Test Generation for Sequential Nets of Abstract State Machines*

Paolo Arcaini[1], Francesco Bolis[2], and Angelo Gargantini[2]

[1] Dip. di Tecnologie dell'Informazione, Università degli Studi di Milano, Italy
paolo.arcaini@unimi.it
[2] Dip. di Ing. dell'Informazione e Metodi Matematici, Università di Bergamo, Italy
{francesco.bolis,angelo.gargantini}@unibg.it

Abstract. Test generation techniques based on model checking suffer from the state space explosion problem. However, for a family of systems that can be easily decomposed in sub-systems, we devise a technique to cope with this problem. To model such systems, we introduce the notion of *sequential net* of Abstract State Machines (ASMs), which represents a system constituted by a set of ASMs such that only one ASM is active at every time. Given a net of ASMs, we first generate a test suite for every ASM in the net, then we combine the tests in order to obtain a test suite for the entire system. We prove that, under some assumptions, the technique preserves coverage of the entire system. We test our approach on a benchmark and we report a web application example for which we are able to generate complete test suites.

1 Introduction

Model-based testing (MBT) aims to (re)use models and specifications for software testing. One of the main applications of MBT consists in test generation where tests are automatically generated from possibly partial and abstract models of the system under test. We here assume that MBT is performed in a typical black-box way: test suites are derived from models and not from source code.

Although MBT and test generation from models are rather mature topics in software testing and several approaches and tools exist [15], MBT for complex software systems is still an evolving field and its scalability is still questionable.

In a recent and still ongoing MBT project, we have tried to model web applications with Abstract State Machines (ASMs) and use a tool for test generation. Since the used technique is based on model checking [9], 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 in an exponential mode together with the size of the model. Several techniques exist to overcome this limitation, like symbolic representation of states, compact storing of states,

* The second author has been supported by the project Ricerca Applicata per il Territorio - Berg. II - Regione Lombardia and Alcatel-Lucent Spa

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 independent sub-systems that pass the control to each other such that only one sub-system is active at any time. In a web application, for instance, only one page is active at any time.

Such systems can be modeled 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.

In Sect. 4 we present a technique that is able to generate tests for a net of ASMs, reducing the state explosion. A test suite that covers every single machine is generated. These test suites are combined in order to obtain a test suite for the whole system. Under some assumptions, this technique preserves coverage of the entire system and reduces considerably the effort required to generate the whole test suite, as reported in the experiments using a benchmark example (in Sect. 5) and a simple web application (in Sect. 6).

2 Background

Software testing is a costly and time-consuming activity; specification-based (or model-based) testing [10] permits to considerably reduce the testing costs. In specification based testing, a specification describes the expected behavior of the system, and can be used as a test oracle to assess the correctness of the implementation. Moreover, specifications are also usually used to define test adequacy criteria, that determine if a test suite is adequate to test a software; various techniques exist to generate test sequences from formal specifications.

We assume that the reader is familiar with the ASMs [3]. 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 final state s_n is the state where the test goal is achieved.*

Definition 2. *A* test suite *(or* test set*) is a finite set of test sequences.*

Definition 3. *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.*

Several coverage criteria have been defined in [9] for ASMs. 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 a state in a test sequence in which r_i does not fire.

2.1 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 the model checkers to produce counterexamples [7]. The method consists of steps:

1. The test predicates set $\{tp_i\}$ 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. In this case, 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[3] and its ATGT tool [8], based on the model checker SPIN [11].

3 Sequential Nets of Abstract State Machines

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

A typical example is that of web applications. In a web application just one web page is active at any time, and the active page *decides* which is the next page to be displayed. 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 Description of the web application case study

We describe a web application case study taken from [12] we used in our experiments. There are six php pages in the web application under test and each of them, as well as their corresponding ASM, is described below.

– `index.php` – It serves as the login interface for the website. A user is required to enter a username and a password in order to access the other three pages of the site. The *Reset* button clears all text entries, while the *Submit* button opens up `main.php`, as long as the identification credentials are correct. If any information is missing, an error message page is displayed.

[3] `http://asmeta.sourceforge.net/`

– `error_b.php` – It is activated from `index.php` if any information is missing, or username or password are wrong.

– `main.php` – It permits users to execute different actions. Specifically, users can click on a link (at top left corner of page), upload a file by clicking on the *Browse* button, enter text into a textbox, select a checkbox, and click on a *Submit* button which loads `random.php`.

– `error_a.php` – It is displayed if any information is missing in `main.php`.

– `random.php` – It permits users to execute actions not available in `main.php`. Two links bring the user back to `index.php` and `main.php`. There are also drop-down lists, radio buttons, and a *Submit* button which loads `end.php`.

– `end.php` – It serves as the *end* of the web application. The user has the option of closing the web browser, or clicking on a link to return to `index.php`.

3.2 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 4. *A* sequential net of machines *is a set of Abstract State Machines* $M1, \ldots, Mn$ *such that:*

1. *each machine has only one initial state,*
2. *the machine* $M1$ *is the initial machine,*
3. *only one machine is active at any time,*
4. *the active machine decides when and to which machine the control is passed,*
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 do not run in parallel, pass the control to each other, and do not share information, although they 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 an unique sequence among the machines, since every machine can decide the next machine depending on local decisions. A sequential net is a graph, where each node is a machine and an arc is a transfer of control between two machines.

A possible way to model every single machine M_i of the net, so that it can signal the transfer of control, is the following:

1. add a domain $AsmDomain = \{M_1, \ldots, M_n\}$ to its signature;
2. add a 0-ary function *currAsm* of type *AsmDomain* to its signature; *currAsm*, in the initial state, must assume the value M_i;
3. write 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.

Every machine M_i can be independently executed. 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). For the sake of brevity, we do not specify the internal actions of the machines.

```
asm M1
signature:
  enum domain
    AsmDomain = {M1, M2, M3}
  monitored a: Integer
  controlled currAsm: AsmDomain
definitions:
  rule r_m1 =
    if a = 2 then
      currAsm := M2
    else if a = 5 then
      currAsm := M3
    else // do machine M1 actions
    endif endif

  main rule r_main1 =
    if currAsm = M1 then r_m1[]
    endif
default init s0:
    function currAsm = M1
```

Code 1. Machine M1.

```
asm M2
signature:
  enum domain
    AsmDomain = {M1, M2, M3}
  monitored b: Integer
  controlled currAsm: AsmDomain
definitions:
  rule r_m2 =
    if b = 2 or b = 30 then
      currAsm := M1
    else if b = 5 or b = 100 then
      currAsm := M3
    else // do machine M2 actions
    endif endif

  main rule r_main2 =
    if currAsm = M2 then r_m2[]
    endif
default init s0:
    function currAsm = M2
```

Code 2. Machine M2.

```
asm M3
signature:
  enum domain
    AsmDomain = {M1, M2, M3}
  monitored c: Integer
  controlled currAsm: AsmDomain
definitions:
  rule r_m3 =
    if c = 2 then
      currAsm := M2
    else if c = 5 then
      currAsm := M1
    else // do machine M3 actions
    endif endif

  main rule r_main3 =
    if currAsm = M3 then r_m3[]
    endif
default init s0:
    function currAsm = M3
```

Code 3. Machine M3.

$$\begin{tikzcd}
{} \arrow[r] & M1 \arrow[r, bend left=10, "a=2"] \arrow[rr, bend left=30, "a=5"] & M2 \arrow[l, bend left=10, "b=2\vee b=30"] \arrow[r, bend left=10, "b=5\vee b=100"] & M3 \arrow[l, bend left=10, "c=2"] \arrow[ll, bend left=30, "c=5"]
\end{tikzcd}$$

Fig. 1. Three ASMs constituting a sequential net.

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.

One possible simple way is to merge all the machines in an unique *product* ASM as follows:

- the signatures of the machines are merged in a single signature; there is just one copy of the *AsmDomain* domain and of the *currAsm* function in the product machine;
- all macro rules (except the main rules) of the single machines are included;
- in the main rule *r_main*, rules *r_m*i[] are individually called according to the value of the function *currAsm*;
- the initial states are merged; the function *currAsm* is initialized to the value *M1* (the first sub-system is active in the initial state).

Given the sequential net shown in Fig. 1, the product machine is the one shown in Code 4.

4 Test Generation for Sequential Nets of ASMs

In order to efficiently test a system modeled as a sequential net of ASMs, it is not enough to test the single sub-systems, since also the interaction among

```
asm ProductM
signature:
    enum domain AsmDomain = {M1, M2, M3}
    monitored a: Integer
    monitored b: Integer
    monitored c: Integer
    controlled currAsm: AsmDomain
definitions:
    rule r_m1 = if a = 2 then currAsm := M2
                else if a = 5 then currAsm := M3 else // do machine M1 actions
                endif endif
    rule r_m2 = ...
    rule r_m3 = ...
    main rule r_main = if currAsm = M1 then r_m1[]
                       else if currAsm = M2 then r_m2[] else r_m3[] endif endif
default init s0:
    function currAsm = M1
```

Code 4. Product machine of the sequential net in Fig. 1.

them must be tested. So, we must generate test sequences that cover the whole application and not just the single sub-systems.

The first idea is to derive the test sequences directly from the product machine that already contains all the interactions among sub-systems. 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 just on the single machines and not on the product machine; indeed, 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 an 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 the test suites for every machine

We use model checking as in [9] to generate a test suite for every ASM. Given the test sequences of a machine M_i, we define *inner* those sequences that terminate in a state in which *currAsm* is M_i, and *exiting* 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.

4.2 Building the test sequence graph

The generated test sequences constitute a graph, called *test sequence graph*, where every node is a machine and every arc is a test sequence. Test sequences that do not change the current machine are self loops of a node; test sequences that change the current machine, instead, are arcs between different nodes.

4.3 Combining the tests by visiting the test sequence graph

The algorithm used to visit the graph and build the *combined* test sequences is shown in Alg. 1.

Algorithm 1 Visiting the test sequence graph. Procedure *visitGraph*.

```
Require: the node $n$ to visit
Require: a test sequence prefix that permits to reach the node
 1: $visitedNodes \leftarrow visitedNodes \cup n$
 2: $testSet \leftarrow testSet \cup prefix$
 3: for $arc \in outArcs(n)$ do
 4:     $prefixToFn \leftarrow prefix + testSeq(arc)$
 5:     if $finalNode(arc) \notin visitedNodes$ then
 6:         $visitGraph(finalNode(arc), prefixToFn)$
 7:     else
 8:         $testSet \leftarrow testSet \cup prefixToFn$
 9:     end if
10: end for
```

The procedure executes a depth-first search of the graph. It takes as argument a node n to visit and a test sequence *prefix* that permits to reach n; n is marked as *visited* (line 1) in order to not be visited again and *prefix* is added to the test suite *testSet* we are building (line 2). Then, for each exiting arc of n

– the new prefix *prefixToFn* is built concatenating the current *prefix* with the test sequence that brings to the final node *fn* of the arc (line 4);
– if *fn* has not already been visited, *fn* is visited using as prefix *prefixToFn* (line 6); otherwise, *prefixToFn* is added to the test suite (line 8).

The procedure *visitGraph* is invoked using as argument the initial machine $M1$ of the net and the empty test sequence ϵ.

Note that the visit of the test sequence graph has linear complexity with the number of arcs and nodes and it requires a negligible amount of time with respect to the generation of the test suites.

It is straightforward to prove that the test sequences obtained with the presented algorithm are valid sequences for the product machine.

4.4 Coverage

We are interested in investigating the relationship between the coverage provided by a test suite obtained from the single machines and the coverage provided by using the product machine instead.

Definition 5. *A coverage criterion C is* preservable *if any test suite TS, obtained by the combination of tests suites TS_1, ..., TS_n that satisfy C over the single machines M_1, ..., M_n, satisfies C over the product machine.*

If a criterion is preservable, we can satisfy it on the product machine deriving the test sequences from the single machines and combining them later. The *rule coverage* criterion, for example, is *preservable* because of the following reasons:

1. by definition of *sequential net*, every machine is reachable starting from the initial machine; in each single machine M_i, every transition from M_i to another machine is specified with the update of the *currAsm* function;

2. if the *rule coverage* criterion is satisfied in every machine, it means that every rule is executed, including all the updates of the function *currAsm*. So, for each transition, there is a test sequence that contains it;
3. by construction, the *visitGraph* algorithm assures that, if a node of the test sequence graph is reachable, a test sequence that reaches that node is built;
4. in the main rule, the product machine describes the sequential net without adding or removing any transition: at each step it simply executes the rule of the machine specified by *currAsm*.

4.5 Limits of the approach

The major limit of the proposed approach is that not all criteria are preservable. A criterion, in order to be preservable, must satisfy a necessary (but not sufficient) condition: it must require that, for each machine M_i (with $i \neq 1$), there exists a test sequence of another machine that reaches M_i. The rule coverage criterion satisfies such condition, since it covers all the transitions to other machines. Let's see a criterion that, since it does not satisfy such condition, is not preservable:

C_{np}: A test suite satisfies the criterion C_{np} if every macro rule r_i is fired in at least one test sequence.

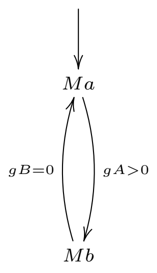
Let's see the test generation process using C_{np}. Let Ma and Mb be two ASMs, shown, respectively, in Code 5 and 6, that constitute a net. The product machine is shown in Code 7.

```
asm Ma
signature:
   enum domain AsmDomain = {Ma, Mb}
   monitored gA: Integer
   controlled currAsm: AsmDomain
definitions:
   rule r_mA = if gA > 0 then
                 currAsm := Mb endif
   main rule r_mainA =
      if currAsm = Ma then r_mA[] endif
default init s0:
   function currAsm = Ma
```

Code 5. Machine Ma.

```
asm Mb
signature:
   enum domain AsmDomain = {Ma, Mb}
   monitored gB: Integer
   controlled currAsm: AsmDomain
definitions:
   rule r_mB = if gB = 0 then
                 currAsm := Ma endif
   main rule r_mainB =
      if currAsm = Mb then r_mB[] endif
default init s0:
   function currAsm = Mb
```

Code 6. Machine Mb.



```
asm ProductMaMb
signature:
   enum domain AsmDomain = {Ma, Mb}
   monitored gA: Integer
   monitored gB: Integer
   controlled currAsm: AsmDomain
definitions:
   rule r_mA = if gA > 0 then currAsm := Mb endif
   rule r_mB = if gB = 0 then currAsm := Ma endif
   main rule r_main = if currAsm = Ma then r_mA[] else r_mB[] endif
default init s0:
   function currAsm = Ma
```

Code 7. Product machine of the machines Ma and Mb.

In the machine Ma, the criterion C_{np} is satisfied if there exists a test sequence in which the macro rule r_mA fires; C_{np} is satisfied, for example, by the test suite

$TS_A = \{ts_A\} = \{[(gA = 0, currAsm = Ma), (gA = 0, currAsm = Ma)]\}$. In the machine Mb, C_{np} can be satisfied if there exists a test sequence in which the macro rule r_mB fires; it is satisfied, for example, by the test suite $TS_B = \{ts_B\} = \{[(gB = 0, currAsm = Mb), (gB = 1, currAsm = Ma)]\}$[4]. The test sequence graph obtained from test suites TS_A and TS_B is shown in Fig. 2.

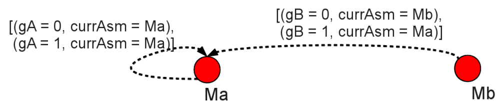


Fig. 2. Test sequence graph obtained with the criterion C_{np} over Ma and Mb.

The test suite obtained from the visit of the test sequence graph is $TS_{AB} = \{ts_A\} = \{[(gA = 0, gB = 345, currAsm = Ma), (gA = 0, gB = 7, currAsm = Ma)]\}$, where the values of gB are randomly chosen. In the product machine *ProductMaMb*, shown in Code 7, C_{np} is not satisfied using the test suite TS_{AB}, since macro rule r_mB never fires.

Nevertheless, it is possible to build a test suite that satisfies the criterion C_{np} in *ProductMaMb*, such as $TS_P = \{[(gA = 1, gB = 235, currAsm = Ma), (gA = 456, gB = 1, currAsm = Mb), (gA = 73, gB = 3, currAsm = Mb)]\}$.

Another limit of our approach is that the model checker may fail to find any test sequence that reaches one machine, although such sequence would be required by the (preservable) criterion. This may happen, for instance, because of the state explosion problem in a single machine. Of course, if this case occurs, it would be even more likely that the model checker would fail on the product machine as well.

The assumption that the machines do not share information limits the applicability of our technique. It can be applied only if the different sub-systems modeled by different ASMs either do not share any information or share information that does not influence the behavior of the machines. For instance, in the case study application of Sect. 3.1, all the pages share the username (which is shown in the web pages) and the session information, which, however, do not appear in the ASMs since they do not influence the behavior. If the web pages shared behavioral information, then our approach would not be applicable. We plan in the future to introduce in sequential nets of ASMs also a way for the machines to share information.

5 Initial Experiment

In order to evaluate our approach, we have experimented it with a small system. It resembles the combination lock finite state machine [13], for which generating a transition covering test suite becomes exponentially expensive. The problem is that of discovering the key of an electronic combination lock made of n digits

[4] Any not empty test suite (with any value for monitored functions gA and gB) satisfies the criterion over machines Ma and Mb because the execution of macro rules r_mA and r_mB does not depend on the evaluation of any guard.

$$\begin{tikzcd}
M_1 \arrow[r, "a_1=1"] & M_2 \arrow[r, "a_2=1"] \arrow[l, bend right=60, "a_2>x/2"'] & M_3 \arrow[r, "a_3=1"] \arrow[ll, bend left=60, "a_3>x/2"] & \cdots M_i \cdots \arrow[r, "a_{n-1}=1"] \arrow[lll, bend right=60, "a_i>x/2"'] & M_n \arrow[llll, bend left=60, "a_n>x/2"]
\end{tikzcd}$$

Fig. 3. The sequential net of ASMs for the combination lock problem.

having values from 1 to x. We have modeled the system as a sequential net of ASMs (see Fig. 3). The net is composed of n machines; every machine M_i has a monitored function a_i in the range $[1, x]$. If a_i (with $i = 1, \ldots, n-1$) takes the specific value 1 then the next machine M_{i+1} becomes active; if a_j (with $j = 2, \ldots, n$) becomes greater than $x/2$ then the system goes back to machine M_1, otherwise the machine M_j remains active.

We have evaluated our method depending on the number of digits (machines) n and/or the base x (the cardinality of the codomain of functions a_i).

For each combination of n and x we have built n single machines, where each machine has nx states since the signature of each machine M_i is composed of two 0-ary functions, a_i and $currAsm$, whose codomain sizes are, respectively, x and n. Then we have built the unique product machine that has nx^n states, since there are n 0-ary functions whose codomain size is x, and a 0-ary function whose codomain size is n.

Then we have generated the test sequences both for the product machine and for the sequential net of machines by the method introduced in this paper. As expected, we discovered that it is easier to execute n times the model checker over the single machines rather than executing the model checker one time over the product machine. The results of the experiment are shown in Fig. 4; the dependence between the execution time and the number of single machines n is reported. If the single machines are used, the execution time grows linearly with the number of machines; if the product machine is used, instead, the execution time grows exponentially with the number of machines. We made several experiments with different values for x (the cardinality of the codomain of functions

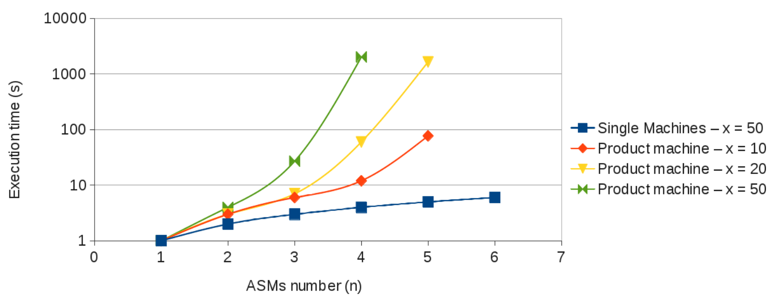


Fig. 4. Model checker executions times (sec.).

a_i); as expected, in the product machine the value of x influences the execution time (even for small changes of x), whereas in the single machines it is irrelevant. We report the experiments made with the product machine with x equal to 10, 20 and 50, and the experiment made with the single machines with x equal to 50. We set a time limit of 1 hour for each experiment setting. All the experiments were executed on a Linux PC with 8 Intel(R) Xeon(R) CPUs E5430 @ 2.66GHz and 8 GB of RAM.

Test suite sizes In Table 1 we report the sizes of the test suites obtained using the sequential net method and the product machine method. We report the sizes obtained with different number of machines; we do not report the value of x because it does not influence the test suite size.

# ASMs	1	2	3	4	5	6
Sequential net	3	5	7	9	11	13
Product machine	3	7	11	15	19	n/a

Table 1. Test suite size.

From our experiments it seems that the test suites derived from the test sequence graph are smaller than those obtained directly from the product machine. However, we must notice that this can not be taken as a general law; we plan to do additional experiments to define more clearly the relationship between the sizes of the test suites obtained with the two methods.

Code coverage As sanity check, we measured also the code coverage obtained by using the two methods. We implemented the system, previously specified in ASM, into Java and translated the test suites in JUnit. We obtained the same code (statement and branch) coverage by using both the test sequences generated from the product ASM and from the sequential net.

6 Model-based Testing of Web-based Applications

We have studied the test generation for sequential nets of ASMs in the context of MBT of web-based applications [6]. In this context, every machine represents a single page of the application. The main purpose is to automatically generate test cases for web applications using a model-based approach. This is accomplished by first creating an ASM for each web page of the web application; in this scenario the *AsmDomain* can be interpreted as the set of web pages and the *currAsm* function as the current active page. The methodology introduced in Sect. 4 is applied to obtain a test suite for the whole web application; finally, each test sequence can be mapped to a SAHI script [1] to exercise the tests directly on the web application. In the following we report the experiment made with the case study described in Sect. 3.1.

Modeling every page with an ASM The first idea was modeling the complete web application with a single ASM. The model construction was feasible but the model checking was not able to complete the test generation. So, we modeled the web application using a sequential net of ASMs where every page is represented by an ASM and the domain *AsmDomain* is composed by the web pages. The obtained sequential net is shown in Fig. 5.

$$\begin{tikzcd}
\texttt{error\_b} \arrow[r, shift left] & \texttt{index} \arrow[l, shift left] \arrow[r, shift left] & \texttt{main} \arrow[l, shift left] \arrow[r, shift left] \arrow[rrr, bend left] & \texttt{random} \arrow[l, shift left] \arrow[r] \arrow[ll, bend left] & \texttt{end} \arrow[lll, bend right] & \texttt{error\_a} \arrow[lll, bend left]
\end{tikzcd}$$

Fig. 5. Web-based application case study - Sequential net.

For translating a web page behavior into an ASM, we have put on a table the inputs of the web page (e.g., the values of the text fields) and identified, for every combination of inputs, a transition to another page or a set of state updates. In this way we have built an ASM for each web page.

Test generation For the test generation we have used, as described in Sect. 4.1, the ATGT tool over each ASM, using as coverage criteria all those described in [9].

Test sequence graph construction Then we have built the test sequence graph (see Fig. 6) as described in Sect. 4.2. Each transition of the sequential net has been covered in the test sequence graph.

Table 2 reports, for each ASM, the number of test sequences, divided between *inner* and *exiting*.

	`index`	`error_b`	`main`	`error_a`	`random`	`end`
# tests	24	3	36	3	45	2
# inner - # exiting	18 - 6	1 - 2	26 - 10	1 - 2	32 -13	1 - 1

Table 2. Test sequences number.

Test sequence combination Then, we have applied the technique presented in Sect. 4.3 in order to obtain a single test suite for the whole web application. The obtained test suite contains 212 test sequences and it satisfies all the coverage criteria used to generate the test suites over the single machines.

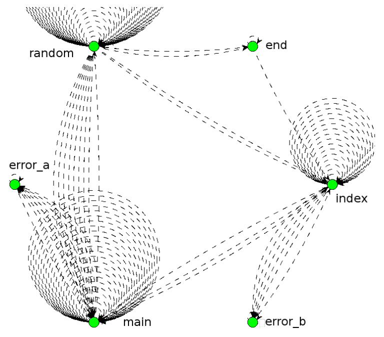


Fig. 6. Web-based application case study - Test sequence graph.

Test of the web application Finally, each test sequence of the test suite has been automatically mapped to a SAHI script; the execution of all the scripts has permitted us to test all the aspects of the web application. Code 8 shows one of the produced SAHI scripts.

```
_navigateTo("index.php");
_setValue(_textbox("username"),"admin");
_setValue(_textbox("password"),"pwd");
_click(_submit("submit"));
_click(_checkbox("agree"));
_setValue(_textarea("text"),"someText");
```

Code 8. SAHI script example.

7 Related Work

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 [5] 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 [14] 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 [5], 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 [4] 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.

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 *currentAsm* function, which is used in the test goals, may be influenced by all the functions.

In [2] a web application is modeled by means of FSMs. They also face the state explosion problem; they try to overcome it by partitioning a web application into clusters that can contain web pages and other clusters. For each cluster an FSM is built; an *Application FSM* represents the entire application. Test sequences are derived from single FSMs. They share with us the need of decomposing the model into smaller models in order to keep the state space size tractable. As we do, they provide a technique for combining test sequences obtained from the single FSMs into a test suite to be used for testing the whole web application. The technique they propose also permits to propagate inputs among FSMs, while, in our approach, we currently do not allow the ASMs to exchange any information.

8 Future work and Conclusion

We have tried to address the state explosion problem in test generation by model checking. For sequential nets of ASMs, our approach makes the test generation more scalable, without reducing the coverage obtained by the tests. Initial experiments show that our approach provides excellent benefits. We plan to extend the model of ASM nets by considering cases in which a single machine has several initial states and the machines share some locations. In such case, we believe that the test generation can not be done in advance for all the machines, but the construction and the visit of the graph must be done together.

We assume that the designer keeps the models separated from the beginning; as future work, we plan to study a 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 sub-models, even those techniques which do not suffer so much from the size of the model under test.

References

1. Sahi website. `http://sahi.co.in/`.
2. A. A. Andrews, J. Offutt, and R. T. Alexander. Testing Web applications by modeling with FSMs. *Software and Systems Modeling*, 4:326–345, 2005.
3. E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer Verlag, 2003.
4. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50:752–794, 2003.
5. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.
6. G. A. Di Lucca and A. R. Fasolino. Testing Web-based applications: The state of the art and future trends. *Inf. Softw. Technol.*, 48:1172–1186, 2006.
7. G. Fraser and A. Gargantini. An evaluation of model checkers for specification based test case generation. In *ICST 2009, 1-4 April 2009, Denver, Colorado, USA*, pages 41–50. IEEE Computer Society, 2009.
8. A. Gargantini and E. Riccobene. ASM-Based Testing: Coverage Criteria and Automatic Test Sequence Generation. *J.UCS*, 7:262–265, 2001.
9. A. Gargantini, E. Riccobene, and S. Rinzivillo. Using Spin to Generate Tests from ASM Specifications. In *ASM 2003 - Taormina, Italy, March 2003. Proceedings, LNCS 2589*, 2003.
10. R. Hierons and J. Derrick. Editorial: special issue on specification-based testing. *Software Testing, Verification and Reliability*, 10(4):201–202, 2000.
11. G. Holzmann. *Spin model checker, the: primer and reference manual*. Addison-Wesley Professional, first edition, 2003.
12. A. M. Memon and O. Akinmade. Automated Model-Based Testing of Web Applications. In *Google Test Automation Conference 2008*, 2008.
13. E. F. Moore. Gedanken experiments on sequential machines. In *Automata Studies*, pages 129–153, Princeton, 1956.
14. S. Park and G. Kwon. Avoidance of State Explosion Using Dependency Analysis in Model Checking Control Flow Model. In *ICCSA 2006*, volume 3984 of *LNCS*, pages 905–911. Springer Berlin / Heidelberg, 2006.
15. M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan-Kaufmann, 2006.