Test Generation for Sequential Nets of Abstract State Machines with Information Passing

Paolo Arcaini[a], Angelo Gargantini[a]

[a]*Dipartimento di Ingegneria, Università degli Studi di Bergamo, Italy*

Abstract

Model-based test generation consists in deriving system traces from specifications of systems under test. There exist several techniques for test generation, which, however, may suffer from scalability problems. In this paper, we assume that the system under test can be divided in several subsystems such that only one subsystem is active at the time. Moreover, each subsystem decides when and to which other subsystem to pass the control, by also initializing the initial state of the next subsystem in a desired way. We model these systems and we show how it is possible to generate tests in a very efficient way that exploits the division of the entire system in subsystems. Test generation for the whole system is performed by visiting each subsystem and generating tests for it. The tests are combined in order to obtain valid system traces. We show how several visiting policies influence the completeness of the test generation process.

Keywords: Test case generation, state explosion problem, information passing, abstraction, State Machines

1. Introduction

Model-based automatic test generation consists in automatically generating tests from (abstract) models of systems under test. In this way, models and specifications are reused for software testing without the need for testers to write test suites by hand. The advantage of using models instead of code, is mainly the possibility to use specifications as test oracles and that specifications provide an abstract view of the system without all the implementation details. Although model-based test generation techniques have been successfully employed [1, 2, 3] even for complex systems, the scalability of these approaches is still a challenge.

We have worked on test generation from Abstract State Machines (ASMs) and used a tool for test generation for several years. However, since the test generation algorithm is based on model checking [4], one of the main obstacles has been the scalability of the approach and soon we encountered the well known *state space explosion problem*. Indeed, the problem of the model checking method is that the computational complexity increases exponentially together with the size of the model. Several techniques exist to overcome this limitation, like symbolic representation of states, compact storing of states, and efficient state space exploration. However, these techniques may still fail or weaken the coverage of the state space.

On the other hand, the system under test may have some peculiarities that can be exploited to limit the state explosion. We focus on systems that are composed of several subsystems that pass the control to each other such that only one subsystem is active at any time. This topology can be exploited for generating the test sequences over the single subsystems and combining them later, instead of generating the tests over the entire system. So, since the state space exponentially grows with the size of the system, decomposing the system exponentially reduces the complexity of the problem.

In this paper, we extend the approach in [5] by allowing information passing among the machines: the active machine decides the next machine and also its initial state by setting some location values as in classical value-passing of programming languages. This situation often occurs when modeling complex systems. For instance, the reader can think of a robotic system in which multiple small robots work in the same environment and pass to each other a job to be completed. The same scenario is common also in programming: a program is divided in multiple subprograms, but at every time only one subprogram is active and every subprogram calls another subprogram by passing some information.

Such systems can be modeled in an abstract way as *sequential net*s of ASMs, defined in Sect. 3, that are sets of ASMs having some features including that only one ASM is active at every time.

The test generation for the entire system modeled by a whole unique specification may be infeasible, but the topology of the system can be exploited by the test generation algorithm. In this paper, we present a technique in Sect. 4, that builds tests for single submodels and combine them in order to obtain valid system traces. Differently from [5], the test generation and test combination are performed at the same time, in order to visit the ASM only starting from valid initial states. We present several policies in which the ac-

tivities of test generation and combination can be performed together. The *basic* strategy implements a classical depth first search, while the *retrying* method performs some extra visits in order to improve the testing coverage. Moreover, we present a *backward* search which is able to build tests by backward visiting the net and possibly improving the coverage.

The paper is organized as follows. Sect. 2 presents the ASM formalism and the use of model checkers for test generation. In Sect. 3 we formalize the concept of sequential nets of ASMs. Sect. 4 reports the three strategies we propose for generating test suites for sequential nets. Sect. 5 presents the relations existing between the three strategies. Sect. 6 describes the experiments conducted on three versions of the running case study that we use throughout the paper. Sect. 7 discusses the limitations of the presented approach. Sect. 8 relates our work with similar contributions, and Sect. 9 concludes the paper.

2. Background

2.1. Abstract State Machines

Abstract State Machines (ASMs) [6] are an extension of FSMs, where unstructured control states are replaced by states with arbitrary complex data. The *states* of an ASM are multi-sorted first-order structures, i.e., domains of objects with functions and predicates defined on them. *Static* functions never change during any run of the machine. *Dynamic* functions are distinguished between *monitored* (only read by the machine and modified by the environment), and *controlled* (read and written by the machine).

ASM states are modified by *transition relations* specified by "rules" describing the modification of the functions interpretation from one state to the next one. There is a limited but powerful set of *rule constructors* including guarded actions (`if-then`) and simultaneous parallel actions (`par`). The constructor `choose` expresses nondeterminism in a compact way. A rule can be declared with a name (*macro* rule) and called in another rule simply by its name.

A *computation* of an ASM is a finite or infinite sequence $s_0, s_1, \ldots, s_n, \ldots$ of states of the machine, where s_0 is an initial state and each s_{i+1} is obtained from s_i by executing the machine (unique) *main rule*. An ASM can have more than one *initial state*. Because of the nondeterminism of the choose rule and of the environment moves, an ASM can have several different runs starting in the same initial state.

An ASM state s_i is represented by a set of couples (*location*, *value*). ASM *locations*, namely pairs (*function-name*, *list-of-parameter-values*), represent the abstract ASM concept of basic object containers (memory units). Location *updates* represent the basic units of state change and they are given as assignments, each of the form $loc := v$, where *loc* is a location and v its new value.

2.2. Test generation for ASMs

In model based testing [2, 1], the specification describing the expected behavior of the system is used as a test oracle to assess the correctness of the implementation. Tests are derived from specifications and used generally in conformance testing. In the following we give some basic definitions about test generation from ASMs.

Definition 1. A *test sequence* (or *test*) is a finite sequence of states $s_1, \ldots, s_n$ whose first element s_1 is an initial state, and each state s_i (with $i \neq 1$) follows the previous one s_{i-1} by applying the transition rules. The *test length* is given by the number of states of the sequence. A *test suite* (or *test set*) is a finite set of test sequences.

Informally, a test sequence is a partial ASM run and represents an expected system behavior. It contains both the inputs (monitored locations) and the expected outputs (controlled locations), i.e., the specification is used as test oracle. In MBT, specifications are also usually used to define testing criteria, that determine if a test suite is adequate to test a software. In ASMs, test adequacy can be measured as follows.

Definition 2. A *test predicate* is a formula over the state and determines if a particular testing goal is reached. A coverage criterion C is a function that, given a formal specification, produces a set of test predicates. A test suite TS satisfies a coverage criterion C if each test predicate generated with C is satisfied in at least one state of a test sequence in TS.

Note that a test sequence can cover many test predicates in different states. Several coverage criteria for ASMs have been defined in [4]. One of the basic criteria for ASMs is the *rule coverage*. A test suite satisfies the *rule coverage* criterion if, for every rule r_i, there exists at least one state in a test sequence in which r_i fires and there exists at least one state in a test sequence in which r_i does not fire.

2.3. Test generation for ASMs by Model Checking

In order to build test suites satisfying some coverage criteria, several approaches have been defined. In this paper we use a technique based on the capability of model checkers to produce counterexamples [7]. The method consists of three steps:

1. The test predicates set $\{tp_1, \ldots, tp_m\}$ is derived from the specification according to the desired coverage criteria;
2. The specification is translated into the language of the model checker;
3. For each test predicate tp_i the *trap property* $\Box\neg tp_i$ is proved, where $\Box$ means *always*. If the model checker finds a state s where tp_i is true, it stops and returns as counterexample a state sequence leading to s: such sequence is the test covering tp_i. If the model checker explores the whole state space without finding any state where the trap property is false, then the test predicate is said *infeasible* and it is ignored. In the worst case, the model checker terminates without exploring the whole state space and without finding a violation of the trap property (i.e., without producing any counterexample), usually because of the state explosion problem. This case is *inconclusive* since the user does not know if either the trap property is true (i.e., the test is infeasible), or it is false (i.e., there exists a sequence that reaches the goal).

In this paper we use the Asmeta framework [8] and its ATGT tool [9], based on the model checker SPIN [10]. Note that any framework supporting the model checking of ASMs, either directly [11] or through a mapping into the syntax of a model checker [12, 13, 14], could be used in our approach.

3. Sequential Nets of Abstract State Machines

We focus our attention on those systems that are composed of several subsystems that pass the control and some information to each other, so that only one subsystem is active at any time. Usually, in order to describe such kind of systems, a model of each subsystem is developed. A model of coordination is needed for representing the execution of the entire system, i.e., the activation/deactivation of subsystem models according to their local decisions.

An example is that of web applications. In a web application only one web page is active at any time, and the active page *decides* which is the next page to be displayed. A page can pass to the next one some information (like

the shopping cart or the session ID). The coordination is performed by the web browser and the web server that are responsible of closing the current page and visualizing the next one (passing the control among pages).

3.1. Definition of sequential net of ASMs

We assume that each component of the system is modeled with an ASM and we introduce the notion of sequential net of ASMs as follows.

Definition 3. A *sequential net of machines* is a set of Abstract State Machines $\{M_1, \ldots, M_n\}$ such that:

1. there exists a unique initial machine M_1,
2. only one machine is active at any time,
3. the active machine decides when and to which machine the control is passed,
4. each machine has its initial state selected by the previous machine,
5. the net is connected, i.e., each machine is reachable from the initial machine.

A sequential net of ASMs allows one to model a set of machines that run in sequence one after another, pass the control and some initial information to each other, and share the same environment. We call the net *sequential* because only one machine is running at any time, so the machines are not concurrent; however, there may not be a unique sequence among the machines, since every machine can decide the next machine depending on local decisions. We can represent a sequential net by a graph, where each node is a machine and an arc is a possible transfer of control between two machines.

Definition 4 (Control passing state)**.** Let S^i be the set of states of an ASM M_i. We define $S^i_{cp} \subseteq S^i$ the set of *control passing state*s in which M_i passes the control to another machine. For each machine M_i, function

$$nextM_i : S^i_{cp} \to \{M_1, \ldots, M_{i-1}, M_{i+1}, \ldots, M_n\}$$

signals, for each control passing state, the machine to which the control is passed.

Note that states keeping the control in machine M_i are not considered as *control passing*; so the codomain of $nextM_i$ does not include M_i.

Definition 5 (Initial state identification)**.** In order to model the information passing between machines M_i and M_j, we define the partial function

$$is_{ij} : S^i_{cp} \rightarrow S^j_o$$

where S^j_o is the set of initial states of machine M_j. is_{ij} is defined only for states $\{s \in S^i_{cp} | nextM_i(s) = M_j\}$, i.e., only for those states that actually pass the control to machine M_j.

Definition 6 (Net run)**.** A *net run* r for a sequential net is a (possibly infinite) sequence of valid runs $r_1, \ldots$ such that:

- r_1 is a run of the initial machine M_1;
- only the last run of a finite sequence can be infinite;
- for each (r_i, r_{i+1}) (with $i = 1, \ldots, k-1$), let r_i be a run of machine M_a, r_{i+1} be a run of machine M_b, fs be the final state of r_i and is be the initial state of r_{i+1}:
  1. $fs \in S^a_{cp}$: the final state of the run in M_a is *control passing*,
  2. $M_b = nextM_a(fs)$: M_a passes the control to M_b,
  3. $is \in is_{ab}(fs)$: M_a chooses the initial state is of M_b.

Note that different runs could belong to the same machine and some machines could not have corresponding runs.

In order to efficiently test a system modeled as a sequential net of ASMs, testing the single subsystems is not enough, since also the interaction among them must be tested. So, test suites for a net can not simply be the union of the tests for the single ASMs (as defined in Sect. 2.2) but they must contain tests that cover the whole application including the control and information passing. Therefore, Def. 1 must be extended for sequential nets of ASMs.

Definition 7 (Sequential net test)**.** A *test* for a sequential net is a *finite* net run, i.e., the sequence is finite and the last run is finite.

3.2. Modeling a sequential net with a set of ASMs

A possible way to model a sequential net is to model every single machine M_i in a classical way by a standard ASM and adding a means to signal the transfer of control and of information to the next machine as follows:

1. model every machine M_i as usual;
2. add a domain $AsmDomain = \{M_1, \ldots, M_n\}$ to each signature;
3. add a 0-ary function $currAsm$ of type $AsmDomain$ to each signature; in the initial state, $currAsm$ must assume the value M_i;

4. write the body of the main rule as follows:

   if currAsm $= M_i$ **then**
   r_mi[]
   endif

   where r_mi[] is a macro rule that contains the actions of the machine.
5. if a machine M_i wants to call a machine M_j (with $i \neq j$):
   (a) it has to perform the update

   currAsm $:= M_j$

   In this way, $S^i_{cp} = \{s \in S^i | [\![currAsm]\!]_s \neq M_i\}$ and $nextM_i(s) = [\![currAsm]\!]_s$.
   (b) it can identify an initial state of M_j by setting the value of some controlled functions of M_j. If M_i must set the function f in the initial state of M_j, M_i must have in its signature a function f and M_j must have in its signature a monitored function *fMon* having the same domain and codomain of f. *fMon* is only used in the initial state to initialise f, in the following way:

   function f = fMon

   When M_i passes the control to M_j, the value of the monitored function *fMon* is set to the value of the function f in machine M_i. In this way, machine M_i models the function is_{ij}.

Every machine M_i can be independently executed, for instance, by simulation for validation purposes. It asks the environment for (part of) its initial state, and it executes some useful actions until it changes the value of *currAsm*; after that, any other step of execution does not produce any change in the controlled part of the machine.

Example 1. Consider, for instance, the three ASMs shown in Codes 1, 2 and 3. They constitute a sequential net of ASMs (see Fig. 1). A *producer* produces *tokens* until it receives the signal that it must pass the control to some consumer (when the monitored function *consume* is *true*): if there are at least 100 tokens, it passes the control to a *fast consumer*, otherwise, if there are at least 20 tokens, it passes the control to a *slow consumer*. The fast consumer keeps consuming 5 tokens at a time while there are more than 50 tokens; then it passes the control to the slow consumer. The slow consumer consumes one token at a time until there are no more tokens.

Example 2. Given the ASM shown in Code 1, a test predicate generated with the *rule coverage* criterion (requiring that the update rule currAsm := FC is executed) is

currAsm = PR and consume and tokens >= 100

The test sequence that permits to satisfy the test predicate is

State 1	...	State 100	State 101
tokens = 0		tokens = 99	tokens = 100
currAsm = PR		currAsm = PR	currAsm = PR
consume = false		consume = false	consume = true

3.3. Product machine

Several validation and verification activities can be performed directly on the single machines. However, if we want to do a more general evaluation of the system (e.g., simulation of the transitions among machines, or test generation for the whole system), we must also provide a model of the coordination.

Instead of introducing a multi-agent scheduler, one possible simpler way is to merge all the machines in a unique *product* ASM as follows:

- the signatures of the machines are merged in a single signature:
  - there is only one copy of the *AsmDomain* domain and of the *currAsm* function;
  - controlled functions that are shared among the machines are declared only once;
  - domains that are shared among the machines are declared only once;
  - monitored functions of non-initial machines (i.e., M_j with $j = 2, \ldots, n$), used in the initial states to initialise the controlled functions shared among the machines (i.e., those declared with the *Mon* suffix), are not declared.
- all macro rules (except the main rules) of the single machines are included;
- in the main rule *r_main*, rules *r_mi*[] are individually called according to the value of the function *currAsm*;
- the initial state of the product ASM is the initial state of machine M_1;

```
asm producer // producer

signature:
    enum domain AsmDomain = {PR | FC | SC}
    domain SubIntDom subsetof Integer
    controlled currAsm: AsmDomain
    controlled tokens : SubIntDom
    monitored consume : Boolean

definitions:
    domain SubIntDom = {0..110}

    rule r_producer =
        if consume then
            if tokens >= 100 then
                currAsm := FC
            else if tokens>= 20 then
                currAsm := SC
            endif endif
        else
            tokens := tokens + 1
        endif

    main rule r_main =
        if currAsm = PR then
            r_producer[]
        endif

default init s0:
    function currAsm = PR
    function tokens = 0
```

Code 1: Producer

```
asm fastConsumer // fast consumer

signature:
    enum domain AsmDomain = {PR | FC | SC}
    domain SubIntDom subsetof Integer
    controlled currAsm: AsmDomain
    controlled tokens : SubIntDom
    monitored tokensMon: SubIntDom

definitions:
    domain SubIntDom = {0..110}

    rule r_fastConsumer =
        if tokens <= 50 then
            currAsm := SC
        else
            tokens := tokens - 5
        endif

    main rule r_main =
        if currAsm = FC then
            r_fastConsumer[]
        endif

default init s0:
    function currAsm = FC
    function tokens = tokensMon
```

Code 2: Fast consumer

```
asm slowConsumer // slow consumer

signature:
    enum domain AsmDomain = {PR | FC | SC}
    domain SubIntDom subsetof Integer
    controlled currAsm: AsmDomain
    controlled tokens : SubIntDom
    monitored tokensMon: SubIntDom

definitions:
    domain SubIntDom = {0..110}

    rule r_slowConsumer =
        if tokens > 0 then
            tokens := tokens - 1
        endif

    main rule r_main =
        if currAsm = SC then
            r_slowConsumer[]
        endif

default init s0:
    function currAsm = SC
    function tokens = tokensMon
```

Code 3: Slow consumer

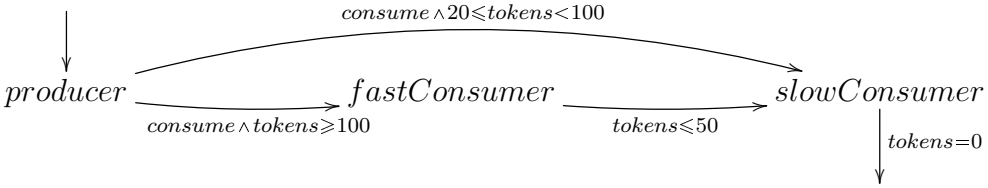

Figure 1: Sequential net *PCseqNet1*

- for each machine M_i, every time the control is passed to M_i (i.e., *currAsm* is updated to M_i), a rule r_initi[] is executed, that sets the *private* signature of M_i to the initial state of M_i. The *private* signature of a machine M_i is composed by the controlled functions of M_i that are not shared with any other machine.

Example 3. Given the sequential net shown in Fig. 1 (composed by machines shown in Codes 1, 2 and 3), the product machine is the one shown in Code 4. A test predicate for the product machine, generated with the *rule coverage* criterion (requiring that the update rule currAsm := SC in the macro rule *r_producer* is executed), is

currAsm = PR and consume and tokens < 100 and tokens >= 20

The test sequence that permits to satisfy the test predicate is

State 1	State 2	...	State 20	State 21
tokens = 0	tokens = 1		tokens = 19	tokens = 20
currAsm = PR	currAsm = PR		currAsm = PR	currAsm = PR
consume = false	consume = false		consume = false	consume = true

4. Test Generation for Sequential Nets of ASMs

We want to reuse the techniques described in Sect. 2.3 to generate valid sequential net tests, by using only classical ASMs (as those described in Sect. 3.2 and Sect. 3.3). We aim to devise *sound* and *complete* techniques.

Definition 8. A test generation method is *sound* if each produced test sequence is a valid sequence for the sequential net.

```
asm PCseqNet1ProductMachine

signature:
    enum domain AsmDomain = {PR | FC | SC}
    domain SubIntDom subsetof Integer
    controlled currAsm: AsmDomain
    controlled tokens : SubIntDom
    monitored consume : Boolean

definitions:
    domain SubIntDom = {0..110}

    rule r_fastConsumer =
        if tokens <= 50 then
            currAsm := SC
        else
            tokens := tokens − 5
        endif

    rule r_slowConsumer =
        if tokens > 0 then
            tokens := tokens − 1
        endif

    rule r_producer =
        if consume then
            if tokens >= 100 then
                currAsm := FC
            else if tokens >= 20 then
                currAsm := SC
            endif endif
        else
            tokens := tokens + 1
        endif

    main rule r_main =
        par
            if currAsm = PR then
                r_producer[]
            endif
            if currAsm = FC then
                r_fastConsumer[]
            endif
            if currAsm = SC then
                r_slowConsumer[]
            endif
        endpar

default init s0:
    function currAsm = PR
    function tokens = 0
```

Code 4: Product machine of the sequential net *PCseqNet1* (see Fig. 1)

Definition 9. A test generation method is *complete* if all the feasible test predicates of all the single machines are covered by the test sequences produced by the method.

Generating test sequences using the product machine. The first idea is to derive the test sequences directly from the product machine that already contains all the interactions among subsystems. A test sequence t of the product machine is transformed in a sequential net test in the following way:

1. the sequence t is divided in k subsequences, such that in each subsequence t_i the value of *currAsm* is constant and, given two consecutive subsequences t_i and t_{i+1}, the values of *currAsm* in t_i and in t_{i+1} are different;
2. given a subsequence t_i, let M_a be the value of *currAsm* in t_i. t_i is transformed in a run r_i of machine M_a by removing the functions that do not belong to machine M_a;
3. the concatenation of runs $r_1, \ldots, r_k$ is a sequential net run.

Theorem 1. *The* product machine *test generation method is sound and complete.*

PROOF. The technique explained before is sound since it can derive only valid sequential net runs from the test sequences generated from the product machine. The method is complete since all the feasible test predicates of the single machines are reachable in the product machine.

However, since test generation algorithms based on model checking may need to visit the whole state space of the model, the generation of test sequences from the product machine may suffer from the state explosion problem. It would be desirable to have a method in which the model checking must be executed only on the single machines and not on the product machine; indeed, as we have shown in [5], it is computationally easier to execute the model checker several times over small models, rather than executing it one time over a big model. The method should also provide a mechanism for combining the test suites produced for the single machines in a unique test suite to use for testing the whole system; the time taken by the combination of the test suites should be negligible.

4.1. Generating test suites in absence of information flow

A particular class of sequential nets is given by *independent* sequential nets in which there is no information passing between the machines, that only pass the control to each other. For this kind of nets, we proposed a test suite generation method in [5]. Given the independent sequential net $\{M_1, \ldots, M_n\}$, the technique works as follows:

1. A test suite TS_i is built for each machine M_i, using the test generation technique described in Sect. 2.3. Given the test sequences of machine M_i, we define *inner* as those sequences that terminate in a state in which *currAsm* is M_i, and *exiting* as those sequences that terminate in a state in which *currAsm* is M_j (with $j \neq i$). Inner test sequences keep the control of the net in the current machine, whereas exiting sequences pass the control to another machine.
2. The generated test sequences of test suites $\{TS_1, \ldots, TS_n\}$ constitute a graph, called *test sequence graph*, where every node is a machine and every arc is a test sequence. Inner test sequences are self loops of a node, whereas exiting test sequences are arcs between different nodes. Note that a test sequence graph has a similar structure to the

corresponding sequential net (for example Fig. 1): the nodes are the same, but a transition t is present in the test sequence graph only if there is a test executing t.

3. A recursive procedure traverses the test sequence graph starting from the initial machine. During the traversal, it concatenates the test sequences it encounters; the obtained sequences are valid sequential net runs and so tests for the sequential net (see Def. 7).

4.2. Combining the test generation and test concatenation for dealing with information passing

If the sequential net is not independent, i.e., the machines pass to each other some information, the technique briefly described in Sect. 4.1 is no longer valid, because the initial state of a machine M_j is determined by the machine M_i that passes the control to it. Therefore, it is no more possible to independently build the test suites for the single machines, because we do not know the initial state from which a machine will be executed. In this case, we have to combine the generation of the test sequences in the single machines with the visit of the sequential net for building the sequential net tests. We propose a general technique for building a test suite for a sequential net: it is based on a recursive procedure *testGenVisit* described in Alg. 1. The procedure is abstract since it contains two placeholders, `markVisited` and `hasToContinueVisit`, that must be filled in order to define a concrete test suite generation method. In this section we propose two methods: the *basic* method and the *retrying* method. They differ in the condition that stops the recursive visit. At the end of Alg. 1 we show how the two methods implements the two placeholders.

4.2.1. Basic *method*

Let's now describe the procedure, instantiated using the *basic* method. The recursive visit of a machine M_i works as follows:

1. M_i is marked as *visited* (line 1);
2. a set of tests *testsM* for covering the test predicates of machine M_i (initialised with the selected initial state *is*) is computed using the technique described in Sect. 2.3 (line 2);
3. if no test has been produced, the test sequence *prefix* is added to the test suite *testSuite* (line 4); otherwise, for each test in *testsM*:
   (a) a test *newTest* is obtained by concatenating the tests *prefix* and *test* (line 7);

Algorithm 1 General test suite generation method – Procedure *testGenVisit*

Require: the machine M_i to visit
Require: an initial state is of machine M_i
Require: a test sequence *prefix* that permits to reach machine M_i

```
1:  markVisited
2:  testsM ← allTestsFor(M_i, is)
3:  if testsM = ∅ then
4:    testSuite ← testSuite ∪ {prefix}
5:  else
6:    for test ∈ testsM do
7:      newTest ← prefix + test
8:      fs ← finalState(newTest)
9:      if fs ∈ S^i_cp then
10:       M_j ← nextM_i(fs)
11:       nextMis ← is_ij(fs)
12:       if hasToContinueVisit then
13:         testGenVisit(M_j, nextMis, newTest)
14:       else
15:         testSuite ← testSuite ∪ {newTest}
16:       end if
17:     end if
18:   end for
19: end if
```

Basic generation method:

1: `markVisited` : $visitedMs \leftarrow visitedMs \cup \{M_i\}$
12: `hasToContinueVisit` : $M_j \notin visitedMs$

Retrying generation method:

1: `markVisited` : $visitedMs \leftarrow visitedMs \cup \{(M_i, is)\}$
12: `hasToContinueVisit` : $(M_j, nextMis) \notin visitedMs \wedge hasUncovTps(M_j)$

(b) if the final state *fs* of *newTest* is a control passing state (line 9) and the selected machine M_j has not been visited yet (line 12), the recursive visit continues from M_j with the selected initial state *nextMis*, using *newTest* as prefix (line 13); otherwise *newTest* is added to the test suite (line 15).

```
rule r_slowConsumer2 =
    if tokens >= 30 then
        tokens := tokens − 2
    else if tokens > 0 then
        tokens := tokens −1
    endif endif
```

Code 5: Modified rule of the slow consumer in the sequential net *PCseqNet2*

The procedure *testGenVisit* is invoked using as argument the initial machine M_1 of the net, an initial state of M_1 randomly chosen, and the empty test sequence ϵ.

Note that the traversal of the graph representing the sequential net has linear complexity with the number of arcs and nodes (transitions and machines), and it requires a negligible amount of time with respect to the generation of the tests in the single machines (line 2 of the algorithm).

Theorem 2. *The* basic *method is sound.*

Proof. The *basic* method is sound because the tests are finite runs of the sequential net (see Def. 7). The finiteness of the runs is guaranteed by the fact that, during the traversal of the sequential net, already visited machines are not visited anymore, and by the fact that each machine has a finite number of test predicates and so a finite number of tests.

Theorem 3. *The* basic *generation method is not complete.*

Proof. In order to prove that the *basic* method is not complete, we give an example where some feasible test predicates are not covered by the produced tests. We consider a modified version of sequential net *PCseqNet1* that we call *PCseqNet2*. In *PCseqNet2* the slow consumer consumes 2 tokens (instead of only 1) while there are at least 30 tokens, as shown in Code 5 where we report the modified rule *r_slowConsumer*, called *r_slowConsumer2*. In this case, if machine *slowConsumer* is visited starting from machine *producer*, in the initial state *tokens* could be lower than 30 and so rule *r_slowConsumer2* would be only partially covered since the guard tokens >= 30 is never true.

4.2.2. Retrying generation method

The *retrying* method implements the *testGenVisit* procedure (see Alg. 1) with a less restrictive stopping condition for the recursive visit. Indeed a machine can be visited starting from different initial states. So, the procedure marks as visited the couple (*machine*, *initial state*) at line 1, and, at line 12, the recursive visit continues if next machine M_j has never been visited with the selected initial state *nextMis* and if there are still uncovered test predicates in machine M_j.

As for the *basic* method, also the *retrying* method invokes procedure *testGenVisit* using as argument the initial machine M_1 of the net, a randomly chosen initial state of M_1, and the empty test sequence ϵ.

Note that the *retrying* method guarantees to cover all the test predicates of rule *r_slowConsumer2*: it is visited (if not from *producer*, for sure from *fastConsumer*) with an initial value of *tokens* such that guard tokens >= 30 is true.

Theorem 4. *The* retrying *method is* sound.

Proof. The *retrying* method is sound because the tests are finite runs of the sequential net (see Def. 7). The finiteness of the runs is guaranteed by the fact that each machine has a finite number of test predicates and so a finite number of tests. Note that a machine M_i could have an infinite number of initial states but, since the number of tests that reach M_i is finite, M_i will be visited a finite number of times.

Theorem 5. *The* retrying *method is not complete.*

Proof. In order to prove that the *retrying* method is not complete, it is sufficient to show an example where some feasible test predicates are not covered by the produced tests. We consider a modified version of sequential net *PCseqNet1* that we call *PCseqNet3*. In *PCseqNet3* the slow consumer consumes one token at a time if the number of *tokens* is greater than 28, and consumes all the tokens at once when there are 27 tokens, as shown in Code 6 reporting the modified rule *r_slowConsumer* (called *r_slowConsumer3*). In this case, if machine *slowConsumer3* is visited with any initial state in which *tokens* is different from 27, the update rule that resets *tokens* to 0 can not be covered. Reaching such a state can be very difficult using the *retrying* method (and even more difficult using the *basic* method).

```
rule r_slowConsumer3 =
    if tokens = 27 then
        tokens := 0
    else if tokens > 28 then
        tokens := tokens − 1
    endif endif
```

Code 6: Modified rule of the slow consumer in the sequential net *PCseqNet3*

4.3. Backward *test generation*

In order to cover those test predicates that are not covered by the *basic* and the *retrying* methods, we introduce a technique for building a test for a given test predicate by backwards traversing the graph of the sequential net.

Alg. 2 shows a recursive procedure that, in order to cover a test predicate tp of machine M, builds a test sequence by starting from machine M and going backwards until the initial machine of the net is reached. The recursive visit of a machine M works as follows:

1. machine M is marked as *visited* (line 1);
2. if the test predicate tp is not feasible a *null* test sequence is returned (line 18); otherwise, if tp is feasible:
   (a) a test sequence is produced for covering it (line 3);
   (b) if M is the initial machine of the net, the test is returned (line 5); otherwise the procedure continues with the following points;
   (c) a test predicate *tpInit* is computed starting from the initial state $initState_M$ of the test (line 8); such test predicate describes the condition that must be satisfied by another machine of the net for reaching machine M and selecting $initState_M$ as initial state of M. The test predicate is $currAsm = M \wedge \bigwedge_{f \in InitFun} f = [\![f]\!]_{initState_M}$, where *InitFun* is the set of controlled functions that are initialized in the initial state through monitored functions, as explained in Sect. 3.2;
   (d) for each machine M_{in} that reaches M, if M_{in} has not been visited yet, it is recursively visited using *tpInit* as test predicate; in this way we check if machine M_{in} has a test sequence that permits to reach M and that selects the desired initial state $initState_M$;
   (e) as soon as a machine is found that can reach M with a not empty test *backTest* (line 12), the loop is quit and the concatenation of

Algorithm 2 *Backward* test sequence generation method – Procedure *backwardVisit*

Require: the machine M to visit
Require: the test predicate tp to cover
Ensure: the test t covers tp

```
1:  visitedMs ← visitedMs ∪ {M}
2:  if isFeasible(tp) then
3:    test ← getTest(tp)
4:    if M = M_1 then
5:      return test
6:    end if
7:    initState_M ← getInitState(test)
8:    tpInit ← buildTpFrom(initState_M, M)
9:    for M_in ∈ entering(M) do
10:     if M_in ∉ visitedMs then
11:       backTest ← backwardVisit(M_in, tpInit)
12:       if isNotNull(backTest) then
13:         return backTest + test
14:       end if
15:     end if
16:   end for
17: end if
18: return NULL
```

the test sequences *backTest* and *test* (line 13) is returned;

(f) if no machine is found, the *null* test sequence is returned (line 18).

Note that the *backward* method guarantees to cover the update rule that resets *tokens* in machine *slowConsumer3* (see Code 6). Both the *basic* and the *retrying* methods may not reach machine *slowConsumer3* with the required initial state with *tokens* initialised to 27. The *backward* method, instead, finds the test, since it starts building the test directly from machine *slowConsumer* in which the initial state is not fixed (no value for *tokensMon* is given), so that all the initial states are considered.

Theorem 6. *The* backward *method is* sound.

Proof. The *backward* method is sound because the produced test is a finite run of the sequential net (see Def. 7). The finiteness of the run is guaranteed

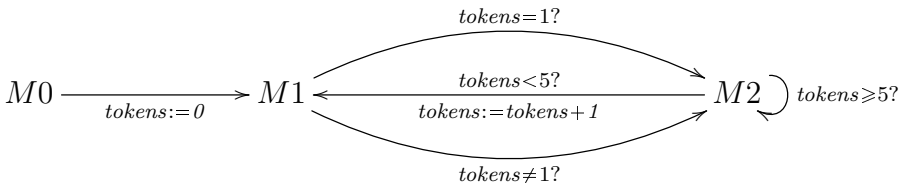

Figure 2: *FiveTokens* sequential net

```
asm M0

signature:
  enum domain AsmDomain = {M0 | M1 | M2}
  domain SubIntDom subsetof Integer
  controlled currAsm: AsmDomain
  controlled tokens : SubIntDom

definitions:
    domain SubIntDom = {0..5}

    rule r_m0 =
        par
            currAsm := M1
            tokens := 0
        endpar

    main rule r_main =
        if currAsm = M0 then
            r_m0[]
        endif
```

Code 7: M0

```
asm M1

signature:
  enum domain AsmDomain = {M0 | M1 | M2}
  domain SubIntDom subsetof Integer
  controlled currAsm: AsmDomain
  controlled tokens : SubIntDom
  monitored tokensMon : SubIntDom

definitions:
    domain SubIntDom = {0..5}

    rule r_m1 =
        if tokens = 1 then
            par
                //do some actions
                currAsm := M2
            endpar
        else
            par
                //do some other actions
                currAsm := M2
            endpar
        endif

    main rule r_main =
        if currAsm = M1 then
            r_m1[]
        endif

default init s0:
  function tokens = tokensMon
```

Code 8: M1

```
asm M2

signature:
  enum domain AsmDomain = {M0 | M1 | M2}
  domain SubIntDom subsetof Integer
  controlled currAsm: AsmDomain
  controlled tokens : SubIntDom
  monitored tokensMon : SubIntDom

definitions:
    domain SubIntDom = {0..5}

    rule r_m2 =
        if tokens < 5 then
            par
                currAsm := M1
                tokens := tokens +1
            endpar
        else
            //do some other actions
        endif

    main rule r_main =
        if currAsm = M2 then
            r_m2[]
        endif

default init s0:
  function tokens = tokensMon
```

Code 9: M2

by the fact that, during the backward visit, already visited machines are not visited anymore.

Theorem 7. *The* backward *generation method is not complete.*

Proof. In order to prove that the *backward* method is not complete, we show that it is not able to produce tests for some particular test predicates. Consider, for example, the sequential net shown in Fig. 2, whose machines are shown in Codes 7, 8 and 9. In such sequential net, the function *tokens* is incremented only by $M2$ when it passes the control back to $M1$. After having passed the control 5 times, $M2$ performs some internal activities. Let us introduce the test predicate $tp_1 : currAsm = M1 \wedge tokens = 1$ related

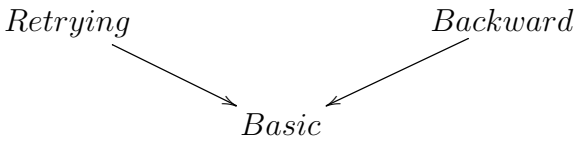

Figure 3: Subsumption relations between the test generation methods

to the **then** branch of the conditional rule in rule r_m1 of machine $M1$. For covering the test predicate tp_1, machine $M1$ must be visited starting from an initial state where *tokens* is *1*. The *backward* method can not cover *tp* because it behaves as follows:

1. it starts the visit from machine $M1$;
2. it visits backwards machine $M2$, since it is the only machine that reaches $M1$ setting *tokens* to *1*. $M2$ decides to pass the control to $M1$ only if *tokens* is less than 5; in particular, it passes the value 1 when *tokens* has been initialized to 0;
3. it tries to visit backwards machine $M1$, since it is the only machine that reaches $M2$ setting *tokens* to 0; however, since $M1$ has already been visited, the backward visit is stopped, without producing a test for tp_1.

Note that tp_1 is feasible and would be covered by the retrying method (see proof of Thm. 12).

5. Subsumption relations between the test generation methods

Definition 10. We say that a test generation method Y subsumes another test generation method X if, when X finds a test that covers a test predicate *tp*, then also Y guarantees to find a test for *tp*.

Assuming that the model checker never returns inconclusive results, the subsumption relations existing between the three presented test generation methods are those depicted in Fig. 3.

Theorem 8. *The* retrying *method subsumes the* basic *method.*

Proof. The *retrying* method extends the *basic* method by permitting to visit a machine more than once, with different initial states. So, if a test is built for a test predicate *tp* by the *basic* method, a test for *tp* would be also built by the *retrying* method.

Theorem 9. *The* basic *method does not subsume the* retrying *method.*

PROOF. There are some test predicates that, although covered by the *retrying* method, may not be covered by the *basic* method. Consider, for example, the sequential net *PCseqNet2*. The *retrying* method assures to cover all the test predicates; the *basic* method, instead, may not cover the test predicates related to the **then** branch of the conditional rule of rule *r_slowConsumer2* (see Code 5), if it visits machine *slowConsumer* with an initial state in which function *tokens* is lower than 30.

Theorem 10. *The* backward *method subsumes the* basic *method.*

PROOF. If the *basic* method is able to build a test for covering a test predicate *tp* in machine M_i, it means that the test predicate is feasible, and that a net run exists from machine M_1 to machine M_i. Since the *backward* method assures to find tests for feasible test predicates and to reach the initial machine if a net run exists to reach machine M_i, it is proved that the *backward* method subsumes the *basic* method.

Theorem 11. *The* basic *and the* retrying *methods do not subsume the* backward *method.*

PROOF. As seen before, the *basic* and the *retrying* methods may not cover the test predicate $currAsm = SC \wedge tokens = 27$ of machine *slowConsumer3* (see Code 6) that, instead, is covered by the *backward* method.

Theorem 12. *The* backward *method does not subsume the* retrying *method.*

PROOF. Let us consider the *FiveTokens* sequential net (Fig. 2, machines in Codes 7, 8, and 9). We have shown in proof of Thm. 7 that the *backward* method can not cover the test predicate tp_1 related to the **then** branch of the conditional rule in rule r_m1 of machine $M1$. The *retrying* method can cover tp_1, since it can visit machine $M1$ starting from machine $M2$ that, when passing the control to $M1$, actually sets *tokens* to the desired value 1.

Since no subsumption relation holds between the retrying and the backward methods, one may wonder if their combination is complete, i.e., if all the feasible test predicates can be covered by at least one of the two methods.

Theorem 13. *The combination of the* retrying *and the* backward *generation methods is not complete.*

PROOF. In order to prove that the combination of the *retrying* and the *backward* method is not complete, we show that both methods cannot produce tests for some particular test predicates. Consider, for example, the sequential net *FiveTokens* shown in Fig. 2, whose machines are shown in Codes 7, 8 and 9. Let us consider the test predicate for covering the **else** branch of the conditional rule of rule r_m2 of machine $M2$; it can be covered when *tokens* is at least 5. The *retrying* method, in order to reach machine $M2$ initialising *tokens* to 5, should visit machine $M1$ six times: however, the third visit can not be executed since, by that time, all the test predicates of machine $M1$ are already covered (in Alg. 1 the guard at line 12 is false). Also the *backward* method requires to visit machine $M1$ six times; in this case the generation is stopped before the second visit of machine $M1$, since the guard at line 10 of Alg. 2 is false (machine $M1$ has already been visited).

6. Experiments

In order to have a preliminary comparison of the proposed visiting policies, we have generated test suites achieving structural coverage of the three Producer-Consumers sequential nets *PCseqNet1* (see Example 1), *PCseqNet2* (see Code 5), and *PCseqNet3* (see Code 6). We have run all the experiments on a Linux machine, Intel(R) Core(TM) i7, 4 GB RAM. For obtaining short counterexamples, we have used the breadth-first-search option in Spin. We have applied *monitoring*, i.e., when generating a test for a given test predicate, we check if it accidentally covers also other test predicates that we can mark as covered and ignore. All the reported results are the average of 100 runs of each experiment.

6.1. Comparison of the basic *and the* retrying *methods*

In the first experiment we want to compare the *basic* and the *retrying* methods. Table 1 shows the results obtained by the two methods over the three versions of the running example. It reports the number of covered and uncovered test predicates (and the total number), the time taken by the generation method, the number of tests, the total length of all the tests, and the average length of a test. We can notice that, for the first sequential net *PCseqNet1*, the two methods cover the same number of test predicates, since

	method	**covered tps**	**uncovered tps**	**# tps**	**time (sec.)**	**# tests**	**total length**	**avg. test length**
PCseqNet1	B	31.0	2.0	33.0	2.82	4.37	55.82	13.37
	R	31.0	2.0	33.0	3.37	4.25	51.42	12.1
PCseqNet2	B	33.08	3.92	37.0	3.41	4.5	53.39	12.08
	R	35.0	2.0	37.0	4.50	5.71	81.33	14.25
PCseqNet3	B	30.44	6.56	37.0	4.25	4.06	33.75	8.26
	R	32.0	5.0	37.0	6.36	5.31	76.49	14.35

Table 1: Results using the *basic* (B) and the *retrying* (R) methods (average of 100 runs of each experiment)

the *basic* method is powerful enough to cover all the feasible test predicates. We have independently proved that the two uncovered test predicates are infeasible. For the sequential net *PCseqNet2*, instead, the *retrying* method can cover more test predicates, since it permits to reach machine *slowConsumer* with two different initial states from which all the test predicates can be covered. The *basic* method, instead, could visit machine *slowConsumer* with an initial state in which *tokens* is lower than 30: from that initial state it is not possible to cover the test predicates related to the **then** branch of the conditional rule in rule *r_slowConsumer2*. For the sequential net *PCseqNet3*, the *retrying* method, since it is more powerful, can still cover more test predicates than the *basic* method; however, both methods can not cover some test predicates of machine *slowConsumer* (those related to the **then** branch of the conditional rule in rule *r_slowConsumer3*), since they do not reach that machine with the unique initial state (in which *tokens* is initialised to 27) that allows to cover those test predicates.

6.2. Backward *method evaluation*

In the second experiment, we want to evaluate the effectiveness of the *backward* method. We have executed the *basic* and the *retrying* methods (called *forward* methods) and then, for each uncovered test predicate, we have applied the *backward* method. Table 2 shows the results of the experiment.

		covered tps			uncovered tps	# tps	time (sec.)	# tests	total length	avg. test length	# tests BA
		forward method	BA	fw method + BA							
PCseqNet1	B + BA	31.0	0.0	31.0	2.0	33.0	5.75	4.55	54.99	12.38	0.0
	R + BA	31.0	0.0	31.0	2.0	33.0	6.43	4.55	55.49	12.75	0.0
PCseqNet2	B + BA	33.23	1.77	35.0	2.0	37.0	10.32	4.38	55.14	13.10	1.77
	R + BA	35.0	0.0	35.0	2.0	37.0	7.53	5.7	82.08	14.47	0.0
PCseqNet3	B + BA	30.08	4.92	35.0	2.0	37.0	18.77	3.93	29.62	7.46	4.92
	R + BA	32.0	3.0	35.0	2.0	37.0	15.92	5.23	73.41	13.88	3.0

Table 2: Results of the combination of the *basic* (B) and *retrying* (R) methods with the *backward* method (BA) (average of 100 runs of each experiment)

In addition to the fields reported in Table 1, it also reports the number of covered test predicates divided into those covered by the *forward* method and those covered by the *backward* method, and the number of tests generated with the *backward* method. For the sequential net *PCseqNet1*, using the *backward* method is useless, because the test predicates that have not been covered by the *forward* methods are actually infeasible. For the sequential net *PCseqNet2*, the *backward* method permits to cover, on average, almost two test predicates that have not been covered by the *basic* method; the *retrying* method, instead, is powerful enough to cover all the feasible test predicates, and so the *backward* method is useless in this case. For the sequential net *PCseqNet3*, the *backward* method permits to cover test predicates that have not been covered by both *forward* methods. The *forward* methods have not been able to reach the particular initial state from which those test predicates can be covered.

As expected, the experiments suggest that a *forward* method together with the *backward* method can improve the testing coverage at the expenses of an increase in test generation time. Note that, the *backward* method together with the *basic* method requires more time than together with the *retrying* method. The *backward* method is very expensive in terms of computation time, and therefore the more coverage is obtained by a *forward* method, the smaller the total time is.

7. Threats to validity

Since the approach is based on finite state model checking, some limitations exist on the set of specifications that can be verified: for example, only finite domains can be used. Moreover, some limitations are due to the translation tools: for example, some complex data structures (e.g., sets) are difficult to map in the model checker syntax and so they are not admitted.

Another source of incompleteness for the *basic* and *retrying* methods can be the use of *non-preservable* coverage criteria, i.e., criteria that can be satisfied without requiring the passage of control to another machine, although the passage is possible. In these cases, a machine could not be covered since it never receives the control from another machine. See [5] for a detailed discussion about *preservable* criteria which, however, are quite rare in practice.

In this paper, we do not experiment the scalability of our approach: experiments show that the use of the backward method is very expensive in terms of time and this may limit the scalability of our approach. We have already demonstrated for sequential nets without information passing that decomposing a system can greatly improve the scalability of test generation [5]. In the future, we plan to experiment scalability also for sequential nets with information passing and with the use of the backward method. We also plan to measure the incompleteness of the proposed methods for a wide set of sequential nets.

8. Related work

Our work presents a model-based testing (MBT) approach, in which tests are derived from formal specifications. For a survey on tools and techniques for MBT, using also other notations, see [3]. For instance, [1] combines the use of UML state machines and the formal notation B for test generation.

Besides our previous work [9, 4], another approach about test generation from ASMs is described in [15, 16]; they simulate an ASM and, at the same time, build a corresponding FSM and produce some test sequences. Differently from our approach, although they acknowledge that their approach suffers form the state explosion problem, they do not propose any solution to reduce it.

Our approach tries to mitigate the state space explosion problem during model checking for test generation. Traditionally several techniques attempt to solve the same problem for the verification of properties. They share the

concept of building an abstract version of the original system that preserves properties.

The *cone of influence* (coi) technique [17] reduces the size of the transition graph by removing from the model the variables that do not influence the variables in the property one wants to check. In [18] the cone of influence technique is used to reduce the state space of *fFSM* models, a variant of Harel's Statecharts; models that could not be verified before, have been verified successfully after its application. The *data abstraction* technique [17], instead, consists of creating a mapping between the data values and a small set of abstract data values; the mapping, extended to states and transitions, usually reduces the state space, but it may not preserve properties. In [19] a technique to iteratively refine an abstract model is presented. The technique assures that, if a property is true in the abstract model, so it is in the initial model; if it is false in the abstract model, instead, the *spurious* counterexample may be the result of some behavior in the abstract model not present in the original model. The counterexample itself is used to refine the abstraction so that the *wrong* behavior is eliminated.

A technique for sequential *modular* decomposition for property verification of complex programs is presented in [20]. The approach consists in partitioning the program into sequentially composed subprograms (instead of the typical solution of partitioning the design into units running in parallel). Based on this partition, the authors present a model checking algorithm for software that arrives at its conclusion by examining each subprogram in separation. Similarly to our approach, they identify *ending states* in the component where the computation is continued in another component and some information passed to the next subprogram. The algorithm then tries to formally prove the property in each component finding the necessary assumptions about the initial (entering) states of the component. The algorithm proceeds backwards until it finds that the property is true in every sub-component starting from any initial state of the system. Since the goal is formal verification, the algorithm must guarantee that the property holds in *any* state, while, since we want to find only a counterexample, we just need to find a path leading to interesting states.

For test generation, these techniques may need to be modified, since they do not have to preserve properties but counterexamples to be used as tests. The coi technique can be used as it is also for test generation, but it may not simplify our models, since the *currAsm* function, which is used in the test goals, may be influenced by all the functions.

An approach performing test generation by decomposing sequential *programs*, called SMART, is presented in [21]. Although it starts from programs instead of models, it proposes a sequential decomposition technique similar to ours: given a program calling several functions inside it, these called functions are tested in isolation and complete tests are composed only at the end. The main difference is that tests for sub-functions are not real tests but they are expressed as *summaries* using input preconditions and output postconditions, and then re-used when testing higher-level functions. The main advantage is that SMART is both sound and complete compared to monolithic test generation (like our product machine), while our approach is only sound. A disadvantage is that SMART must maintain the summaries and it can solve them only at the end. Sometimes constraints on some inputs can not be expressed (for instance a `hash` function) and sometimes all the collected constraints are very hard to solve, leaving some issues still open.

9. Conclusion and Future work

We have tried to address the state explosion problem in test generation by model checking. For sequential nets of ASMs, even in the presence of information passing, we are able to exploit the system structure and decompose it in order to facilitate the test generation. In order to obtain valid test sequences for the whole system, our approach performs test generation for single submachines together with the exploration of the graph representing the sequential net.

We assume that the designer keeps the models separated from the beginning; as future work, we plan to study a decomposition methodology able, if possible, to split an existing complex ASM in a sequential net of ASMs.

Although our method shows its great usefulness when used in combination with (explicit state) model checking for test generation, we believe that any test generation technique can benefit from dividing the model in submodels, even those techniques which do not suffer so much from the size of the model under test.

Regarding the incompleteness of the *backward* method, we plan to extend it by letting it visit the same machine several times with different initial states (similarly to what is done by the *forward retrying* method).

References

[1] M. Utting, B. Legeard, Practical Model-Based Testing: A Tools Approach, Morgan-Kaufmann, 2006.

[2] R. Hierons, J. Derrick, Editorial: special issue on specification-based testing, Software Testing, Verification and Reliability 10 (4) (2000) 201–202.

[3] R. M. Hierons, K. Bogdanov, J. P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, G. Lüttgen, A. J. H. Simons, S. Vilkomir, M. R. Woodward, H. Zedan, Using formal specifications to support testing, ACM Comput. Surv. 41 (2) (2009) 9:1–9:76.

[4] A. Gargantini, E. Riccobene, S. Rinzivillo, Using Spin to generate tests from ASM specifications, in: E. Börger, A. Gargantini, E. Riccobene (Eds.), Proceedings of the 10th International Workshop on Abstract State Machines, Advances in Theory and Practice (ASM 2003), Taormina, Italy, March 3-7, 2003, Vol. 2589 of Lecture Notes in Computer Science, Springer, 2003, pp. 263–277.

[5] P. Arcaini, F. Bolis, A. Gargantini, Test Generation for Sequential Nets of Abstract State Machines, in: J. Derrick, J. Fitzgerald, S. Gnesi, S. Khurshid, M. Leuschel, S. Reeves, E. Riccobene (Eds.), Proceedings of the Third International Conference on Abstract State Machines, Alloy, B, VDM, and (ABZ 2012), Pisa, Italy, June 18-21, 2012, Vol. 7316 of Lecture Notes in Computer Science, Springer, 2012, pp. 36–50.

[6] E. Börger, R. Stärk, Abstract State Machines: A Method for High-Level System Design and Analysis, Springer Verlag, 2003.

[7] G. Fraser, A. Gargantini, An evaluation of model checkers for specification based test case generation, in: ICST 2009, 1-4 April 2009, Denver, Colorado, USA, IEEE Computer Society, 2009, pp. 41–50.

[8] P. Arcaini, A. Gargantini, E. Riccobene, P. Scandurra, A model-driven process for engineering a toolset for a formal method, Software: Practice and Experience 41 (2) (2011) 155–166.

[9] A. Gargantini, E. Riccobene, ASM-Based Testing: Coverage Criteria and Automatic Test Sequence Generation, J.UCS 7 (2001) 262–265.

[10] G. Holzmann, The Spin model checker: Primer and Reference manual, Addison-Wesley Professional, 2003.

[11] M. Veanes, N. Bjørner, Y. Gurevich, W. Schulte, Symbolic Bounded Model Checking of Abstract State Machines, Int. J. Software and Informatics 3 (2-3) (2009) 149–170.

[12] P. Arcaini, A. Gargantini, E. Riccobene, AsmetaSMV: a way to link high-level ASM models to low-level NuSMV specifications, in: M. Frappier, U. Glässer, S. Khurshid, R. Laleau, S. Reeves (Eds.), Proceedings of the Second International Conference on Abstract State Machines, Alloy, B and Z (ABZ 2010), Orford, QC, Canada, February 22-25, 2010, Vol. 5977 of Lecture Notes in Computer Science, Springer, 2010, pp. 61–74.

[13] G. D. Castillo, K. Winter, Model checking support for the ASM high-level language, in: S. Graf, M. Schwartzbach (Eds.), Tools and Algorithms for Construction and Analysis of Systems, 6th International Conference, TACAS 2000, Vol. 1785 of Lecture Notes in Computer Science, Springer, 2000, pp. 331–346.

[14] R. Farahbod, U. Glässer, G. Ma, Model Checking CoreASM Specifications, in: A. Prinz (Ed.), Proceedings of the ASM'07, The 14th International ASM Workshop, 2007.

[15] W. Grieskamp, Y. Gurevich, W. Schulte, M. Veanes, Generating finite state machines from abstract state machines, in: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis, ISSTA '02, ACM, New York, NY, USA, 2002, pp. 112–122.

[16] W. Grieskamp, L. Nachmanson, N. Tillmann, M. Veanes, Test case generation from AsmL specifications, in: E. Börger, A. Gargantini, E. Riccobene (Eds.), Proceedings of Abstract State Machines, Advances in Theory and Practice, 10th International Workshop, ASM 2003, Taormina, Italy, March 3-7, 2003, Vol. 2589 of Lecture Notes in Computer Science, Springer, 2003, p. 413.

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

[18] S. Park, G. Kwon, Avoidance of State Explosion Using Dependency Analysis in Model Checking Control Flow Model, in: ICCSA 2006, Vol. 3984 of Lecture Notes in Computer Science, Springer, 2006, pp. 905–911.

[19] E. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith, Counterexample-Guided Abstraction Refinement for Symbolic Model Checking, J. ACM 50 (2003) 752–794.

[20] K. Laster, O. Grumberg, Modular model checking of software, in: B. Steffen (Ed.), Tools and Algorithms for the Construction and Analysis of Systems, Vol. 1384 of Lecture Notes in Computer Science, Springer, 1998, pp. 20–35.

[21] P. Godefroid, Compositional dynamic test generation, in: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '07, ACM, New York, NY, USA, 2007, pp. 47–54.