "State" which allows to draw a state with its name
- "Transition" which allows to draw a transition between two states with the respective values of input and output

The properties of the FSM and of States and Transitions can be modified by using the properties view. Figure 3 shows a screenshot of the editor.

4 Test generator

We choose to implement the test generator part of our tool as an Eclipse plugin for two reasons: (1) we want to integrate design, implementation, and testing of FSMs into the same environment and (2) the developer may extend the test generation by introducing new algorithms as eclipse plugins. In this way, our tool and Eclipse can provide the developer with an environment to work with FSMs, from the beginning to the end of the development lifecycle, into the same application, avoiding switching between different applications.

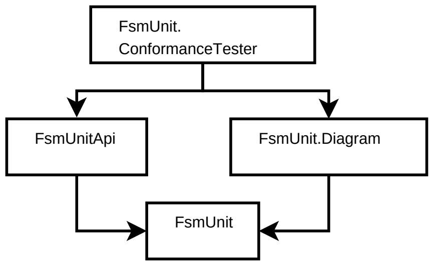

Fig. 4. Components Diagram

Our application is composed of several components, some of them provide only the model of the FSM and the different test generators and others that provide the user interface and the graphical editor. More in detail, as shown in Figure 4, the components are:

FsmUnit: This component contains the definition of the EMF model.
FsmUnit.Diagram: It implements the FSMs editor, as explained in the previous section.

FsmUnitAPI: It contains the implementations of the algorithms that generate the test sequences.

FsmUnit.ConformanceTester: uses both *FsmUnitAPI* and *FsmUnit.Diagram* to integrate the test generation algorithms into the graphical user interface. This component is presented in Section 5.

All the test generators for the methods introduced in Section 2 are implemented in the *FsmUnitAPI* component. This component exposes the following interfaces:

IFsmBuilder: represents the interface of a builder that allows the user to build a FSM in an incremental way.

IVisitor: represents the interface of a visitor that can visit an IFsm object and all the entities into the model.

ITestGenerator: represents the interface of the test generators, both for state and transition coverage. It declares a method to compute the sequence of inputs and another one to compute the list of expected outputs. All the test generators were implemented following the *Strategy* pattern.

Our architectural choice presents more than one advantage. A first one is that separating the API from the graphical user interface, we provide a solution that can be implemented using different editors to build Finite State Machines (another solution could be, for example an editor based on *xText*). The API component provides developers with a logical model to implement and work with FSMs. Another advantage is that our solution provides a very easy way to add new test generators, using algorithms and criteria different from the ones already implemented into the API.

To add new test generators, we create the *FsmUnitAPI.coverageCriteria* extension point.

Listing 1. Definition of FsmUnitAPI.coverageCriteria Extension Point

```
<element name="criteria">
  <complexType>
    <attribute name="class" type="string" use="required">
        <meta.attribute kind="java"
            basedOn=":it.fsmunitapi.fsm.testgenerators.ITestGenerator"/>
    </attribute>
    <attribute name="description" type="string" use="required"/>
    <attribute name="name" type="string" use="required"/>
    <attribute name="kindOf" use="required">
      <simpleType>
          <enumeration value="stateCoverage"/>
          <enumeration value="transitionCoverage"/>
      </simpleType>
    </attribute>
  </complexType>
</element>
```

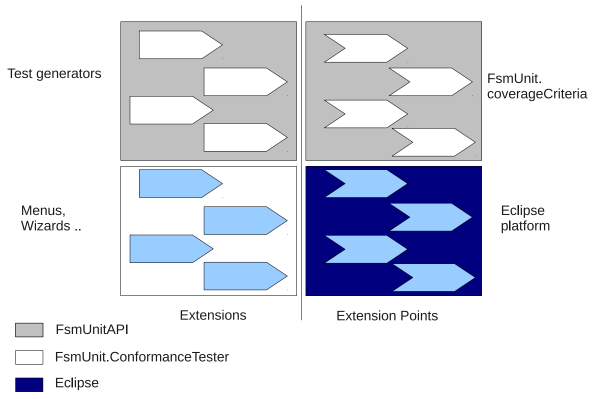

Fig. 5. FsmUnitAPI provides its extension point to declare new test generators. FsmUnit.ConformanceTester uses Eclipse's extension points for the GUI.

It has four different attributes, as shown in the listing 1:

Class : it is the full qualified name of the class that implements the test generator. This class must implement the *ITestGenerator* interface.
Description: it is a brief description of the test generation algorithm. This description will be shown into the preference page.
Name: contains the name of the test generator.
KindOf: it is the kind of coverage criteria. If its value is "stateCoverage" the generator provides only state coverage, if it's "transitionCoverage" the generator provides transition coverage, and obviously also state coverage.

Listing 2 shows as example of the use of the CoverageCriteria extension point, the extension used to add the test generator that implements the W Method algorithm.

Listing 2. Example of use of FsmUnitAPI.coverageCriteria Extension Point

```
<criteria
  class="it.fsmunitapi.fsm.testgenerators.internal.WMethodTestGenerator"
  description="Transition coverage with W Method."
  kindOf="transitionCoverage"
  name="WMethodTestGenerator">
</criteria>
```

The Figure 5 shows the complete structure of extensions and extension points. Both our components *FsmUnitAPI* and *FsmUnitDiagram* are Eclipse plugins, and the first one provides the extension point *FsmUnitAPI.coverageCriteria*.

Into our tool we provide the implementation of four test generation algorithms that uses the above-mentioned extension point: (i) State cover set (ii) Transition Tour (iii) W method (iv) Wp method.

Number	Input sequence	Expected output sequence	Number	Input sequence	Expected output sequence
1	r.A	0	8	r.B.B.A.B	1.1.0.0
2	r.B	1	9	r.B.A.A	1.1.1
3	r.A.A	0.0	10	r.B.A.B	1.1.1
4	r.A.B	0.1	11	r.B.B.A	1.1.0
5	r.B.A	1.1	12	r.B.B.B	1.1.0
6	r.B.B	1.1	13	r.B.B.B.A	1.1.0.0
7	r.B.B.A.A	1.1.0.0	14	r.B.B.B.B	1.1.0.1

Table 1. Testing sequences

Table 1 reports the sequence of inputs and expected outputs that our prototype computes using the *W method* on the FSM presented before in the Figure 1. In the table, we use the input r to represent the *reset* input.

5 Conformance Unit Testing

The component *FsmUnit.ConformanceTester* integrates the *FsmUnitAPI* and the *FsmUnit.Diagram*. It adds a simple menu where the user can recall the various features of the plugin and a section in the preferences of eclipse in order to change the appearance of the editor and choose the algorithm used to perform the desired test coverage.

The developer, during design phase, draws the graph of the Finite State Machine S. He can then generate a testing sequence that can be used during the testing phase to prove that an implementation conforms to a FSM specification S. To perform the conformance testing our tool provides three different approaches.

1. The first approach consists in testing if another FSM conforms to the original one S, by simply executing the testing sequence on it. So, first the user builds another model of the FSM I, then he/she executes on it the testing sequence and finally he/she compares the expected output sequence and the real output sequence. If the two sequences are equal, than I conforms to S.

2. A second approach, that we call *FsmUnit test*, consists in executing the testing sequence at runtime on a *".class"* file that contains the bytecode of a class that implements S. First we dynamically load the class, then we execute on it the testing sequence and we compare real output sequence and the expected output sequence. If the two sequences are equal, the implementation conforms to the specification. Note that in this case, the test sequence obtained from the FSM which is *abstract* (i.e. it does not depend on the implementation) must be translated into a *concrete* test sequence in some Java code.

3. The third approach is to create a JUnit test that check if a class that implements the Finite State Machine conforms to the specification. As the two cases mentioned before if the expected output sequences equals the real output sequence the implementation conforms to the specification.

If the developer wants to apply approaches 2 and 3, the class under test must implement *ITestableFsm* and satisfies the contracts defined in it. *ITestableFsm* signals that the class implements a Finite State Machine and it is provided in the *FsmUnitAPI* component.The interface provides the contract of four methods:

void init() to initialize the implementation of the Fsm,

void executeInput(IInput input) that executes the input, moves the Fsm to a new state and generates a new output. This method translates the *input* of the FSM to the right method call of the class under test.

List<IOutput> executeInputSequence(List<IInput> seq) that executes a sequence of inputs,

IOutput getLastOutput() that returns the last generated output.

Number	Input sequence sequence	Expected output sequence	Obtained output sequence	Number	Input sequence sequence	Expected output sequence	Obtained output sequence
1	r.A	0	0	8	r.B.B.A.B	1.1.0.0	1.**0**.0.0
2	r.B	1	1	9	r.B.A.A	1.1.1	1.1.1
3	r.A.A	0.0	0.0	10	r.B.A.B	1.1.1	1.1.**0**
4	r.A.B	0.0	0.0	11	r.B.B.A	1.1.0	1.**0**.0
5	r.B.A	1.1	1.1	12	r.B.B.B	1.1.0	1.**0**.0
6	r.B.B	1.1	1.**0**	13	r.B.B.B.A	1.1.0.0	1.**0**.0.0
7	r.B.B.A.A	1.1.0.0	1.**0**.0.0	14	r.B.B.B.B	1.1.0.1	1.**0**.0.1

Table 2. Test results

Basically, the developer has two different ways to implement the *ITestableFsm* interface. The first one is to generate a new class only for testing purposes that wraps the preexistent class and maps all the inputs and outputs from the methods of the interface to the methods of the real class. While the first approach is useful if one wants to add the support for *FsmUnit* testing to an existent class, a better approach is to develop the class directly using the methods provided by the *ITestableFsm* interface and integrate them into the class.

For instance, if we write a Java class that implements the FSM presented in Figure 1 and we want to test it, we need only to implement the interface *ITestableFsm*, provided by the *FsmUnitAPI* component, and in order to run a *FsmUnit* test on it, we only need to select the class and apply to it the testing sequence.

Suppose that our implementation has an output fault on the transition from state *S2* to state *S3* and we have as result of the test the output sequence in the Table 2. It can be seen quite clearly that the implementation of the class has a fault because the expected output sequence is not equal to the obtained output sequence (the error is in bold). If the test is executed using a FsmUnit test our prototype highlight the fault into the Eclipse IDE, while if the test is executed using the JUnit generated by our tool the fault is detected using JUnit assertions.

6 Related work

Several tools exist for editing and programming with FSMs. State Machine Compiler [3] is a tool that allows developers to define FSMs, using a specific language, and then the tool automatically generates the source code that implement those FSMs. StateForge is another tool [4] that allows developers to define Finite State Machines using XML and to generate automatically the source code that implement them. The Unimod project, [5], provides both a Java FSM metamodel, a tool to validate some simple properties on FSMs and an Eclipse plugin that allows developers to draw FSMs. These tools are different from our tool primarily because they do not offer any support to the conformance testing phase. There exist several academic and research tools that are capable of generating tests from FSMs (or Extended FSM) [6]. However, they require specifications written in specific notations and no one includes an editor for FSM. Most of them are no longer maintained and used. For instance, TSG [1] supports test generation of FSM by several classical algorithms but it is distributed only as binary for an old SunOS. Another frequently cited tool, TAG, [13] is no longer distributed. These tools do not support directly testing of programs. Other more recent tools target also program testing. For instance, ModelJUnit [2,15] is a Java library that extends JUnit to support model-based testing for Finite State Machine. Models are extended finite state machines that are written in Java, following a template which requires the user to implement the Java inteface `FsmModel` in a similar way as our approach explained in Section 5.

7 Conclusions and Future Work

In this paper, we have presented an approach supported by a prototype tool for conformance testing based on Finite State Machines. We have developed a graphical editor for FSMs using GMF, which is based on the definition of a metamodel for FSMs and it is largely automatically generated by creating several GMF artefacts. We found this approach very powerful and convenient in order to build a working prototype for a graphical editor for FSMs. Since several testing criteria have been defined for FSMs and new ones may be introduced, we have exploited the plugin development environment of Eclipse to define a suitable extension point which permits the definition and implementation of several test generation algorithms which can be easily integrated in the tool. We have already implemented several basic classical test generation algorithms which we found in literature. We plan to extend the tool beyond test case generation including support to debug and animate the models. For instance, the results produced by a failed test case could be visualized on the FSM models to facilitate the understanding of the failure.

We have started using our tool for conformance testing of Java classes. In this case the Java code must be modified in order to be executed with the test sequences generated. We plan to extend the use of our tool with Java by defining the links between Java method invocations and FSM inputs in an external file or by annotating the original Java code, together with AspectJ for monitoring the method calls. This approach would minimize the modifications of the original code. A similar approach is taken by the MOP tool for run time monitoring of Java code [7], which however does not tackle the part of generation of tests, since it assumes that the Java code in checked against its specification (given as FSM) during its use.

References

1. TSG:FSM-based test sequence generator. `http://www.site.uottawa.ca/~ural/tsg/`, 1997.
2. modeljunit. `http://www.cs.waikato.ac.nz/~marku/mbt/modeljunit/`, 2011.
3. SMC - the state machine compiler. `http://smc.sourceforge.net/`, 2011.
4. StateForge - finite state machine diagram and code generators. `http://www.stateforge.com/`, 2011.
5. Unimod - executable uml. `http://unimod.sourceforge.net/`, 2011.
6. A. Belinfante, L. Frantzen, and C. Schallhart. *Model-Based Testing of Reactive Systems*, chapter Tools for Test Case Generation, pages 391–438. Number 3472 in Lecture Notes in Computer Science. Springer, 2005.
7. Feng Chen and Grigore Roşu. Java-MOP: A monitoring oriented programming environment for java. In Nicolas Halbwachs and Lenore D. Zuck, editors, *TACAS*, volume 3440 of *Lecture Notes in Computer Science*, pages 546–550. Springer, 2005.
8. James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from java source code. In *Proceedings of the 22nd international conference on Software engineering*, ICSE '00, pages 439–448, New York, NY, USA, 2000. ACM.
9. Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, MA, 1995.
10. Angelo Gargantini. Conformance testing. In Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, editors, *Model-Based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*, pages 87–111. Springer, 2004.
11. D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - A survey. In *Proceedings of the IEEE*, volume 84, pages 1090–1126, 1996.
12. K. Naik and P. Tripathy. *Software Testing and Quality Assurance: Theory and Practice*, chapter Test Generation from FSM Models, pages 265–. John Wiley & Sons, 2008.
13. Q.M. Tan, A. Petrenko, and G. von Bochmann. A test generation tool for specifications in the form of state machines. In *Communications, 1996. ICC 96, Conference Record, Converging Technologies for Tomorrow's Applications. 1996 IEEE International Conference on*, volume 1, pages 225 –229 vol.1, jun 1996.
14. Harold W. Thimbleby. The directed chinese postman problem. *Softw, Pract. Exper*, 33(11):1081–1096, 2003.
15. Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach.* Morgan-Kaufmann, 2006.