Combined Algorithm for Approximating a Finite State Abstraction of a Large System

Margus Veanes
Microsoft Research
Redmond WA 98052, USA
margus@microsoft.com

Rostislav Yavorsky
Microsoft Research
Redmond WA 98052, USA
and Steklov Mathematical Institute
Moscow 119991 Russia
yavorsky@microsoft.com

Abstract

We study the problem of abstracting a finite state machine (FSM) from a given software system whose operational behavior is described by an abstract state machine (ASM). ASMs are typically infinite state systems and the problem of extracting a finite abstraction, the true FSM, *from an ASM with respect to a given abstraction function is in general undecidable. To approximate the true FSM we utilize the two key features of ASMs, that they are* executable *and have* precise mathematical semantics *that allows for a symbolic representation of the overall system behavior. On one hand, executability of ASMs is exploited for true computation path exploration; this yields an* under-approximation *of the true FSM. On the other hand, the symbolic representation of the system behavior enables us to use theorem proving to exclude impossible computation paths; this yields an* over-approximation *of the true FSM. In this paper we discuss an algorithm that combines both into a unified approach.*

1. Introduction

One way to deal with the state space explosion of systems with very large or even infinite state spaces is to abstract away from irrelevant details of the state and to consider a reduced finite state description of the original system. In our case the original system is an abstract state machine (ASM) [2] written in the ASM specification language AsmL [1]. The problem of generating a finite state machine (FSM) from an ASM is orginally studied in [4], where an algorithm is proposed that generates an FSM by *executing* the given ASM. The algorithm described in [4] has been extended is several ways and is implemented in the AsmL test tool, that is part of the AsmL toolkit [1]. The nodes of the FSM, called *hyperstates*, are equivalence classes of the states of the ASM, where the equivalence relation is an *indistinguishability* relation expressed through a finite number of Boolean conditions on the ASM state, i.e. two states are equivalent if they cannot be distinguished by the given conditions. In general the conditions may be not just Boolean, but arbitrary finite valued properties on the state. The *true FSM* has a trasition from hyperstate h_1 to hyperstate h_2 if and only if there is a transition in the ASM from some reachable representative of h_1 to some representative of h_2.

Generation of the true FSM is in general an undecidable problem [4]. The main property of the algorithm in [1] is that it yields an *under-approximation* of the true FSM, i.e. some links may be missing but no superfluous links are generated. One of the open areas mentioned in [4] is to study extensions to the algorithm that yield better approximations of the true FSM. The meaning of "better" depends of course on the context where the algorithm is applied. In the context of test case generation [1] for example, it is desirable not to have links that are not part of the true FSM because the generated test cases should correspond to feasible computation paths. On the other hand, in the context of model-checking, the more abstract model must *simulate* the original system so that certain properties of the original model are preserved. In order to ensure the simulation property, standard data-abstraction aglorithms yield an *over-approximation* of the ideal data-abstraction [3] (the true FSM).

In this paper we combine the algorithm in [4] with a symbolic analysis that uses theorem proving to rule out links that correspond to infeasible transitions. In other words the links that are ruled out are known to be outside the true FSM. Thus, all the remaining links correspond to an over-approximation of the true FSM. The difference between the over and the under-approximations can for example be used to estimate the quality of the approximations. One applica-

tion is to estimate the coverage of the test suite produced by the test tool mentioned above. The use of AsmL in scenario-based modeling is addressed in another paper in this volume [20]; the paper illustrates how the obtained model of scenarions can be finitized into an FSM that is subsequently used for test case generation. Here we look at the problem of measuring and improving the quality of such a generated FSM by using symbolic methods. This helps to measure the quality of the generated test suite.

The remainder of this paper is organized as follows. In Section 2 we provide some background definitions and related references. In Section 3 and 4 we explain the main parts of the algorithm. An example illustrating the main ideas of the approach is given in Section 5.

2. Background

2.1. The true FSM

An AsmL specification of a computational system contains the following parts:

- definitions of data types and static functions;
- declarations of state variables v_1, v_2, ..., v_s that characterize the state space of the considered system;
- rules that describe the transition relation of the system (i.e. how the variables change), notice that rules may be nondeterminsitic.

Let $P_1(v_1, \ldots, v_s), \ldots, P_n(v_1, \ldots, v_s)$ be Boolean conditions on the state variables. Two states $(a_1, \ldots, a_s)$ and $(b_1, \ldots, b_s)$ are *indistinguishable* if for each condition P_i one has

$$P_i(a_1, \ldots, a_s) \leftrightarrow P_i(b_1, \ldots, b_s).$$

A *hyperstate* is an equivalence class of the system states with respect to this equivalence relation. (Each hyperstate h_i is characterized by an appropriate complete conjunction H_i of the conditions and their negations.) Hyperstates are the nodes of the true FSM.

Given two hyperstates h_1 and h_2 the true FSM has a transition (h_1, h_2) if there exist two system states s_1, s_2 such that $s_1 \in h_1$, $s_2 \in h_2$, s_1 is reachable, and the transition (s_1, s_2) is enabled by the specification, i.e. some rule of the specification takes the system from state s_1 to state s_2.

If the considered system is initialized and s_0 is an initial state, then a hyperstate h of the corresponding FSM is initial if $s_0 \in h$.

Note that the true FSM may be (and usually is) nondeterministic even if the initial system is deterministic, and conversely, the true FSM may be deterministic even if the original system is nondeterministic. The notion of true FSM is similar to the notion of ideal abstraction used in the context of model checking [3].

2.2. Approaches to FSM extraction

Finite state machines are widely used in various areas, in particular, different test generation techniques are based on using finite state machines or finite labeled transition systems (see e.g. [13, 9]). Extraction of FSMs from model-based specifications for the purpose of test case generation was first studied in [8]. The approach in [8] is based on a finite partitioning of the state space using full disjunctive normal forms of conditions in the spec. The FSM is generated by symbolic analysis via theorem proving and produces an over-approximation of the true FSM.

An algorithm for FSM generation from AsmL specifications that is described in [4] is utilized in the testing tool that accompanies AsmL distribution [1]. In this paper we suggest an extension of the algorithm that uses both: model execution and static analysis via theorem proving. The approach we are considering here aims at getting the benefits of both approaches while avoiding the drawbacks.

3. Approximation of the true FSM

Given n Boolean conditions the number of hyperstates is at most 2^n. Thus, the true FSM is a subgraph of the complete graph with 2^n nodes. To derive the true FSM one has to decide for each edge of the complete graph if the corresponding transition is possible or not. To describe our approximation algorithm we will use the following definitions. Given an edge $e = (h_1, h_2)$ in the complete graph of hyperstates, we say that

- e is *green* if it belongs to the true FSM and an example of the corresponding computation path is found;
- e is *red* if we were able to prove that there is no representative s_1 of h_1 and no representative s_2 of h_2 such that the transition (s_1, s_2) is enabled by the specification;
- e is *blue* if for some representative s of h_1 there is a transition to some representative of h_2, yet it is not clear whether s is reachable from the initial state (on the other hand it follows that this edge is certainly not red);
- e is *gray* if it is neither green, nor red, nor blue, i.e. if the corresponding transition was not studied yet, or an attempt to decide it failed.

Notice that a blue edge may or may not belong to the true FSM due to the reachability requirement. Note also that not all true FSM edges are necessarily green, some of them may be blue and others may be gray.

3.1. Key procedures

The main part of the algorithm is described at the end of this section. We start with the main subroutines that are used by it.

Procedure: exhaustive model execution. Starting from a given state s of the system that corresponds to hyperstate h execute the specification and compute the corresponding hyperstate h' for the obtained next state s'.

Update the color of the edge (h, h'): if the procedure started from a reachable state then the edge should become green. Otherwise, if s was just randomly generated, (h, h') should become blue unless it was already green.

The execution contuinues until no new edges are found.

Since the considered system could be nondeterministic, one may have several subsequent states for s. In this case proceed with those that correspond to new edges. For advanced version of this procedure see [4].

Procedure: random state generation. Let the Boolean conditions used to define the hyperstates be fixed. Given a formula (formed from the conditions) try to generate a state satisfying this formula. To solve this probelm efficiently we can use the access driven filtering approach to parameter generation that is originally implemented in the Korat [5] tool and has been extended and implemented in the AsmL test tool [1]. As an additional improvement, the formula could be represented by a binary decision diagram [14] and the state variables could be ordered $(v_{i_1}, v_{i_2}, \ldots, v_{i_s})$ correspondingly to the order of the conditions in the BDD. Such a representation could help to catch unsatisfiable combinations early on during the state space exploration. Given a time limit (or another resource limit) the procedure may terminate with failure without generating any appropriate state.

Procedure: theorem proving. Given two Boolean properties (combinations of the conditions) Q_1 and Q_2 this procedure tries to prove that whenever Q_1 holds in a state s, Q_2 must hold in a subsequent state $s' = T(s)$. Usually it is done by formulating a proof goal

$$Q_1(s) \rightarrow Q_2(T(s)),$$

where s denotes the vector of state variables, T is a transition function. The proof attempt may fail due to a given time limit or undecidability.

A typical use case for this subroutine is the follwing. In order to classify an arrow (h_1, h_2) as red one has to prove that

$$H_1(s) \rightarrow \neg H_2(T(s)).$$

In other words, provided H_1 is true in a state s, H_2 will be false in the subsequent state of the system.

We have explored some heuristics that could reduce the number of calls to the theorem prover.

First of all, one could utilize the following condition dependencies. For example, one can perform standard data-flow analysis on the specification in order to determine the dependencies between the conditions. Given a system specification one could rather easily compute, for each state variable v_i, the set of state variables $Dep(v_i) = \{v_{i_1}, \ldots, v_{i_l}\}$ whose values completely determine the value of v_i in the next state. Using these variable dependencies one could compute the dependencies for the Boolean conditions, i.e. for each P_j find out those $\{P_{j_1}, \ldots, P_{j_m}\}$ that determine the truth value of P_j in the next state.

For each condition P_j one can consider all possible truth value combinations of the conditions it depends on. Once it is proved that, some combination of the properties implies that P_j is true (false) in the subsequent state, it gives us all the red arrows that follow from this implication.

Another way to produce several red arrows with one invocation of a theorem prover is the following. Given several implications that are expected to be valid, one may try to prove a more general property. For example, instead of considering separately the following goals:

$$\begin{aligned} \neg A \wedge B &\rightarrow \neg(A' \wedge B') \\ A \wedge B &\rightarrow \neg(A' \wedge B') \\ \neg A \wedge B &\rightarrow \neg(A' \wedge \neg B') \\ A \wedge B &\rightarrow \neg(A' \wedge \neg B') \end{aligned}$$

it is reasonable to try proving that

$$B \rightarrow \neg A'.$$

If the latter proof succeeds, all the prior implications follow. To realize this idea we use the notion of *multiarrow* as described below.

3.2. Lattice of FSM multiarrows

It was already mentioned above that each FSM hyperstate is characterized by a complete conjunction of the conditions and their negations. (Complete means that for any condition P_i either P_i or $\neg P_i$ appears in the conjunction.) Each incomplete conjunction characterizes the set of all hyperstates corresponding to the completions of the conjunction. Similarly, since each edge is characterized by a pair of complete conjunctions, a pair of incomplete conjunctions corresponds to a set of edges, we call it a *multiarrow*.

Given multiarrows (φ_1, ψ_1) and (φ_2, ψ_2) the first one is called to be more general than the second one if all conjuncts of φ_1 belong to φ_2 and all conjuncts of ψ_1 belong to

ψ_2. We extend the edge coloring to multiarrows: a multiarrow (φ, ψ) is *green* if for some reachable state s and transition T one has $\varphi(s)$ and $\psi(T(s))$, a multiarrow (φ, ψ) is *red* if

$$\forall \vec{s}(\varphi(\vec{s}) \rightarrow \neg\psi(T(\vec{s})).$$

The remaining colors are defined similarly.

It follows immediately that a multiarrow is green iff it is more general than some green arrow; a multiarrow is blue if it is not green, but at least one of less general arrows is blue; a multiarrow is red iff all of less general arrows are red; otherwise it is gray.

For our algorithm, the main important property of multiarrows coloring is the following: once an arrow is marked to be enabled (i.e. green or blue) all multiarrows that are more general are enabled too. And vice versa, if some multiarrow is proved to be imposible (red) all the less general arrows should be red too.

Thus, we come up with the following strategy for using the theorem prover: each time choose the most general multiarrow that remains gray. If the prover succeeds to make it red then all the less general arrows should be red too; if not, mark it with a tag indicating failure.

3.3. The main algorithm

Step 1 **Initialization.**

- Generate the complete graph with 2^n nodes corresponding to all Boolean combinations of the conditions; make all the edges gray;
- compute dependencies for the conditions and generate the graph of multiarrows;
- compute the concordant orderings for the conditions and the state variables for the random state generation procedure.

Step 2 **Generating green arrows.** Invoke the exhaustive model execution procedure starting from the initial state of the system.

Step 3 **Random generation of a hyperstate instances.** Consider the disjunction of all the unreached hyperstates. Invoke the random state generation procedure for this formula.

If each hyperstate was already reached, take hyperstates that have outgoing gray arrows.

Step 4 **Generating blue links.** Call the model execution procedure for each new state generated.

Step 5 **Generating red links.** For a maximal gray multiarrow (that was not treated before) invoke the theorem prover in an attempt to make it red.

Step 6 Go back to step 3 until all potential FSM edges are resolved/treated or the time limit is reached.

3.4. Remarks on the algorithm performance

Originally we had in mind applications where the number of the Boolean conditions is realtively small, so that the number of hyperstates and edges would be easily visualizable. For small values one could expect rather good performance.

Note also that most of the steps of the algorithm are independent, so the performance could be significantly improved by using parallel execution of different parts: random state generation for different hyperstates, execution of different computation paths, treating different arrows with theorem prover.

4. The algorithm output

This is the output of the algorithm.

- Under-approximation of the true FSM (the set of green arrows). This result is the same as for the FSM generation algorithm described in [4]. The important advantage of our approuch is that it allows to estimate the quality of the approximation.
- Overapproximation of the true FSM — the set of all not red arrows (i.e. green, blue, and gray ones). This graph might be useful for software monitoring: each correct step of the system should correspond to some edge in the overapproximation.
- Red arrows — formally verified properties of the system. The analysis of the generated red arrows may help do derive new properties of the system: transition invariants an more complex properties in terms of preconditions and postconditions (see [11] for a similar approach).

5. Railroad Crossing Example

The following example is a model of a single track railroad crossing. The model has been adapted to AsmL together with a model of the environment from its original realtime ASM model in [16, 17].

Data types and function definitions:

```
enum Status
  Coming
  Crossing
  Empty
enum Signal
  Open
  Close
```

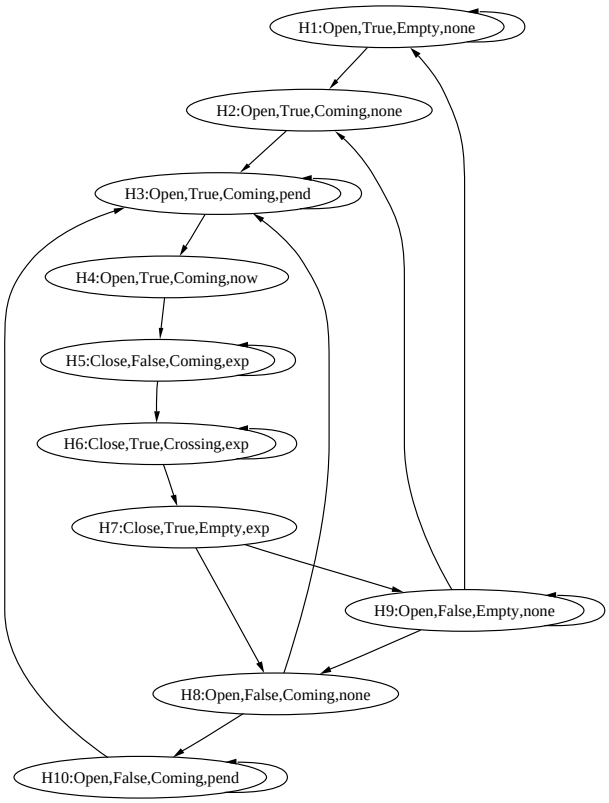

Figure 1. The green links of the FSM generated by *asmlt* from the Railroad model.

```
structure Infinite

function less(t1 as Integer,
    t2 as Integer or Infinite) as Boolean
  return t2 is Infinite or else t1< (t2 as Integer)
function SafeToOpen() as Boolean
  return (TrackStatus() = Empty) or
          less(CurrentTime, Deadline)
function Expired(timer as Integer) as Boolean
  timer <= CurrentTime
function TrackStatus() as Status
  if Expired(TrackTimer) then Empty
  elseif Expired(TrackTimer - CrossingTime)
    Crossing
  else Coming
function GateIsOpen() as Boolean
  (Direction = Open) and Expired (GateTimer)
function GateIsClosed() as Boolean
  (Direction = Close) and Expired (GateTimer)
```

There are four constant positive integer valued parameters of the model: 1) `ComingTime`, the time in which the train can reach the crossing from the moment it appears on the track (i.e. triggers a track sensor); 2) `CrossingTime`, the time it takes for the train to pass the crossing; 3) `OpeningTime`, gate opening time; 4) `ClosingTime`, gate colsing time. An additional constraint is that the durations are at least 3 time units each and that `ClosingTime + 2 < ComingTime`. The particular values chosen below for the constants are ignored by the symbolic analysis, but must be present in order to execute the model.

```
const ComingTime = 9
const CrossingTime = 12
const OpeningTime = 4
const ClosingTime = 4
var CurrentTime as Integer = -1
var TrackTimer as Integer = -1
var GateTimer  as Integer = -1
var Direction as Signal = Open
var Deadline  as (Integer or Infinite) =
  Infinite()
```

The main program of the model:

```
procedure TheCompleteStep(randomTrain as Boolean)
  if Direction = Open and not SafeToOpen() then
    Direction := Close
    GateTimer := CurrentTime + ClosingTime
  elseif Direction = Close and SafeToOpen() then
    Direction := Open
    GateTimer := CurrentTime + OpeningTime
  if TrackStatus() = Coming
     and (Deadline is Infinite) then
    Deadline := CurrentTime +
                (ComingTime - ClosingTime) - 1
  elseif TrackStatus() = Empty
         and not (Deadline is Infinite) then
    Deadline := Infinite()
  if TrackStatus() = Empty and randomTrain then
    TrackTimer := CurrentTime +
                  ComingTime + CrossingTime
  CurrentTime := CurrentTime + 1
```

We consider four hyperstate properties: 1) the diretion of the train (`Direction`); 2) expiration status of the gate timer (`Expired(GateTimer)`); 3) the status of the track (`TrackStatus()`); 4) the status of whether deadline for closing the gate has been reached:

```
if Deadline is Infinite then "none"
elseif Deadline as Integer>CurrentTime then "pend"
elseif Deadline=CurrentTime then "now"
else "exp" //expired
```

We used the AsmL tester tool [1] to generate green and blue links of the FSM, i.e. links whose discovery required execction of the model. The green links discovered this way are shown in Figure 1. The property values of the discovered hyperstates are illustrated by node labels. The initial state belongs to `H1`. For the symbolic analysis part we translated the model into HOL by using the translation scheme introduced in [18]. In the following we mention only some highlights, details can be found in [19].

After completing step 2, how close is the generated FSM to the true FSM? In order to answer to this question we went through the remaining steps. In this example we restricted our attention to outgoing links from the discovered hyperstates. First we generated random represenatives in the already reached hyperstates, and in this way discovered 7 new

hyperstates and 13 blue links (i.e. links that are possible but not necessarily reachable). We then used HOL to show that these were the only possible ougoing links. Out of the new links and hyperstates, one hyperstate and 3 links turned out to be green. Four of the remaining hyperstates were unsafe in the sence that they allowed the train to be crossing while the gate is not completely closed. We believe that these hyperstates are in fact not reachable, but we didn't prove so. The last two hyperstates are not unsafe but we believe that they too are not reachable

Initially, when we started out with this example we had identified only the first three properties and we believed that we had discovered all the relevant hyperstates and that our FSM was the true FSM. It was not until we used the symbolic analysis that made us realize that our properties were not adequate (we needed the fourth property to make the partitioning of state space more precise). The number of reasonable links *decreased* as a result. Further analysis lead to the discovery of the one missing true hyperstate and the three missing green links. See the technical report for more details [19].

Acknowledgements

We are thankful to Yuri Gurevich, Mike Barnett, Colin Campbell, Wolfgang Grieskamp, Lev Nachmanson, Wolfram Schulte, and Nikolai Tillmann for the productive cooperation.

Great thanks to Vladimir N. Krupski (Moscow University) for many helpful comments on the text.

References

[1] Foundations of Software Engineering Group, Microsoft Research. Abstract state machine Language, Website: http://research.microsoft.com/fse/AsmL.

[2] Abstract State Machines: A Formal Method for Specification and Verification. Website: http://www.eecs.umich.edu/gasm/

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

[4] W. Grieskamp, Y. Gurevich, W. Schulte, M. Veanes. Generating Finite State Machines from Abstract State Machines. *ISSTA 02, Software Engineering Notes* 27(4)112-122, 2002.

[5] C. Boyapati, S. Khurshid and D. Marinov, Korat: Automated testing based on Java predicates, *ISSTA 02, Software Engineering Notes* 27(4)123-133, 2002.

[6] A. Gargantini and E. Riccobene, "Encoding Abstract State Machines in PVS". In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, eds., International Workshop on Abstract State Machines, Monte Verita, Switzerland, Local Proceedings, TIK-Report 87, Swiss Federal Institute of Technology (ETH) Zurich, March 2000, 152–173.

[7] G. Schellhorn and W. Ahrendt, "Reasoning about Abstract State Machines: The WAM Case Study", Journal of Universal Computer Science, vol. 3, no. 4 (1997), 377–413.

[8] J. Dick and A. Faivre, Automating the generation and sequencing of test cases from model-based specifications, Proc. *FSE93, LNCS* 670, p. 268–284, Springer, 1993.

[9] D. Lee and M. Yannakakis, Principles and methods of testing finite state machines – a survey, *Proceedings of the IEEE*, volume 84, number 8, p. 1090-1123, IEEE, Berlin, 1996.

[10] C. Flanagan, K.R.M. Leino, M. Lillibridge, G. Nelson, J.B. Saxe, and R. Stata. "Extended Static Checking for Java." PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, Berlin, Germany June 2002.

[11] Toh Ne Win and M. Ernst, *Verifying distributed algorithms via dynamic analysis and theorem proving.*MIT Laboratory for Computer Science technical report 841, (Cambridge, MA), May 25, 2002.

[12] D. K. Peters and D. L. Parnas, *Requirements-based Monitors for Real-Time Systems.* IEEE Transactions on Software Engineering, vol. 28, no. 2, 2002.

[13] 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.

[14] R. Drechsler and B. Becker, *Binary Decision Diagrams - Theory and Implementation.* Kluwer Academic Publishing, 1998.

[15] The HOL System Reference. http://hol.sourceforge.net/

[16] Y. Gurevich and J. K. Huggins, "The Railroad Crossing Problem: An Experiment with Instantaneous Actions and Immediate Reactions," in Computer Science Logic, Selected papers from CSL'95, ed. H.K. Büning, Springer Lecture Notes in Computer Science 1092, 1996, 266–290.

[17] D. Beauquier and A. Slissenko, "The Railroad Crossing Problem: Towards Semantics of Timed Algorithms and their Model-Checking in High-Level Languages", in M. Bidoit and M. Dauchet, eds., "Proceedings of TAPSOFT'97: Theory and Practice of Software Development, 7th International Joint Conference CAAP/FASE", Springer Lecture Notes in Computer Science 1214, 202-212

[18] R. Yavorsky, Translation of AsmL to HOL. MSR technical report, MSR-TR-2003-22.

[19] M. Veanes and R. Yavorsky, Finite approximation of AsmL models through execution and symbolic analysis. MSR technical report, 2003.

[20] M. Barnett, W. Grieskamp, Y. Gurevich, W. Schulte, N. Tillmann and M. Veanes. Scenario-oriented Modeling in AsmL and its Instrumentation for Testing. ICSE/SCESM workshop 2003, Portland. In this volume.