Conformance Testing with Abstract State Machines

Wolfgang Grieskamp, Yuri Gurevich, Wolfram Schulte, and Margus Veanes

Microsoft Research, One Microsoft Way, Redmond, WA 98052, USA
{wrwg, gurevich, schulte, margus}@microsoft.com

Abstract. One benefit of executable specifications is that they allow one to test the conformance of implementations to their specifications. We illustrate this on the example of Universal Plug and Play devices. The necessary test sequences are generated automatically from ASM specifications.

1 Introduction

Working with Microsoft's product groups, we began to appreciate how important it is to use specifications for generating test suites. Test suites allow you to check whether the given implementation conforms to the specification. Is the ASM approach good for the purpose? Egon Börger is optimistic [4, New Frontiers]:

> If we succeed to exploit ASMs for defining and implementing methods for generating test suites from high level specifications, this will turn a dark and at present overwhelming part of software development into an intellectually challenging and methodologically well supported task of enormous practical value.

Here we are concerned with conformance testing for devices. The specification of a device is an abstract state machine [15] with a fixed initial state. The ASM may have many, possibly infinitely many, states. In general, you don't have the resources to explore all possible runs of the ASM. To this end, we group the ASM states into finitely many hyperstates. This gives rise to a finite state machine, or FSM, which is then used to generate a test suite using existing techniques. A test suite is a set of test sequences that describe expected input-output behaviors of the device. Exactly what information is encoded in the test sequences depends on the method used to traverse the FSM and the method used to apply the test suite to the device under test. Both issues are well studied in the literature and are outside the scope of this paper. The basic assumption we make here is that the device program text is not available to us.

It is important that the generated FSM and the original ASM are related in a natural way. The runs of the ASM starting from a given initial state can be seen

as a directed graph whose vertices are states and whose edges are labelled by actions that trigger the state transitions.

There are various ways that an ASM program can distinguish between ASM states, the most important of them is by the means of guards. The guards reflect the state distinction that the programmer cared enough about to make it explicit. Any rule involving nested conditional rules can be normalized so that all its guards appear on the top level. Say that two states of the ASM are *guard distinguished* by the specification program if there exists a guard in the unnested form of the program that is satisfied by one of the two states but not by the other. The guard-indistinguished property is an equivalence relation. It is desirable that, during testing, you visit the equivalence class of every reachable state. Toward this goal, we define our hyperstates as equivalence classes of guard-indistinguished states.

Generating a test suite from an FSM naturally raises questions regarding the *coverage* of the FSM. Think of the FSM as a directed graph with labeled edges and a distinguished initial node. What parts of the directed graph does the test suite cover? For example, do you visit every reachable node and thus achieve *node coverage*? Do you visit every link in the reachable part and thus achieve *link coverage*?

We may also look at our ASM specification program text, and consider its *structural* coverage. Syntactically our ASM specification program is given as a set of ASM rules, one rule per every input action. Full *guard coverage* is ensured if the test suite involves, for each guard, some invocation of that rule with the guard being true. In general, the problem of determining whether the given guard may become true in some state is undecidable; see Section 3.1.

We restrict attention to deterministic programs; the generalization to the case of nondeterministic programs is in preparation.

The rest of the paper is organized as follows. Section 2 provides a high-level view of the two-step test-generation problem. The first step is to generate an FSM from an ASM specification; it is treated in Section 3. The second step is to generate test cases from the FSM; it is treated briefly in Section 4. Open problems are discussed in Section 5. Related work is discussed in Section 6. Executable AsmL [13] specification of the algorithm in Section 3 is given in Appendix A.

2 Conformance Testing with ASMs

We are given a device and a specification for the device in the form of an ASM program P of the following normal form. For each input action a in a fixed set *Actions* of input actions, the ASM has a rule P_a called the *action program* for a. Each P_a is a do-in-parallel block of rules

$$
\begin{array}{l}
P_a \texttt{ =} \\
\quad \texttt{if } g_1 \texttt{ then } R_1 \\
\quad\quad \ldots \\
\quad \texttt{if } g_k \texttt{ then } R_k
\end{array}
\tag{1}
$$

where each '`if` g_i `then` R_i' will be called a *clause* (of the action program) with g_i as its *guard* and R_i as its *body*. The guard of a clause is a Boolean valued term (essentially a first-order formula without free individual variables). The body of a clause is nonbranching, i.e., has no if-then-else subrules.

A few words on the normalization. Every sequential ASM program is normalizable [15]. The reducing algorithm generalizes to the case of programs with the do-for-all construct and to all cases that came our way in the connection with our project related to Universal Plug and Play [22].

The ASM specification P describes a device whose state is altered by one outside agent (the user or the tester). At each step, independently of the current state of the device, the agent invokes one of the input actions. We would like to find out whether the device behaves according to the specification. The problem that we address here is how to find a good conformance test suite. The method we are describing consists of two, largely independent, steps that are treated in the subsequent sections.

1. Extract a finite state machine M from P.
2. Generate a test suite from M.

Remark. In reality the device may have its own rules and may be influenced by more than one agent. For testing purposes, the more general scenario reduces to the simplified one.

Door example. The following sample specification is a stripped-down version of a Universal Plug and Play door controller specification. For the purposes of the presentation, we omitted several actions and state variables (that are used for the locking and latching of the door as well as for the explicit setting and resetting of the counter). This specification will be used as a running example throughout the paper.

Consider a door that can be closed and opened. The controller of the door has a counter for the number of times it has been opened and then closed. The counter has a maximum value that is a positive integer; when this value is reached, the counter starts over from 0. The ASM specification has the dynamic nullary functions `open` and `counter`.

```
class Door
  var open as Boolean
  var counter as Integer
  Open()  = if not open then open:=true
```

```
Close() = if open and counter<Max then
            open    := false
            counter := counter+1
          if open and counter=Max then
            open    := false
            counter := 0
```

The program can be simplified by counting modulo `Max` but current form has its advantage; it stresses the importance of the border case.

2.1 Coverage

One can define various notions of coverage in terms of the generated FSM M, the ASM program P, and the generated test suite T, reflecting how good the test suite is.

We look at the ASM program structure. *ASM guard coverage* is achieved under the following condition. For every action program and each of its clauses, there is some state where the action is invoked with the guard of the clause being true. This notion of coverage is related to C2 coverage, also called branch coverage, in path testing of software [2].

One achieves *node coverage* if all nodes in M are visited by T. One achieves *link coverage* if all links in M are visited by T. In general, guard coverage does not imply node or link coverage. Conversely, link coverage does not always imply guard coverage. Both directions are illustrated below using the door ASM. Link coverage implies node coverage though, because there are no isolated nodes in the generated FSM.

3 The Extraction Algorithm

In order to find a test suite we extract a finite state machine from the ASM specification and then use test generation techniques for FSMs. In this section, we concentrate on the extraction problem.

The extraction uses the syntactic form of the ASM specification in an essential way. Notice that the state of the specification does not include information on which, if any, is the current control action.

Recall that two states are *equivalent* if they are guard indistinguished. A *hyperstate* is a class of this equivalence relation. If the program has m guards then the hyperstates can be succinctly represented as binary strings of length m. This leads to a finite state machine[1].

[1] Usually only some binary strings of length m represent reachable states. Many combinations of guards represent unreachable states or violate the integrity constraints and thus represent no states at all.

The extraction algorithm works with a given specification ASM and is itself represented as an ASM. In the following we use the door controller as our sample specification.

A *test state* is an encoding of the dynamic part of the state of the specification ASM. Test states are represented here by structured data (other representations are certainly possible).

```
structure TestState
  open as Boolean
  counter as Integer
```

The rule `fire` is used by the FSM generator to invoke any of the actions in the specification by its name.

```
fire(S as TestState, A as String) as TestState =
  machine
    let D = new Door(S.open, S.counter)
    if A = "Close" then D.Close() else D.Open()
  step
    return TestState(D.open,D.counter)
```

The function `hyperstate` returns the representation of the hyperstate of the given test state (for example as a string of 0s and 1s). In this version of the algorithm, `representative(H,S)` is a binary dynamic relation expressing that `H` is a hyperstate and `S` is a test state that has been deposited as a representative of `H`. The nullary function `S0` provides the initial test state of the specification. The generated state transitions are recorded in the ternary dynamic relation `links`. The dynamic function `frontier` includes all the test states still to be processed.

Initially, `representative` is {`(hyperstate(S0),S0)`}, `links` is the empty set, and `frontier` is {`S0`}. The core of the extraction algorithm is to repeatedly execute the following rule until the frontier is empty. See Appendix A for more details.

```
genFSM() =
  choose S ∈ frontier do
    frontier(S) := false
    forall A  ∈ Actions do
      let T = fire(S,A)
      representative(hyperstate(T),T) := true
      links(S,A,T) := true
      if relevant(T) then frontier(T):=true
```

The definition of what is a *relevant* test state plays a central role in the algorithm. The following definition states that the only relevant test states are those whose hyperstates have not yet been encountered.

```
relevant(T as TestState) as Boolean =
   ¬(∃ x ∈ representative where first(x)=hyperstate(T))
```

At another extreme, the following definition states that a test state is relevant if it has not been encountered yet. With this definition the algorithm may not terminate, unless the total number of states is finite.

```
relevant(T as TestState) as Boolean =
   ¬(∃ x ∈ representative where second(x)=T)
```

Unless stated otherwise, we will assume that `relevant` is defined according to the first definition.

Door example continued. There are 3 different guards in the door ASM and therefore at most 2^3 possible hyperstates, with each guard being either true or false. Some of the combinations are ruled out by being inconsistent. The guards are:

g_1 : `open and counter=Max`
g_2 : `open and counter<Max`
g_3 : `not open`

Every hyperstate can be characterized by a binary sequence $b_1b_2b_3$ of 0's and 1's, as the set of all states that satisfy g_i if and only if $b_i = 1$. We will denote such a hyperstate by $H_{b_1b_2b_3}$.

The initial state `S0` is such that the counter is 0 and the door is closed. Thus, g_1 and g_2 are false and g_3 is true, i.e.,

$$\texttt{hyperstate(S0)} = H_{001}$$

When we invoke the action `Close` in the initial state, the state does not change. When we perform the action `Open` in the initial state we get a new state `S1` where the value of `counter` is 0 and the value of `open` is true.

```
fire("Close",S0) = S0
fire("Open",S0)  = S1
```

The hyperstate of `S1` is H_{010}, i.e., the door is open and the counter is less than the maximum value.

When we perform the actions in the state `S1`, we get back states that are representatives of either H_{010} or H_{001} and the algorithm terminates:

`fire("Close",S1)` = `S0'`, (where `hyperstate(S0')` = `hyperstate(S0)`)
`fire("Open",S1)` = `S1`

The generated finite automaton is illustrated in Figure 1. Its nodes are the generated hyperstates and there is a transition with label L from node H to node H' if there is a link with label L from some representative of H to some representative of H'.

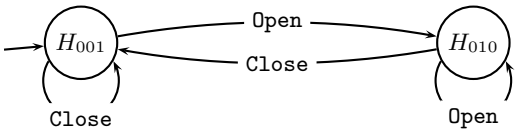

Fig. 1. Finite automaton generated from the door example.

Remark. Notice that the input ASM program P is *executed* in the process of generating the FSM. The potential state explosion problem of the generated FSM is ameliorated by producing only those hyperstates that are reachable from the given initial state. Still, the total number of reachable hyperstates may be exponential in the number of guards in P.

3.1 Complexity

Even though the process described in the previous section works in practice (at least in our practice), in general the problem of extracting the finite state machine is hard. As the following complexity results show, it may not even be possible or computationally feasible to have an algorithm that generates all possible hyperstates that are reachable from a given initial hyperstate.

Consider for example, the case when there is only one action a with the program

$P_a =$
`if` $p(x_1, x_2, x_3, x_4) \neq 0$ `then` *Updates*
`if` $p(x_1, x_2, x_3, x_4) = 0$ `then` *Halt*

where $x_1, \ldots, x_4$ are integer state variables and p is a polynomial. *Updates* can be chosen so that varying the polynomial makes the problem of the existence of a halting (hyper)state (where $p(x_1, x_2, x_3, x_4) = 0$) undecidable. This uses [20]. Now replace $p(x_1, x_2, x_3, x_4) \neq 0$ with $\phi(b_1, \ldots, b_n)$ where $b_1, \ldots, b_n$ are Boolean state variables and ϕ is a propositional formula. The problem of the existence of a halting (hyper)state becomes NP complete.

4 Generating a test suite

The extraction algorithm produces a finite state machine. View the machine as a directed graph and mark each edge with the cost of executing the corresponding

action at the corresponding hyperstate. You want to walk through the graph in a cheapest possible way traversing every edge at least once. That is the well known *Chinese Postman Problem* [14] that naturally arises in conformance testing [17, 19]. The problem has an efficient solution in the case when the finite state machine is deterministic and strongly connected.

In general, the deterministic case has been studied extensively in the literature and there exist several other methods for exploiting the structure of a deterministic FSM, see e.g. [23]. The most common of them is the *transition-tour* method, also known as the T-method, and one version of the T-method uses the Postman Tour. We have integrated an efficient Postman Tour algorithm [24] with our extraction algorithm.

Nondeterministic case. In our applications, nondeterminism is limited but it does arise. Nondeterminism in the FSM is either caused by nondeterminism in the device and thus in the specification, or by the extraction algorithm as a result of abstraction. We are currently investigating different ways to deal with nondeterminism.

5 Future Work

There are several open issues with the methodology that we have described. We focus below on the so-called *non-discovery problem* that has to do with the fact that not all reachable hyperstates may be discovered. There are other issues that we haven't dealt with in this paper. For example state explosion when joining several (essentially independent) ASM programs into one. We are also looking at alternative definitions for forming hyperstates. For example, instead of guards themselves, one may consider sets of smallest closed subformulas of guards as the basis for forming hyperstates. Normally, this leads to more hyperstates but may provide better coverage.

In this paper, we assume that the action programs do not take parameters. A solution to this problem will be presented in a subsequent paper. It builds on grouping the values of parameter vectors according to the guards.

How to deal with nondeterminism is another open issue that has consequences regarding the applicability of known FSM based test case generation techniques. One interesting approach is the test case generation framework based of labeled transition systems [25]. Some techniques for decreasing and sometimes even eliminating nondeterminism have been studied in the context of *extended finite state machines* [9, 18] or EFSMs, EFSMs generalize the finite state machine model by introducing state variables that can be tested and changed during a transition. Notice that the structure generated by our extraction algorithm is richer than an FSM. It is in fact a simple ASM that generalizes an EFSM.

5.1 The Non-Discovery Problem

Consider the door example. The algorithm generates a finite automaton with two nodes, see Figure 1. It never discovers the reachable hyperstate where the value of `counter` equals `Max`. That is the manifestation of the Non-Discovery Problem in this example. If we would treat all new test states as relevant the problem would not arise, but this would often be infeasible.

In general, you start with a test state s and you explore a vicinity V of it. There could be states t in V that are treated as irrelevant but which may potentially lead to new, and thus undiscovered (or non-discovered), hyperstates. A more subtle manifestation of the Non-Discovery Problem is when you have all the hyperstates but you miss certain links.

To illustrate the second phenomenon, let us modify the door controller specification by adding a new action `SetMax`:

```
class Door..
  SetMax() = if true then counter:=Max
```

Clearly, we still have the same hyperstates. The algorithm will now discover the hyperstate H_{100}. It has a single representative corresponding to the door being open and the counter being max. Let us run through the algorithm once with this extended set of actions, staring from the initial state `S0`.

First iteration. The frontier consists of `S0`, i.e. $\neg$`open` $\wedge$ `counter` $=$ `0`.

```
fire("Close",S0)  = S0
fire("Open",S0)   = S1 = open ∧ counter=0
fire("SetMax",S0) = S2 = ¬open ∧ counter=Max
```

We can easily calculate that `hyperstate(S1)` $= H_{010}$, `hyperstate(S2)` $= H_{001}$ and thus only `S1` is added to the frontier.

Second iteration. The frontier consists of `S1`.

```
fire("Close",S1)  = S3 = ¬open ∧ counter=1
fire("Open",S1)   = S1
fire("SetMax",S1) = S4 = open ∧ counter=Max
```

We get that `hyperstate(S3)` is H_{010}, which is an existing hyperstate, and that `hyperstate(S4)` is H_{100} is a new hyperstate. Hence, only `S4` is added to the frontier.

Third iteration. The frontier consists of `S4`.

```
fire("Close",S4)  = S0
fire("Open",S4)   = S4
fire("SetMax",S4) = S4
```

No new states are added to the frontier and the algorithm terminates. The generated automaton is shown in Figure 2.

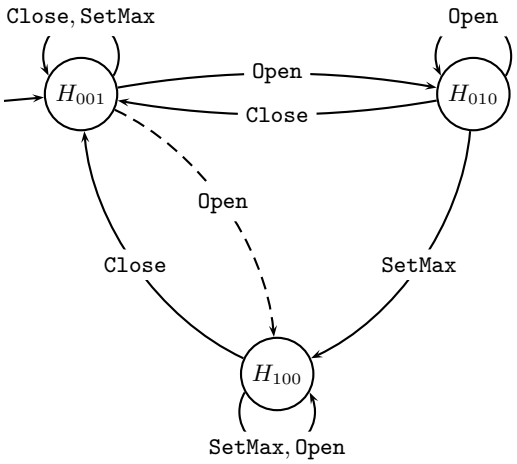

Fig. 2. Finite automaton generated from the door example. The dashed transition is not part of the generated FSM.

As shown, the algorithm does not discover the transition labeled by the action `"Open"` from H_{001} to H_{100}. Notice that this link is obtained by opening the door in a state where the counter has already the maximum value.

In general the Non-Discovery problem is unavoidable: there may be no algorithm that always halts and generates the hyperstates for all reachable states. Nevertheless it must be coped with in one way or another. One can for example make the process interactive and rely on the tester to provide the undiscovered links explicitly.

Symbolic methods. We can try to find the undiscovered hyperstates and links by solving the corresponding equations which may be possible in many cases. This direction seems to be fruitful. We can identify important classes of cases where the problem is solvable.

Randomization. Another approach is to use randomization. If you know the set of states representing a given hyperstate h (rather than knowing only those representative that have been discovered), you can randomly choose one and apply the various actions. If the set is not too big then you can go through all

the representatives. (There are other ways to use randomization for testing. Here is a naive way: start from the initial state and apply a randomly chosen action, then again apply a randomly chosen action, and so on. There are various ways to finish the process. Of course, this does not solve the Non-Discovery Problem.)

A Semi-Automatic Approach Currently we have adopted a pragmatic approach to the non-discovery problem. In the specifications that we have studied so far, coming mostly from the context of UPnP [22], it is very often the case that if a certain hyperstate is not reached, then this is directly reflected in the guard coverage of the ASM. In some sense this is to be expected, because if a guard is never true then either the clause is irrelevant and can be removed completely, or the algorithm was not able to reach a state satisfying the guard. We have developed a prototype environment where the tester may intervene by introducing additional links and rerun the algorithm incrementally. This does not really solve the problem of course, because in general guard coverage does not imply node coverage.

6 Related work

The two main approaches for test case generation are based on *labeled transition systems* (LTSs) and *finite state machines* (FSMs). A review of both approaches is given in [3]. In this section we look briefly at both and their relation to our work.

The main characterizing feature for all the test case generation techniques mentioned below is that they use the specification symbolically. This is in sharp contrast to the ASM approach introduced in this paper where the specification itself is executed to produce the test cases.

6.1 LTS based testing

Conformance testing plays a central role in testing communication protocols, where it is important to have a precise model of the *observable behaviour*. This has lead to a testing theory based on *labeled transition systems*. LTSs may in general be *nondeterministic*. See an overview in [25] and an overview of the literature in [6].

Compared to the ASM approach proposed in this paper and to some FSM and EFSM based techniques, the main drawback of LTS based test case generation is that the specification is normally taken "as is" and no systematic method is used to take advantage of the structure of the specification. It is therefore difficult to produce restricted test suits and make claims about resulting coverage. The best one can do really, is to produce as many test cases as the resources allow. State explosion is another important issue where known verification techniques may be

used [10]. Verification techniques can also be used to generate test cases when a *test purpose* (a property to be tested) is given. TGV [12] is an industrial tool that utilizes this approach to generate test cases from SDL and Lotos specifications.

6.2 FSM based testing

FSM based testing was initially driven by problems arising in functional testing of hardware circuits. The theory has recently also been adapted to the context of communication protocols. Most of the work in this area has dealt with *deterministic* FSMs. See [18, 23] for comprehensive surveys and [21] for an overview of the literature. The *Extended Finite State Machine* (EFSM) approach has been introduced mainly to cope with the *state explosion problem* of the FSM approach. Essentially the problem arises if the system to be modeled has variables that may take values in large, even infinite, domains, for example integers. In an EFSM such variables are allowed and the transitions may depend on and update their values. See [5, 9, 18].

In EFSMs the control part is finite and is separated from the data part, which distinguishes them from ASMs. The purpose of the FSM extraction algorithm in Section 3 is essentially to extract a finite control part out of the ASM. The problem of removing nondeterminism from such an extracted control FSM is closely related to the stabilization problem of EFSMs. The stabilization problem is addressed in [9]. The inability to directly deal with nondeterminism is the main drawback of the FSM based approaches.

In the context of software testing, advanced theorem proving techniques have been introduced for extracting finite automata from Z specifications for test case generation. This is demonstrated by the *disjunctive normal form* approach [11] that is used also in [16]. The overall approach is similar to what is proposed in this paper. Finite automaton based testing for object oriented software is introduced in [26]. In this context [7] introduces techniques for factoring large, possibly nondeterministic, FSMs into smaller deterministic ones. Some of these techniques have been implemented in the KVEST tool [8]. More work related to finite state machine based software testing can be found on the homepage of Model-Based Testing [1].

References

1. Model-Based Testing. `http://www.geocities.com/model_based_testing/`.
2. Boris Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, New York, second edition, 1990.
3. G.V. Bochmann and A. Petrenko. Protocol testing: review of methods and relevance for software testing. In *Proceedings of the 1994 international symposium on Software testing and analysis*, pages 109–124, 1994.
4. Egon Börger. Abstract state machines at the cusp of the millenium. In Y. Gurevich, P. W. Kutter, M. Odersky, and L. Thiele, editors, *Abstract State Machines: Theory*

and Applications, ASM'2000, volume 1912 of *Lecture Notes in Computer Science*, pages 1–8. Springer, 2000.
5. C. Bourhfir, R. Dssouli, and E.M. Aboulhamid. Automatic test generation for EFSM-based systems. Publication departementale 1043, Departement IRO, Universite de Montreal, August 1996.
6. E. Brinksma and J. Tretmans. Testing Transition Systems: An Annotated Bibliography. In F. Cassez, C. Jard, B. Rozoy, and M. Ryan, editors, *Summer School MOVEP'2k – Modelling and Verification of Parallel Processes*, pages 44–50, Nantes, July 2000.
7. I.B. Burdonov, A.S.Kossatchev, and V.V. Kulyamin. Application of finite automatons for program testing. *Programming and Computer Software*, 26(2):61–73, 2000.
8. I.B. Burdonov, A.S.Kossatchev, A. Petrenko, and D. Galter. Kvest: Automated generation of test suites from formal specifications. In J. Wing, J. Woodcock, and J. Davies, editors, *FM'99, Vol. I*, volume 1708 of *Lecture Notes in Computer Science*, pages 608–621. Springer, 1999.
9. K.-T. Cheng and A.S. Krishnakumar. Automatic generation of functional vectors using the extended finite state machine model. *ACM Transcations on Design Automation of Electronic Systems*, 1(1):57–79, January 1996.
10. R.G. de Vries and J. Tretmans. On-the-fly conformance testing using Spin. *Software Tools for Technology Transfer*, 2(4):382–393, March 2000.
11. J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. In *Proc. FSE 93*, 1993.
12. J.C. Fernandez, C. Jard, T. Jéron, and C. Viho. An experiment in automatic generation of test suites for protocols with verification technology. *Science of Computer Programming - Special Issue on COST247, Verification and Validation Methods for Formal Descriptions*, 29(1–2):123–146, 1997.
13. Foundations of Software Engineering, Microsoft Research. Abstract state machine language. `http://research.microsoft.com/fse`.
14. J. Gross and J. Yellen. *Graph Theory and its Applications*. CRC, Boca Raton, 1999.
15. Y. Gurevich. Evolving algebra 1993: Lipari guide. In Egon Boerger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
16. S. Helke, T. Neustupny, and T. Santen. Automating test case generation from Z specifications with Isabelle. In *Proc. ZUM 97*, 1997.
17. G.J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, Englewood Cliffs, NJ, 1991.
18. D. Lee and M. Yannakakis. Principles and methods of testing finite state machines – a survey. In *Proceedings of the IEEE*, volume 84, pages 1090–1123, Berlin, Aug 1996. IEEE Computer Society Press.
19. R.J. Linn and M.Ü. Uyar. *Conformance Testing Methodologies and Architectures for OSI Protocols*. IEEE Computer Society Press, Los Alamitos, CA, 1994.
20. Yuri V. Matiyasevich. *Hilbert's Tenth Problem*. MIT Press, 1993.
21. Alexandre Petrenko. Fault model-driven test derivation from finite state models: Annotated bibliography. In *Proceedings of the Summer School MOVEP2000, Modelling and Verification of Parallel Processes*, 2000. To appear in LNCS.
22. Universal Plug and Play Forum. `http://www.upnp.org`.
23. Deepinder P. Sidhu and Ting-Kau Leung. Formal methods for protocol testing: A detailed study. *IEEE Transactions on Software Engineering*, 15(4):413–426, April 1989.

24. H. Thimbleby. An algorithm for the directed Chinese Postman Problem (with applications). Technical report, Middlesex University School of Computing Science, London, 2000.
25. J. Tretmans and A. Belinfante. Automatic testing with formal methods. In *EuroSTAR'99: 7th European Int. Conference on Software Testing, Analysis & Review*, Barcelona, Spain, November 8–12, 1999.
26. C.D. Turner and D.J. Robson. The state-based testing of object-oriented programs. In *Proc. IEEE Conf. Software Maintenance*, pages 302–310, 1993.

A FSM extraction algorithm

The following is an executable AsmL [13] specification of the FSM extraction algorithm introduced in Section 3.

A.1 Test Harness

A class to be tested must support the following interface.

```
interface Testharness[TestState]
  S0() as TestState
  actions()  as Set[String]
  hyperstate(T as TestState) as String
  fire(S as TestState, A as String) as TestState
```

A.2 FSM Generator

The class GenFSM defines the central reachability algorithm. It is completly generic.

```
class GenFSM[TestState] (t as Testharness[TestState])
  var links as Set of TestState*String*TestState = {}
  var frontier as Set of TestState               = {t.S0()}
  var representative as Set of String*TestState  =
        {(t.hyperstate(t.S0()),t.S0())}

  genFSM() =
    choose S in frontier do
      frontier(S) := false
      forall A  in t.actions() do
        let T = t.fire(S,A)
        representative(t.hyperstate(T),T) := true
        links(S,A,T) := true
        if relevant(T) then frontier(T):=true
```

```
  relevant(T as TestState) as Boolean =
    not(exists x in representative where first(x)=t.hyperstate(T))

class GenFullFSM[TestState] (t' as Testharness[TestState])
  extends GenFSM[TestState] (t')

  relevant(T as TestState) as Boolean =
    not(exists x in representative where second(x)=T)
```

A.3 The Main Program

The main program instantiates the FSM generation with a door controler and runs until no further change occurs.

```
run() =
  machine
    let th = new DoorTestharness()
    let fsm = new GenFSM(th)
  step
    while fsm.frontier ≠ {} do fsm.genFSM()
  step
    writeln("generated FSM = "+
            {(th.hyperstate(S), A, th.hyperstate(T)) |
                   (S,A,T) in fsm.links} )
```

A.4 Door Controler Specification

The following sample specification is a stripped-down version of a Universal Plug and Play door controler specification.

```
Max as Integer = 10

class Door
  var open as Boolean
  var counter as Integer

  Close() =
    if open and counter < Max then
      open:=false
      counter:=counter+1
    if open and counter=Max then
      open:=false
      counter:=0
```

```
  Open() = if not open then open:=true

  SetMax() = counter := Max
```

A.5 Door Controler Test Harness

This is the appropriate instance of a test harness for a door controler. It must be provided by the user.

```
structure DoorTestState
  open as Boolean
  counter as Integer

class DoorTestharness implements Testharness[DoorTestState]
  S0() as DoorTestState = DoorTestState(false,0)
  actions() as Set[String] = {"Close", "Open", "SetMax"}

  hyperstate(T as DoorTestState) as String =
    let b0 = if T.open and T.counter = Max then "1" else "0"
    let b1 = if T.open and T.counter lt Max then "1" else "0"
    let b2 = if not T.open then "1" else "0"
    return(b0 + b1 + b2)

  fire(S as DoorTestState, A as String) as DoorTestState =
    machine
      let P = new Door(S.open, S.counter)
      if A = "Close" then P.Close()
      elseif A = "Open" then P.Open()
      else P.SetMax()
    step
      return DoorTestState(P.open,P.counter)
```