Generating Finite State Machines from Abstract State Machines

Wolfgang Grieskamp
Microsoft Research
Redmond, WA
wrwg@microsoft.com

Yuri Gurevich
Microsoft Research
Redmond, WA
gurevich@microsoft.com

Wolfram Schulte
Microsoft Research
Redmond, WA
schulte@microsoft.com

Margus Veanes
Microsoft Research
Redmond, WA
margus@microsoft.com

ABSTRACT

We give an algorithm that derives a finite state machine (FSM) from a given abstract state machine (ASM) specification. This allows us to integrate ASM specs with the existing tools for test case generation from FSMs. ASM specs are executable but have typically too many, often infinitely many states. We group ASM states into finitely many hyperstates which are the nodes of the FSM. The links of the FSM are induced by the ASM state transitions.

Keywords

finite state machine, FSM, abstract state machine, ASM, test case generation, executable specification

1. INTRODUCTION

The group on Foundations of Software Engineering at Microsoft Research has developed an industrial-strength high-level executable specification language AsmL [13]. AsmL builds on the concept of abstract state machine [18] and provides a modern specification environment that is object-oriented and component-based. Here we are concerned with using AsmL specifications as a source for algorithmic generation of test suites. This is one approach to model-based testing, and such testing is receiving more and more attention in Microsoft's product groups recently. Typically, however, the models used are finite state machines (FSMs). There are pretty good tools for deriving test suites from FSMs. So it is reasonable for us to integrate with those tools. This is how we arrived to the problem addressed in this paper: algorithmic generation of an FSM from a given AsmL spec. Of course the desired FSM should reflect important functionalities of the spec, so that the resulting test suites are meaningful. The FSM also should be of manageable size.

We start with an abstract state machine which models some implementation under test (IUT) and which is written in AsmL; this model will be referred to as the ASM or the ASM spec. The ASM may have too many, often infinitely many, states. To this end, we group ASM states into finitely many hyperstates. This gives rise to a finite directed graph or finite state machine whose nodes are the generated hyperstates. Then state transitions of the ASM are used to generate links between hyperstates. Let us note that it is not necessary to first produce an FSM and then use the FSM for test generation. The graph-generating procedure itself can be used to produce a test suite as a byproduct.

The FSM-generating algorithm is a particular kind of graph reachability algorithm. It starts from the initial state and builds up a labeled state transition graph by invoking actions that are parts of the ASM. If a new state is encountered, it is added to the frontier of unexplored states but only if this new state is considered to be relevant e.g. if it gives a new hyperstate. A suitable relevance condition is an important part of the algorithm that determines the quality of the generated FSM and whether the algorithm terminates.

The generated FSM and the original ASM are related in a natural way. Suppose that an action *a* is applied to an ASM state *s*. What will be the next state of the ASM or – if *a* is nondeterministic – what are the possible next states? That depends on Boolean guards of the spec of *a*. The guards reflect the state distinction that the specification writer cared enough about to make explicit. Say that two states of the ASM are *distinguishable by guards* if there exists a guard that is satisfied in one of the two states but not in the other. *Indistinguishability by guards* is an example of a useful equivalence relation given by a sequence of Boolean conditions, the *distinguishing sequence*. There are other useful distinguishing sequences and corresponding equivalence relations. The algorithm takes a distinguishing sequence as an additional input. *Hyperstates* are the corresponding equivalence classes.

We illustrate the algorithm on a medium size spec of a Universal Plug and Play (UPnP) device. UPnP device architecture is a world-wide industry standard for peer-to-peer network connectivity of various intelligent appliances, wireless devices and PCs [26].

Our reduction of states to hyperstates is a form of data abstraction. In model checking, data abstraction is a well-known technique to cope with state explosion. In the context of model checking, the more abstract model must *simulate* the original model, so that certain properties of the original model are preserved [8]. In order to ensure this simulation, the standard

data-abstraction algorithms used in model checking may yield an over-approximation (with extra transitions) of the ideal data abstraction [8]. An over-abstraction also arises when one uses the method of abstract interpretation of programs [9]. In contrast, the output of our algorithm may under-approximate the *true FSM* so that some reachable hyperstates or some links between reachable hyperstates may be missing. The reason is that we work with the states of the original ASM: every FSM node is the hyperstate of a reachable ASM state, and every FSM link is given by a transition between reachable ASM states. In general, there is no algorithm (guaranteed to terminate) that generates the true FSM; this is proved in 3.3.

This article is written in the AsmL tradition: the same text serves the human reader and the computer. The computer will extract the program part which appears in the special AsmL style and execute our FSM generating algorithm on the UPnP device example.

We do not presume that the reader is familiar with ASMs or AsmL. The article is self-explanatory. In particular, the AsmL rules themselves are pretty much self-explanatory but we also provide additional explanation. The interested reader can always consult the AsmL website [13].

2. BASIC SETUP

We are given an implementation under test (IUT) and an ASM specification *S*. We assume that the actions that the IUT can be invoked with form a fixed and finite set. This set is usually only an approximation of the set of all possible actions that the real IUT can be called with, that may in general be infinite if the real actions have parameters ranging over infinite domains. Parameter selection in testing is a difficult problem all by itself and it is out of the scope of the current paper.

The specification *S* reduces to the following normal form. For each action *a,* there is a rule called the *action spec* for *a*. Each action spec is a *do-in-parallel* block

```
if g1 then R1
...
if gk then Rk
```

where each *clause* `if gi then Ri` is composed from a Boolean-valued *guard* `gi` and *body* `Ri`. The body is a possibly nondeterministic rule without any conditional sub-rules. We say that there is a *transition with label a* or an *a-transition* from a state A_1 to a state A_2, if after firing (the action spec for) *a* in state A_1 a possible resulting state is A_2. A *run* is a finite sequence of transitions where the end state of every transition is the start state of its immediate successor (if any) in the sequence. We require that there is a *fixed initial state* for *S*. The *reachable* states of *S* are those states that can be reached from the initial state of *S* by means of the runs. The fact that action specs have the normal form is convenient but not essential for this paper.

The ASM specification *S* describes the desired behavior of the IUT leaving out various implementation details. IUT is the subject of actions by the external environment or by the user of the IUT. For simplicity, we consider one pool of actions (including those of the environment and those of the user) and have in mind a single testing agent. The problem we are dealing with here is to provide a set of action sequences (a test suite) that the testing agent can use to drive the IUT through as many distinguishable states as possible. You may want not to distinguish between states where the difference is not relevant from the testing standpoint. It is desirable that such indistinguishability relation is an equivalence relation that has only finitely many equivalence classes.

Conceptually, our method consists of two, largely independent, steps:

1. Extract a finite state machine *M* from the given *S*, where each node[1] of *M* represents an equivalence class of the states of *S*.
2. Use *M* to generate a test suite, that is a set of action sequences.

In the following sections we explain in detail how the proposed FSM extraction algorithm works. Only a short section 4 is devoted to the test suite generation.

2.1 Indistinguishability and Hyperstates

We will use Boolean-valued conditions to define the desired equivalence relation between the states of the given spec *S*. Natural candidates for such conditions are those that explicitly appear in *S*, but one may also consider their derivatives by e.g. incrementing or decrementing numerical boundary conditions that occur in *S*. Let *b* be a fixed nonempty sequence $b_0,...,b_{n-1}$ of *n* such conditions. Say that two states are *b-distinguishable* if some b_i distinguishes between them. Any two states that are not distinguishable by *b* are *b-indistinguishable*. We will, as a rule, omit the *distinguishing sequence b* when it is clear from the context. It is easy to see that the indistinguishability relation is a finite equivalence relation. Define *hyperstate* as an equivalence class of this equivalence relation. A transition is *local* if both of its endpoints are in the same hyperstate.

Notice that there are at most 2^n hyperstates. The actual number is less if the conditions in the sequence are not independent. For example, if the disjunction of all the conditions is necessarily true then the hyperstate where all the conditions are false is not realized. Let *H* be a hyperstate; the *index* of *H* is the binary sequence $h_0,...,h_{n-1}$ such that, for all $i<n$, $h_i=1$ if b_i holds in the states in H, and $h_i=0$ otherwise. The index of a hyperstate *H* uniquely determines which conditions of *b* are true and which are false in the states of *H*. We identify hyperstates with their indices. The *index of a state* is the index of its hyperstate.

2.1.1 Selecting the Distinguishing Sequence

There are various natural selections of the distinguishing sequence *b*. One selection comprises the guards that actually appear in *S*. This reflects the case distinctions made explicit by the author of *S*. A second selection comprises the Boolean constituents (that is Boolean-indecomposable parts) of the guards. Typically, the number of hyperstates is larger in the second case,

[1] In order to avoid confusion between the states of S and the states of the finite automaton, we use the term *node* for the latter.

which, depending on the purpose, may be an advantage or a disadvantage. The second selection will be illustrated below. There are intermediate selections. For example, you reduce every guard to a disjunctive (or the full disjunctive) normal form and then use the conjunctions. (In the full disjunctive normal form, every one of those conjunctions contains every constituent, negated or non-negated.)

2.1.2 The True FSM

Ideally, we would like to generate an FSM from S and b that contains all the possible links. We call this the *true FSM for S and b*. The *nodes* of the true FSM are the indices of all the reachable states of S. The *initial node* is the index of the initial state of S. There is a *link* (n_1,a,n_2) from node n_1 to node n_2 with label a in the true FSM if there is an a-transition from a reachable state of S with index n_1 to a state with index n_2. Notice that you may have also unreachable states with index n_1; the restriction to reachable states in the previous sentence is important.

Some remarks: 1) Not all hyperstates are necessarily represented in the true FSM, only the reachable ones are. 2) The true FSM may be nondeterministic even if S is deterministic. This occurs when there exist two state transitions with the same label from two indistinguishable states to two distinguishable states. 3) The true FSM may be deterministic even if S is nondeterministic. Intuitively this means that the nondeterminism in S is not visible at the abstraction level imposed by the distinguishing sequence.

2.2 A Sample Device as IUT

As a running example we use one service of a medium size UPnP device as our IUT. UPnP device architecture is a standard for peer-to-peer network connectivity of various intelligent appliances, wireless devices and PCs; see the website [26] of the industrial UPnP Forum. A distributed ASM model of the UPnP is described in [15][16].

Here we consider a *CD player*. In the full model (see [16]) this device has two services, *ChangeDisc* and *PlayCD*. We use the the first one as our running example; see Figure 1. It allows a user (the *control point* in UPnP terms) to add discs to or remove discs from the CD player, to choose a disc to be placed on the tray, and to toggle (open/close) the door. Figure 1 illustrates the relevant state information associated with the service.

We use AsmL to write a specification for the ChangeDisc service. This will also provide a small introduction to AsmL.

First we describe the variables of the ASM state together with the initial values.

```
class CHANGEDISC
  var occupiedSlots as Set of Integer = {}
  var currentSlot   as Integer = 1
  var doorIsOpen    as Boolean = false
  var doorIsStuck   as Boolean = false
  var result        as RESULT  = undef
```

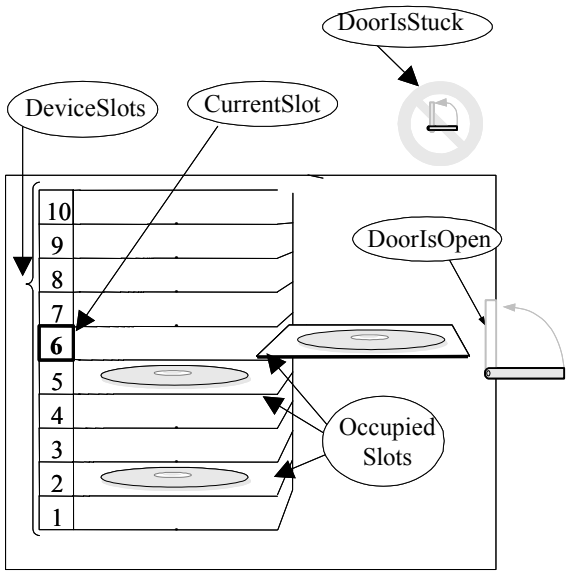

Figure 1. ChangeDisc service of a CD Player.

The reader may wonder what the role of the `result` is. Apply an action a to a state s and let t be a resulting state. The action may produce a Boolean or an error.

```
structure RESULT
  case ERR
    code as String
  case BOOL
    res as Boolean
```

This result is important and the hyperstate of t should reflect it. But here, for simplicity, we all but ignore the results. Our hyperstates do not reflect them. But we leave the result in the state t as a reminder of the importance of results. Alternatively we could attach the results to transitions rather than states. The question arises what is the result of the initial state because it is not a priori produced by any transition. Naturally the result of the initial state is undef.

Now we describe the part of the state that does not change: the set of slots. The additional constraint says that the current slot and all the occupied slots must be members of the set of all slots.

```
class CHANGEDISC ...
  allSlots as Set of Integer
  constraint
    currentSlot in allSlots and
    occupiedSlots subset allSlots
```

The action specs use several private helper functions whose intended meaning is self-explanatory. For example, the trayHasDisc() function checks whether the current slot is occupied. The remaining helper functions are defined in the appendix.

```
class CHANGEDISC ...
  trayHasDisc() as Boolean
    return currentSlot in occupiedSlots
```

The `AddDisc` action opens the door (or leaves it open) and chooses a free slot as the new current slot. There are altogether 11 actions, see the appendix.

```
class CHANGEDISC ...
  AddDisc()
    if not (isFull() or
            isClosedAndStuck()) then
      doorIsOpen := true
      choose slot in emptySlots()
        currentSlot := slot
    else
      ReportAnyError({(isFull(),"full"),
           (isClosedAndStuck(),"stuck")})
```

A hyperstate is defined in this algorithm as a sequence of Booleans.

```
structure Hyperstate
  content as Seq of Boolean
```

We define the distinguishing sequence to be a sequence of all the Boolean-indecomposable conditions that appear in the guards.

```
class CHANGEDISC ...
  GetHyperstate() as Hyperstate
    h1 = doorIsOpen
    h2 = trayHasDisc()
    h3 = successors() = {}
    h4 = predecessors()={}
    h5 = isEmpty()
    h6 = isFull()
    h7 = doorIsStuck
    return
      Hyperstate([h1,h2,h3,h4,h5,h6,h7])
```

2.3 Coverage

One can define various notions of coverage in terms of the generated FSM *M*, the ASM specification *S*, and the generated test suite. Although we will not go into details of any of the notions, since it is a separate topic all by itself, it is worth mentioning that there are in principle two different ways to measure coverage here. One is to look at the structure of *M* (as a directed graph) and the other is to look at the structure of *S*. The IUT itself is assumed to be a black box. In the first case, one may consider *node coverage* and *link coverage*. In the second case, one may consider *statement coverage* or *branch coverage* at the clause level of the action specs [1]. Several notions of structural ASM specification coverage are defined in [14].

3. FSM EXTRACTION FROM ASMS

In this section, we concentrate on the finite state machine extraction problem. The extraction algorithm works with a given ASM spec and is itself described as an ASM. The algorithm keeps executing the actions of the given spec on concrete states of that spec and building up the transition graph as it goes. The end state of a new transition is added to the *frontier* if the transition is *relevant* in an appropriate technical sense; the start state of the transition is deleted from the frontier. Initially the frontier consists of the initial state. The algorithm terminates when the frontier becomes empty.

Typically the state space is very large and you want to prune it as much as possible while reaching as many hyperstates as possible. This brings us to the problem of finding an adequate definition of relevance. First we show two relevance definitions that are the two extreme cases of a wide spectrum of possible relevance definitions, and we explain why these two definitions are problematic. Toward a solution of the relevance problem we introduce the notion of an *improvement* relation between states. Roughly, the improvement relation provides domain-specific knowledge for the algorithm to make better choices in pruning the search space. Using the improvement relation, we define a notion of relevance that is actually used in the algorithm.

In general our algorithm may "under-approximate" the true FSM, that is some links or even nodes may be missing. In general, it takes an unreasonably liberal relevance condition to guarantee that the true FSM is constructed in full. We will return to this problem in Section 3.3.

We now describe the algorithm in detail using AsmL. A *test-state* is the dynamic part of the full state of the spec. For brevity, we will often omit the "test" qualifier. States (that is test states) are naturally represented by AsmL structures.

```
structure State
```

Actions are identified by strings.

```
structure Action
  name as String
```

The FSM generation algorithm operates the spec by means of a test harness

```
class GenFSM
  h as Harness
```

The harness provides the initial state, the set of actions, a function that calculates (the index of) the hyperstate of a given state, and a method `Fire` that invokes a given action at a given state and returns a resulting state. There may be several possible resulting test states due to the nondeterminism of the spec.

```
interface Harness
  Initially() as State
  Actions() as Set of Action
  GetHyperstate(s as State) as Hyperstate
  Fire(s as State, a as Action) as State
```

The dynamic state of the GenFSM algorithm comprises the set of transitions that have been generated, initially empty; the frontier, a sequence of states to be traversed, initially containing only the

initial state; and the set of hyperstates that have been generated, initially containing only the hyperstate of the initial state.

```
class GenFSM ...
  var transitions
    as Set of(State,Action,State) = {}
  var frontier as Seq of State
  var hypers as Set of Hyperstate
```

The initialization of the frontier and hypers fields is given below.

The method `main` is the entry point of the algorithm.

```
class GenFSM ...
  main()
    step while frontier ne []
        generate()
```

The algorithm generates new transitions from the frontier states, handling one state and one action at a time. The AsmL statement `explore e` produces a sequence of all possible return values of the expression `e`. It makes it possible to handle nondeterministic ASM specifications. In this case, `explore` is used to produce all the transitions that exist from a given state on a given action.

```
class GenFSM ...
  generate()
    step
      s = head(frontier)
      frontier := tail(frontier)
    step foreach a in h.Actions()
      nextStates = explore h.Fire(s,a)
      step foreach t in nextStates
          transitions(s,a,t) := true
          if relevant(s,a,t) then
            frontier := frontier + [t]
            hypers := hypers union
                      {h.GetHyperstate(t)}
```

3.1 Relevance

The definition of *relevant* transition plays an important role in the algorithm. One possible definition of relevance (not the one that we will use) stipulates that a transition is relevant only if the hyperstate of the end state has not been encountered yet. This is clearly a minimal requirement.

```
class GenFSM ...
  relevant1(s as State,
            a as Action,
            t as State) as Boolean
    return h.GetHyperstate(t) notin hypers
```

The potential state explosion problem of the generated FSM is somewhat ameliorated by the fact that only reachable hyperstates are produced. But the total number of reachable hyperstates may be also exponential in the length of the distinguishing sequence.

The second definition of relevance (again, not the one that we will use) stipulates that a transition is relevant if the end state itself (rather than hyperstate) has not been encountered yet. With the second definition, the algorithm will not terminate unless the total number of reachable states is finite. If it does terminate the resulting FSM is the true FSM, no reachable state has been omitted.

```
class GenFSM ...
  relevant2(s as State,
            a as Action,
            t as State) as Boolean
    //check if t has been encountered
```

3.1.1 CHANGEDISC Example

Recall the CHANGEDISC service specification in 2.2. A harness for it is given in the appendix.

If we run a 30 slot version of the spec with $relevant_1$ then we obtain 24 hyperstates. Many hyperstates are missing because the algorithm never discovers the ones where the CD player is full. If, on the other hand, we run the same 30 slot version with $relevant_2$ then the state space explodes. See also Table 1 in this connection.

The problem with $relevant_1$ is quite general. In order to discover a new hyperstate, you may need new representatives of the hyperstates that have been encountered, but the relevance conditions forces you to discard all new representatives. This is related to the non-discovery problem discussed in Section 3.3. A partial but practically important solution of this problem is provided by the notion of *improvement* relation.

3.2 Improvement Relations

Consider a fixed specification. We want to use our domain-specific knowledge about its state space to define the relevance condition. How can we do that? Typically there may be certain (more or less abstract) *goals* that we want to achieve. For example, in the case of the CD changer the goal may be that all the slots are occupied. Given a goal, we want the algorithm to make progress towards that goal and thus discover new hyperstates. An appropriate *improvement* relation on states allows us to achieve that. When you encounter a new state, check whether it is an improvement toward the goal comparative to the old state. Improvement relations help us define appropriate relevance conditions.

The GenFSM algorithm itself is independent of the particular improvement relations. They are given by the harness.

```
interface Harness ...
  //returns true if t is
  //an improvement over s toward the goal g
  improved(s as State,
           t as State, g as Goal) as Boolean
  goals() as Set of Goal  //set of all goals
```

Now we are ready to give a useful relevance condition which we will use. A transition `(s,a,t)` is relevant if either `t` leads to a new hyperstate or else `t` is an improvement over the best state seen so far toward some goal. In the `bestState` map we keep track of the best state seen so far for each goal.

```
class GenFSM ...
  var bestState as Map of Goal to State
  GenFSM(h1 as Harness)  //the constructor
    h = h1
    bestState = {g |-> h1.Initially() |
                            g in h1.goals()}
    frontier = [h1.Initially()]
    hypers =
      {h1.GetHyperstate(h1.Initially())}

  relevant(s as State,
           a as Action,
           t as State) as Boolean
    forall g in h.goals() where
           h.improved(bestState(g),t,g)
      bestState(g) := t
    return
      (h.GetHyperstate(t) notin hypers) or
      (exists g in h.goals() where
              h.improved(bestState(g),t,g))
```

3.2.1 CHANGEDISC Example Revisited

We have a single goal to reach any state where the disc changer is full.

```
enum Goal
  DCisFull
```

In order to define an appropriate improvement relation for that goal we use a weight function that gives the minimal distance to (that is the minimal number of actions required to reach) a state where the disc changer has no empty slots.

```
class CDHARNESS implements Harness
  goals() as Set of Goal
    return {DCisFull}
  improved(s as State,
           t as State, g as Goal) as Boolean
    return weight(t as CDState) <
           weight(s as CDState)

  weight(s as CDState) as Integer
    free= size(slots)- size(s.occupiedSlots)
    if s.doorIsOpen and
       (s.currentSlot notin s.occupiedSlots)
      then return 2 * free - 1
      else return 2 * free
```

When we run the 30 slot version of the algorithm with the relevance (called relevance_3 in Table 1 below) given by this improvement relation, it produces 44 hyperstates and 531 links between hyperstates. Some more statistics for the disc changer example is found in Table 1. With the given improvement relation, we discover all hyperstates of the true FSM but only about 85% of the links.

Table 1. Sizes of generated FSMs for the disc changer example with different # of slots and different relevance conditions.

	relevance_1		relevance_3		relevance_2 (true FSM)	
	nodes	**links**	**nodes**	**links**	**nodes**	links
1 slot	**8**	**88**	**8**	**88**	**8**	88
2 slots	**24**	**270**	**24**	**270**	**24**	273
3 slots	**24**	**273**	**40**	**475**	**40**	516
4 slots	**24**	**273**	**44**	**531**	**44**	619
>4 slots	24	273	44	531	44	625

3.2.2 An Example with Multiple Goals

Consider a spec with two actions *inc* and *dec*. There are two integer-valued state variables *x* and *y* with initial value 0. The *inc* action increments *x* by one and the *dec* action decrements *y* by one. The distinguishing sequence contains the conditions *x=max* and *y=min* where *min* is some negative number and *max* some positive number. The two obvious orthogonal goals are to reach a state satisfying the respective boundary condition. The definition of the improvement relation is obvious for both goals.

3.3 Non-Discovery Problem: Undecidability and Complexity

Even though the process described in the previous section works pretty well in practice, at least in our practice, the problem of extracting the true finite state machine is hard in general. To make this claim more precise, we introduce several decision problems and prove that they all are hard. It is hard to discover hyperstates and it is hard to discover links. We refer to all these decision problems together as the *non-discovery problem*.

Hyperstate reachability problem.
Instance: A spec *S*, a distinguishing sequence *b* and an index *h*.
Question: Is hyperstate *h* reachable?

First link discovery problem.
Instance: A spec *S*, a distinguishing sequence *b*, an action *a*, and an index *h*.
Question: Is there an *a*-transition from any state *s* in the initial hyperstate to any state *t* in the hyperstate *h*?

Here the initial hyperstate is the hyperstate of the initial state.

Second link discovery problem.
Instance: A spec *S*, a distinguishing sequence *b*, an action *a*, and an index *h*.
Question: Is there an *a*-link from the initial hyperstate to hyperstate *h*? In other words, is there an *a*-transition from a *reachable* state *s* in the initial hyperstate to any state *t* in *h*.

The third and fourth link discovery problems are like the first and second, except that h is assumed to be reachable.

Theorem 1 *The five decision problems are all undecidable.*

Proof. First we assume that S has a unique action *a*:

```
if ¬(p(x1,..,xk) = 0) then R
if p(x1,..,xk) = 0 then Halt
```

Here $x_1,...,x_k$ are integer variables and p is a polynomial with integer coefficients. In the initial state the integer vector **x** = $(x_1,..,x_k)$ has the value **0** = (0,…,0). The rule R transforms an integer vector **x** = $(x_1,..,x_k)$ to an integer vector **x'** in such a way that the infinite sequence

0, **0'**, **0"**, **0"'**, …

contains every k-dimensional integer vector. The distinguishing sequence *b* consists of one Boolean condition, namely $p(x_1,..,x_k)=0$, so that we have only two possible hyperstates. Finally let *h* be (the index of) the hyperstate where $p(x_1,..,x_k)=0$.

It is known that there is no algorithm that, given a polynomial $p(x_1,..,x_k)$ with integer coefficients and variables, decides whether p has a root [24]. This is the famous Diophantine Equation Problem. By the construction above, it reduces to the first and second link discovery problems and to the hyperstate reachability problem, so these three problems are undecidable.

To prove the undecidability of the third and fourth link discover problems, we extend the spec above by means of another action *a*' with spec

```
y := true
```

The new distinguishing sequence consists of one Boolean condition

```
(p(x1,..,xk) = 0) or y
```

Finally, *h* is the hyperstate where this condition holds. This hyperstate is reachable from the initial state by action *a*'. This gives us a reduction of the Diophantine Equation Problem to the second link discovery problem. QED

The proof of Theorem 1 shows that the three decision problems remain undecidable even if the distinguishing sequence contains only one Boolean condition.

Theorem 1 implies that there is no algorithm for constructing the true FSM unless we allow algorithms that may not terminate. In fact, the second relevance condition (the most liberal one) does allows us to extract the true ASM, but the resulting algorithm may not teminate.

To prove a simple lower complexity bound, define the bounded version of any of the five decision problems by requiring that the variables of the given spec *S* are all Boolean and use the length of the distinguishing sequence as the size of the problem.

Theorem 2 *The bounded versions of the five decision problems are all NP-hard.*

Proof. The proof is similar to the proof of Theorem 1 except that we use SAT, the satisfiability problem for propositional formulas rather than Diophantine Equation Problem. We indicate the necessary changes. In the spec of *a*, replace $p(x_1,..,x_k)=0$ with $\varphi(u_1,...,u_n)$ where $u_1,...,u_n$ are propositional variables and φ is a propositional formula. In the initial state, $u_1 = ... = u_n = 0$ (where 0 represents false). R transforms a Boolean vector **u** into a Boolean vector **u'** in such a way that every *n*-dimensional Boolean vector occurs in the sequence **0**, **0'**, **0"**, **0"'**, …. QED

Theorem 2 is sort of trivial because we write programs in AsmL which has bounded quantification, choice, etc. But Theorem 2 remains true if we restrict attention to any language that expresses the program in the proof for each *n*. For example, we can use the language of sequential ASMs [19] where the only dynamic symbols are propositional variables (nullary relational symbols in terms of [19]). We presume that any values of the propositional variables give rise to a legal (not necessarily reachable) state. In that case the actions are deterministic and the first link discovery problem is also in NP and thus 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 [17] that naturally arises in conformance testing [21][23]. 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. [28]. 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 [29] with our extraction algorithm.

5. REALATED WORK

As far as we know, the first automated technique for extracting FSMs from model-based specifications for the purpose of test case generation was introduced in [11]. The approach of [11] is based on a finite partitioning of the state space of the model using full disjunctive normal forms (full DNFs) of the conditions in the spec and is called the *DNF approach*. The modeling language used of [11] is VDM but the approach is more general. The DNF approach is extended in [20] and applied to Z specs where the explosion of the size of the partitioning is ameliorated by employing DNFs that are not necessarily full. While our partition of the state space is similar to that of the DNF approach, the two approaches are quite different. Most importantly, the DNF approach employs symbolic techniques while we execute the spec. Heuristics are used differently in the two approaches: in the DNF approach, heuristics are used as part of theorem proving, whereas we use heuristics to prune the search space. As far as the problem of scaling is concerned, the DNF approach suffers from

the explosion of terms. Besides, theorem proving is time-consuming. In our approach, there is a trade-off between the computation time and how closely you approximate the true FSM. We can play with relevance conditions. A more restrictive condition gives quicker termination but leads to a more severe under-approximation. A more liberal condition leads to a better approximation but you may have to terminate the algorithm by force.

Finite automaton based testing for object oriented software is introduced in [31]. Article [5] introduces techniques for "factoring" large, possibly nondeterministic, FSMs into smaller deterministic ones. Some of these techniques have been implemented in the KVEST tool [6].

In model checking, data abstraction is used to cope with state explosion. Typically your original model M is an FSM but it may be too large. Data abstraction groups states of M and produces a reduced model M_r which is analogous to our true FSM. Due to efficiency considerations, the standard data abstraction algorithms may yield an over-approximation of M_r; see [8]. Abstract interpretation based program testing is somewhat similar to but distinct from model checking; it also may lead to "the necessary over-approximation" [9]. In contrast, our approach may yield an under-approximation of the true FSM.

Recall that the purpose of our FSM extraction algorithm is to produce an FSM that can be used for test case generation. In general, the two main approaches for test case generation are those based on labeled transition systems (LTSs) and those based on finite state machines. A review of both approaches is given in [2]. In the following we look briefly at both.

Conformance testing plays a central role in testing communication protocols where it is important to have a precise model of the observable behavior of the system. This has lead to a testing theory based on labeled transition systems. See an overview of the approach in [30] and an overview of related literature in [4]. Labeled transition systems are in general nondeterministic; the ability to deal with nondeterminism is a virtue of the LTS approach. Another virtue is compositionality. On the other hand, the FSM approach is able to exploit the FSM graph structure to produce test suites for the desired coverage. In the LTS approach, verification techniques can be used to deal with state explosion and to generate test cases. TGV [12] is an industrial tool that utilizes the LTS approach to generate test cases from SDL and Lotos specifications.

FSM based testing was initially driven by problems arising in functional testing of hardware circuits. The theory has recently been adapted to the context of communication protocols. The bulk of the work in this area has dealt with deterministic FSMs. See [22][28] for comprehensive surveys and [25] 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. Typically the problem arises when the system to be modeled has variables with 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 [3][7][22]. In EFSMs, the control part is finite and is separated from the data part, which distinguishes them from ASMs. An interesting problem in our FSM generation algorithm is to fiddle with the hyperstates in order to avoid nondeterminism in the resulting FSM. This problem is related to the stabilization problem of EFSMs that is addressed in [7]. The inability to directly deal with nondeterminism is the main drawback of the FSM based approaches.

More work related to finite state machine based software testing can be found on the homepage of Model-Based Testing [27].

6. FUTURE WORK

Here are some but definitely not all problems to be addressed.

- We discussed above the problem of non-discovery of hyperstates and links. We are currently investigating other methods, in addition to the method of improvement relations, to get better approximations of the true FSM.
- An important issue that we haven't dealt with in this paper is state explosion that arises from joining several independent ASM specs into one.
- In this paper, we have assumed that the action programs do not take parameters. One possible approach to solving this problem builds on grouping the values of parameters according to the guards (or to the constituents of guards).
- How to best deal with nondeterminism in the generated FSM is another open issue that has consequences regarding the applicability of known FSM based test case generation techniques. Sometimes nondeterminism can be avoided by fiddling with the definition of hyperstates. The nondeterminism problem does in general not arise in the LTS approach. A generalization of the LTS approach to ASMs seems promising.

7. REFERENCES

[1] Boris Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, New York, second edition, 1990.

[2] 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.

[3] C. Bourhr, R. Dssouli, and E.M. Aboulhamid. Automatic test generation for EFSM-based systems. Publication departementale 1043, Departement IRO, Uni- versite de Montreal, August 1996.

[4] 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 Verication of Parallel Processes*, pp. 44-50, Nantes, July 2000.

[5] 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.

[6] I.B. Burdonov, A.S.Kossatchev, A. Petrenko, and D. Galter. Kvest: Automated generation of test suites from formal specications. 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.

[7] 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.

[8] E. M. Clarke, Jr., O. Grumberg and D. A. Peled, *Model Checking*, MIT Press, 1999.

[9] P. Cousot and R. Cousot. Abstract Interpretation Based Program Testing, *Proceedings of the SSGRR 2000 Computer & eBusiness International Conference*, Compact disk paper 248, L'Aquila, Italy, July 31 - August 6, 2000.

[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. FME'93, LNCS 670*, p. 268-284, Springer, 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, Website: http://research.microsoft.com/fse/AsmL

[14] A. Gargantini and E. Riccobene, ASM-based Testing: Coverage Criteria and Automatic Test Sequence Generation, *J. of Universal Computer Science*, 7(11):1050-1067, 2001.

[15] U. Glässer, Y.Gurevich and M. Veanes, High-Level Executable Specification of the Universal Plug and Play Architecture, In *Proc. of the Thirty-Fifth Annual Hawaii International Conference on System Sciences*, IEEE, 2002.

[16] U. Glässer, Y.Gurevich and M. Veanes, Universal Plug and Play Machine Models, Microsoft Research, Technical Report MSR-TR-2001-59, June, 2001.

[17] J. Gross and J. Yellen. *Graph Theory and its Applications*. CRC, Boca Raton, 1999.

[18] Y. Gurevich. Evolving algebra 1993: Lipari guide. In Egon Boerger, editor, *Specification and Validation Methods*, pages 9-36. Oxford University Press, 1995.

[19] Yuri Gurevich. Sequential Abstract State Machines capture Sequential Algorithms, *ACM Transactions on Computational Logic,* Volume 1, Number 1 (July 2000), pages 77-111

[20] S. Helke, T. Neustupny, and T. Santen. Automating test case generation from Z specifications with Isabelle. In *Proc. ZUM97*, *LNCS 1212*, p. 52-71, Springer, 1997.

[21] G.J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[22] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - a survey. In *Proceedings of the IEEE*, volume 84, number 8, pages 1090-1123, Berlin, Aug 1996. IEEE Computer Society Press.

[23] R.J. Linn and M.Ü. Uyar. *Conformance Testing Methodologies and Architectures for OSI Protocols*. IEEE Computer Society Press, Los Alamitos, CA, 1994.

[24] Y. V. Matiyasevich. *Hilbert's Tenth Problem*. MIT Press, 1993.

[25] A. 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. Appears in LNCS.

[26] Universal Plug and Play Forum. Website: http://www.upnp.org

[27] H. Robinson. Model-Based Testing. Website: http://www.geocities.com/model_based_testing/

[28] 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.

[29] H. Thimbleby. An algorithm for the directed Chinese Postman Problem (with applications). Technical report, Middlesex University School of Computing Science, London, 2000.

[30] 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 pp. 8-12, 1999. EuroStar Conferences, Galway, Ireland.

[31] C.D. Turner and D.J. Robson. The state-based testing of object-oriented programs. In *Proc. IEEE Conf. Software Maintenance*, pp. 302-310, 1993.

APPENDIX

Helper functions of the CHANGEDISC model.

```
class CHANGEDISC ...
  emptySlots() as Set of Integer
    return allSlots difference occupiedSlots
  successors()  as Set of Integer
    return {e | e in occupiedSlots
                where e gt currentSlot}
  predecessors() as Set of Integer
    return {e | e in occupiedSlots
                where e lt currentSlot}
```

```
  isEmpty() as Boolean
    return occupiedSlots = {}

  isFull() as Boolean
    return occupiedSlots = allSlots
  isClosedAndStuck() as Boolean
    return doorIsStuck and not doorIsOpen
  isOpenAndStuck() as Boolean
    return doorIsStuck and doorIsOpen
  //return the maximum element from a set
  setmax(s as Set of Integer) as Integer
    return the e | e in s where
                (forall d in s holds e gte d)
  //return the minimum element from a set
  setmin(s as Set of Integer) as Integer
    return the e | e in s where
                (forall d in s holds e lte d)
```

The following rule stipulates that only one of the arising errors is reported; the choice of error to report is left to the implementation.

```
class CHANGEDISC ...
  ReportAnyError(errs as
                  Set of (Boolean,String))
    choose (true, err) in errs
      result := ERR(err)
```

Remaining 8 UPnP actions.

```
class CHANGEDISC ...
  NextDisc()
    if not(isEmpty() or
           isOpenAndStuck()) then
      doorIsOpen := false
      if successors() ne {} then
        currentSlot := setmin(successors())
      else
        currentSlot := setmin(occupiedSlots)
    else
      ReportAnyError({(isEmpty(),"empty"),
                (isOpenAndStuck(),"stuck")})
  PrevDisc()
    if not(isEmpty() or
           isOpenAndStuck()) then
      doorIsOpen := false
      if predecessors() ne {} then
        currentSlot:= setmax(predecessors())
      else
        currentSlot:= setmax(occupiedSlots)
    else
      ReportAnyError({(isEmpty(),"empty"),
                (isOpenAndStuck(),"stuck")})
  RandomDisc()
    if not (isEmpty() or
            isOpenAndStuck()) then
      doorIsOpen := false
      choose slot in occupiedSlots
        currentSlot := slot
    else
      ReportAnyError({(isEmpty(),"empty"),
                (isOpenAndStuck(),"stuck")})
  OpenDoor()
    if not isClosedAndStuck() then
      doorIsOpen := true
    else
      result := ERR("stuck")
  CloseDoor()
    if not isOpenAndStuck() then
      doorIsOpen := false
    else
      result := ERR("stuck")
  ToggleDoor()
    if not doorIsStuck then
      doorIsOpen := not doorIsOpen
    else
      result := ERR("stuck")
  HasTrayDisc()
    result := BOOL(trayHasDisc())
  IsDoorOpen()
    result := BOOL(doorIsOpen)
```

The remaining two actions specify the possible behavior of the external environment. First one says that if the door is open then the disc can either be removed from the tray or placed on the tray. The second one specifies that the door may get stuck or unstuck.

```
class CHANGEDISC ...
  ToggleDiscOnTray()
    if doorIsOpen then
      if trayHasDisc() then
        occupiedSlots(currentSlot) := false
      else
        occupiedSlots(currentSlot) := true
  ToggleDoorStuck()
    doorIsStuck := not doorIsStuck
```

The test state is defined as follows.

```
structure CDState extends State
  occupiedSlots as Set of Integer
```

```
  currentSlot as Integer
  doorIsOpen as Boolean
  doorIsStuck as Boolean
  result as RESULT
```

The following extensions of the CHANGEDISC spec are needed below to set the test state, to get the test state and to dispatch on named actions.

```
class CHANGEDISC ...
  GetState() as State
    return CDState(occupiedSlots,
                   currentSlot,
                   doorIsOpen, doorIsStuck,
                   result) as State
  SetState(s as State)
    s1 = s as CDState
    occupiedSlots := s1.occupiedSlots
    currentSlot := s1.currentSlot
    doorIsOpen := s1.doorIsOpen
    doorIsStuck := s1.doorIsStuck
    result := s1.result
  Fire(a as Action)
    match a.name
      "AddDisc"          : AddDisc()
      "NextDisc"         : NextDisc()
      "PrevDisc"         : PrevDisc()
      "RandomDisc"       : RandomDisc()
      "OpenDoor"         : OpenDoor()
      "CloseDoor"        : CloseDoor()
      "ToggleDoor"       : ToggleDoor()
      "HasTrayDisc"      : HasTrayDisc()
      "IsDoorOpen"       : IsDoorOpen()
      "ToggleDiscOnTray": ToggleDiscOnTray()
      "ToggleDoorStuck" : ToggleDoorStuck()
  Actions() as Set of Action
    names = {"AddDisc", "NextDisc",
             "PrevDisc", "RandomDisc",
             "OpenDoor", "CloseDoor",
             "ToggleDoor", "HasTrayDisc",
             "IsDoorOpen",
             "ToggleDiscOnTray",
             "ToggleDoorStuck"}
    return {Action(n) | n in names}
```

Using the above definitions we can implement the rest of the required harness as follows.

```
class CDHARNESS ...
  slots as Set of Integer
  CDHARNESS(MaxSlot as Integer)
    slots = {1..MaxSlot}
  Initially() as State
    cd as CHANGEDISC = new CHANGEDISC(slots)
    return cd.GetState()
  Actions() as Set of Action
    cd as CHANGEDISC = new CHANGEDISC(slots)
    return cd.Actions()
  Fire(s as State, a as Action) as State
    cd as CHANGEDISC = new CHANGEDISC(slots)
    step
      cd.SetState(s)
    step
      cd.Fire(a)
    step
      return cd.GetState()
  GetHyperstate(s as State) as Hyperstate
    cd as CHANGEDISC = new CHANGEDISC(slots)
    step
      cd.SetState(s)
    step
      return cd.GetHyperstate()
```

The following is the entry point to execute the FSM generation algorithm with the CHANGEDISC harness. It prints out the number of hyperstates and the number of links between them.

```
run()
  step
    writeln("Input nr of slots.")
    write(">")
    MaxSlot = readln() //read a nr of slots
  step
    cd = new CDHARNESS(asInteger(MaxSlot))
  step
    genfsm = new GenFSM(cd as Harness)
    if genfsm = undef then
      throw SearchFailureException()
  step
    genfsm.main()
  step
    nodes = genfsm.hypers
    links =
      {(cd.GetHyperstate(s),a,
        cd.GetHyperstate(t)) |
          (s,a,t) in genfsm.transitions}
    writeln("nr of nodes = " +
            asString(size(nodes)) + "\n" +
            "nr of links = " +
            asString(size(links)))
```