

TESTOR: Deriving Test Sequences from Model-based Specifications

Patrizio Pelliccione², Henry Muccini², Antonio Bucchiarone¹, and Fabrizio Facchini²

¹ Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo" (ISTI-CNR)
Area della Ricerca CNR di Pisa, 56100 Pisa, Italy

`antonio.bucchiarone@isti.cnr.it`

² University of L'Aquila, Computer Science Department

Via Vetoio 1, 67010 L'Aquila, Italy

`{muccini,pellicci}@di.univaq.it`

Abstract. The dependability analysis of a component-based system may be driven by the components/system implementation or by the model-based specification provided prior to or together with the implementation. In particular, model-based specifications of a component-based system allows to explicitly model the structure and behavior of components and their integration, while model-based testing allows to derive test sequences which can be successively refined into test cases and then run onto the system implementation.

Several techniques have been proposed so far to allow model-based testing. However, very few of them show certain characteristics which are peculiar for use in industrial contexts. We here describe TESTOR, a TEST Sequence generaTOR algorithm which allows to extract test sequences from both state machine and scenario diagrams. We detail the algorithm, we apply it to a system study and we provide a link to its implementation.

1 Introduction

Roughly speaking, a component-based software system is an assembly of reusable components, designed to meet the quality attributes identified during the architecting phase [10]. Components are specified, designed and implemented with the intention to be reused, and are assembled in various contexts in order to produce a multitude of software systems.

The dependability of a component-based system strongly depends on both the quality of the assembled components, and on the quality of the assembly and its subsumed architecture. While the quality of a single component may be analyzed in isolation, the quality of the assembly may be verified only after components integration.

While in the past the verification stage to be properly performed required the assembly of already developed components, with the advent of the model-driven development, the models themselves may be analyzed before components

are developed or bought. The essence of model-driven development is the (very old) idea that a system is incrementally obtained by instantiating and refining a specification of system structure and behavior. The essence of model-driven analysis, instead, is the idea of using such models as representative of the (expected) system structure and behavior in order to predict the achievement of certain qualities, and later on to validate the system implementation.

In this paper, we focus our attention on *model-based testing* of component-based systems, at the integration level. We assume a structural and behavioral specification of the components is available, together with architectural information which allows to specify how such components are supposed to interact when assembled in a component-based system. Goal of our algorithm is to extract test sequences, which can be used to test the conformance of an implemented assembly with respect to the specification, considered as an oracle.

Many different approaches and tools have been proposed so far to extract test cases from system models. (Relevant model-based testing approaches are outlined in Section 2). However, very few of them may be considered applicable in “industrial contexts”. Following [7], in fact, a model-based testing approach to be suitable for industrial needs has to comply to some extra requirements and constraints: first of all, we cannot assume that a formal (complete and consistent) modelisation of the software system exists. What we may reasonably assume, instead, is a semi-formal specification in the form of UML diagrams. Moreover, the approach should be *usable* (additional formalisms should not be required), *timeliness* (even incomplete models should allow to start outlining a test plan), and *tool supported* (automated tool support is fundamental for strongly reducing testing costs). Surprisingly, even if many tools have been proposed so far for model-based testing [14, 2, 19], very few of them satisfy such requirements.

We here propose our algorithm, called TEst Sequence generaTOR (TESTOR), which allows to extract test sequences from model-based specifications as produced by practitioners. A behavioral model of each component is provided in the form of UML state diagrams. Such models explicitly specify which components interact and how. Sequence diagrams are used as an abstract or even incomplete (“sketchy”) specification of what the test should include. TESTOR takes in input UML state and sequence diagrams and synthesizes more detailed sequence diagrams (conforming to the abstract scenarios) by recovering missing information from state diagrams. The output represents test sequences (which include a sequence of stimuli to the system under test, and the expected responses to those stimuli [14]) specified in the form of more informative scenarios. Differently from other approaches, the test sequence generation process in TESTOR does not compute the parallel composition of the state machine models, it does not require any additional formalism, and it is completely tool supported.

The following of this paper is organized such that in Section 2 we analyze some relevant related work. Section 3 is utilized to describe the TESTOR algorithm. Section 4 describes how the TESTOR has been implemented. Section 5 provides initial evaluations of the proposed algorithm while Section 6 concludes the paper.

2 Related Work

In recent years the use of explicit models in software development (most notably the use of UML for object oriented analysis and design) has expanded greatly. Several researchers have focused their efforts in finding methods and tools for guiding the testing activities by means of system model description [2]. In this section we briefly summarize some of the recent research results which make use of UML-diagrams for testing purposes.

Many model-based testing approaches are *state-machine based*. Initial attempts to use UML state diagrams for testing purposes are presented by Offutt and Abdurazik [21], who translate these diagrams into formal SRC specifications, Liuying and Zhichang [18], who use a formal semantic of state machines to derive the test cases, and Kim et al. [17], who focus on class testing. More recently, Hartmann et al. [15] extended to a component-based paradigm the approach of [21], and Antoniol et al. [5], considered the derivation of test sequences from UML statecharts by covering selected paths in a FSM.

Some relevant proposals for *scenario-based testing* include: the approach of Graubmann and Rudolph [12], in which the Message Sequence Chart (MSC) inline expressions and High Level MSC (hMSC) are included into Sequence Diagrams for the specification of test cases; the methodology of Harel and Marely [13], which is specifically designed for scenario-based specification of reactive systems; TOTEM (Testing Object-orientED systems with the unified Modelling language) [8], which uses sequence or collaboration diagrams associated to each use case for deriving test cases, test oracles and test drivers, and SCENTOR [25], which uses JUnit as a basis for test case derivation. SeDiTeC [11] automatically generates test stubs for the classes and methods whose behaviour is specified in the sequence diagrams. The Cow Suite tools and methodology recently discussed in [6] provide an integrated and practical approach for generating and planning UML-based test suites for industrial contexts.

There are very few approaches, like our, where *both state machines and scenarios* are required for test case generation. UMLAUT (Unified Modelling Language All pUrposes Transformer)[3] is an approach tool supported which translates UML diagrams into an intermediate formal description understandable by the Test Generation and Verification (TGV) tool. AGEDIS (Automated Generation and Execution of Test Suites for DIstributed Component-based Software)[1], generates and executes test cases for application modelled according to the AML (AGEDIS Modelling Language), which is a specialised UML profile. SCENT (SCENario-based validation and Test of software) [24], creates scenarios in a structured way, formalizing them into statecharts.

Our work shares with all of them the idea of guiding the testing activities by means of system models. In particular, we share with UMLAUT, TGV, AGEDIS and SCENT the combined use of state and sequence diagrams. Differently from other approaches which use UML diagrams to model and test use cases or classes, we put ourselves in a component-based context, where state diagrams are used to model how components are supposed to behave in a component-based system.

3 The Test Sequence Generator Algorithm

According to [14], a model based test generator accepts as main inputs a *model of the software under test*, and a set of *test generation directives* which guide the test cases selection (called test purposes in [16]), and outputs a *test specification*, which includes a set of stimuli the tester should introduce in the system together with expected responses.

The model based test sequence generator TESTOR gets in input the behavioral model of the components (in terms of components' state machines) and a sequence diagram (inSD) representing the test directive, and outputs a set of sequence diagrams (outSD) representing the shortest paths which cover the inSD into the state machines. In other terms, each outSD contains the sequence of messages expressed by the inSD, enhanced/completed with information gathered by the components' state machines. It is important to notice that the TESTOR algorithm identifies the different test sequences outSD by individually simulating the different state machines, and without constructing the global automaton, i.e.: without parallel composition of the different behaviors. This important optimization allows to handle the well known state-explosion problem.

In order to help in explaining how TESTOR works, we go through an explanatory example. Figure 1.a – d illustrates the state machines related to the Siemens C.N.X. Optical Session System (OSS) conceptual architecture. Each state machine denotes a component and the labels on the transitions identify exchanged messages, with the assumption that a message can be exchanged only by a pair of components. For each transition an ? or ! operator is defined with the following meaning: ?[!]*l* identifies an input [output] of the *l* message.

The OSS description and its architecture have been presented in [20], and its discussion is out of scope for this paper. (The state machines' labels have been renamed to improve readability). The sequence diagram used as test directive is illustrated in Figure 1.e.

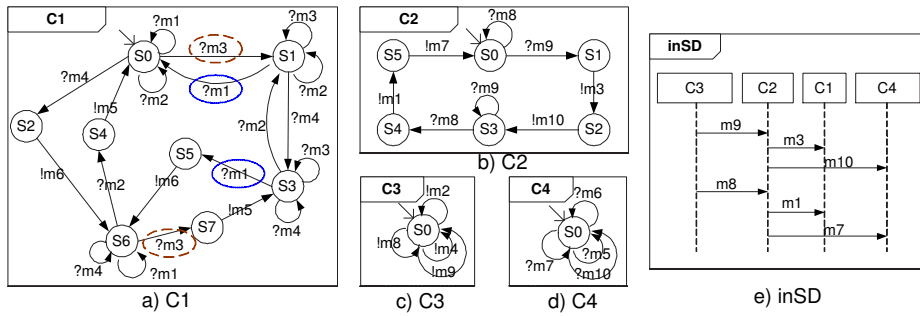


Fig. 1. a – d) SA Components Behavior, e) inSD

TESTOR, focussing on the first (not visited) message *m* in the inSD, and looking inside each state machine, searches a trace which allows to reach *m*,

starting from the current state of the state machine. When such trace is found, TESTOR recursively moves to the next (not visited) message m' in inSD, and checks a trace which allows to reach m' starting from the current state. At the end of this process, TESTOR tries to merge together the different traces in a unique trace (the outSD) which moves from the initial state and covers any message m in the inSD.

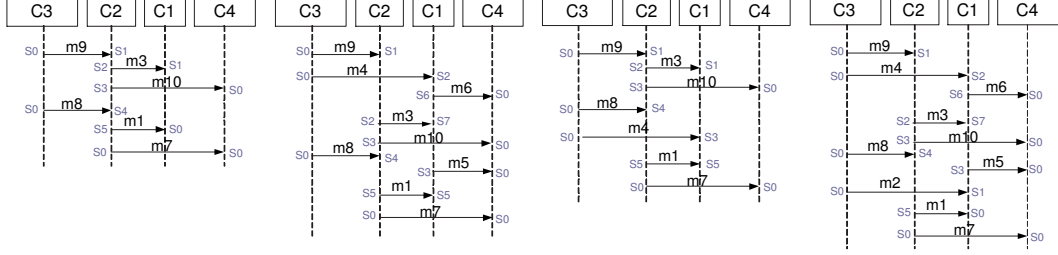


Fig. 2. The four traces generated

Figure 2 shows the four outSD traces generated by applying TESTOR to the case study in Figure 1. To increase readability, we report on the outSDs lifeline of each component, the state reached by the component, after having exchanged the message. Following the TESTOR algorithm, $m9$ is initially selected in the inSD in Figure 1.e. Starting from the initial state of the four state machines, the components C2 and C3 may exchange $m9$. C2 changes its state to $s1$. Then, $m3$ is selected in inSD. C1 and C2 may reach $m3$ from their current states ($s0$ and $s1$, respectively). Whenever the last message in inSD has been analyzed, a different algorithm, **Merge**, merges together the different traces, by checking if and when the different components may synchronize.

The TESTOR algorithm can be split into two macro-steps:

1. *State machines -sm- Linearization*: instead of traversing sm for identifying a trace which allows to reach m in sm , starting from sm current state s , we initially decompose sm in a set of linear *traces*. In order to limit the number of linearized traces, TESTOR avoids loops and repetitions such as each trace is branching free and each pair of traces do not share transitions.
2. *Test Sequence Generation*: the second macro-step involves the test sequence generation. This macro-step looks at each linearized state machine in order to identify the *sup-traces* of the inSD. A sup-trace is *a trace which contains any message contained in the inSD, in the same order they were in inSD, and includes any other message necessary to simulate the inSD trace into the state machines*. Sup-traces generated by TESTOR are minimal, since loops are removed. This algorithm is composed by a *Validation* part, which checks when and how sup-traces need to be combined to produce the outSD.

In the following of this section we detail the TESTOR algorithm. In Section 3.1 and 3.2 we detail the Linearization and Test Sequence Generation algorithms

used by TESTOR. For the sake of brevity this paper omits algorithm's details which can be found in [23].

TESTOR makes use of the following methods:

- **MessageSet** `messages(SD sd)`: is a method that returns the set of messages contained into the sequence diagram `sd`.
- **boolean** `isEmpty(Set s)`: is a boolean method that returns true iff the set `s` is empty.
- `delete(Set s, Element el)`: this method deletes the element `el` from the set `s`.
- `add(Set s, Element el)`: this method adds the element `el` to the set `s`.
- `updateIs(SD sd)` is a method that updates the initial state for each state machine.

For each state machine (`sm`), the Linearization algorithm is invoked (lines 2-5). We suppose that each `sm` is in its own initial state.

TESTOR Pseudo-Code

```

1 Testor(){
   SDSet sdSet;
3   foreach sm ∈ SM{
       Linearize(sm);
5   }

```

In lines 6-24 the TESTOR algorithm attempts to *construct the sup-traces* for each message `m`, and to *concatenate* them in order to obtain the wanted outSD. For each message `m` in the inSD, the method `TestSequenceGeneration` is called. It returns a set of traces, `sdSetMsg`, containing the message `m` and additionally any other message required to properly exchange `m`. Each message causes the construction of a set of traces. In order to produce a unique sup-trace for inSD, the sup-traces of any message in inSD need to be concatenated. Note that the `updateIs` method invocation precedes the `TestSequenceGeneration` method invocation because for each trace we have to set a different initial state. The errors raised in lines 9 and 17 identify the impossibility to reproduce the inSD inside the set SM of state machines. When such errors are raised, an inconsistency between inSD and the state machines is identified. The code in lines 19-22 constructs the set of outSD by appending the obtained traces for the current message `m` to the already stored (partial) outSDs i.e.: the traces built for messages that precede `m` in inSD.

```

6   sdSet = TestSequenceGeneration(m, ∅, ∅);
   if (isEmpty(sdSet)){
8     /* It is not possible to obtain the sequence */
       return Error;
10  }
   foreach m ∈ messages(sd){
12     foreach sd ∈ sdSet{
       updateIs(sd);
14     SDSet sdSetMsg = TestSequenceGeneration(m, ∅, ∅);
       if (isEmpty(sdSetMsg)){
16       /* It is not possible to obtain the sequence */
         return Error;
18     }
       delete(sdSet, sd);
20     foreach sdMsg ∈ sdSetMsg{
       add(sdSet, sd.sdMsg);
22     }
   }
24 }

```

3.1 Linearization

Starting from the initial state in the components' state diagrams, the Linearization algorithm follows each transition exiting from that state until the start state, or a state with output degree $outDegree$ or input degree $inDegree$ greater than one is reached. Roughly speaking, the linearization process creates a trace at any time a state with a branch is reached (i.e., a state with $outDegree > 1$ or $inDegree > 1$). The algorithm is iterated, starting from the previously reached state, until unvisited states still exist.

The number of linear traces generated can be computed as in the following:

$$\#traces = \underline{outDegree}(startState) + \sum \underline{outDegree}(s)$$

for each state s so that $(\underline{outDegree}(s) > 1) \vee (\underline{inDegree}(s) > 1)$

The linearization properties are the following:

1. it partitions states into *intermediate* states (states that are in the middle of a trace) and *exterior* states (states that are the source or the target of a trace).
 - *exterior* states are *i*) all the states with $(outDegree > 1) \vee (inDegree > 1)$, or *ii*) the start state, or *iii*) each sink state; *intermediates* are the other states.
2. the linearization does not loose information i.e.: it is possible to reconstruct the original state machines starting from the linearized traces.

Linearization applied to the Optical Session System: in the following we show the results of the linearization algorithm applied on the case study in Figure 1. As an example we explain how the traces for component $C2$ are obtained. The start state of $C2$ is S_0 (as shown in Figure 1.b). Since $outDegree(S_0)=2$, S_0 is an *exterior* state. The trace 1 is created thanks to the message $m8$. The trace 2 follows the path indicated by $m9$ and ends with the state S_3 that is a state with $outDegree$ equal to two. While trace 3 is generated by the loop message $m9$, trace 4 is the path that allows to reach state S_0 .

C1				
1. $S_0 \xrightarrow{m_1} S_0$	9. $S_1 \xrightarrow{m_4} S_3$			
2. $S_0 \xrightarrow{m_2} S_0$	10. $S_3 \xrightarrow{m_3} S_3$	C2		
3. $S_0 \xrightarrow{m_3} S_1$	11. $S_6 \xrightarrow{m_4} S_6$	1. $S_0 \xrightarrow{m_8} S_0$	C3	
4. $S_1 \xrightarrow{m_2} S_1$	12. $S_6 \xrightarrow{m_2} S_4 \xrightarrow{m_5} S_0$	2. $S_0 \xrightarrow{m_9} S_1 \xrightarrow{m_3} S_2 \xrightarrow{m_{10}} S_3$	1. $S_0 \xrightarrow{m_9} S_0$	C4
5. $S_0 \xrightarrow{m_4} S_2 \xrightarrow{m_6} S_6$	13. $S_6 \xrightarrow{m_3} S_7 \xrightarrow{m_5} S_3$	3. $S_3 \xrightarrow{m_9} S_3$	2. $S_0 \xrightarrow{m_8} S_0$	1. $S_0 \xrightarrow{m_7} S_0$
6. $S_6 \xrightarrow{m_1} S_6$	14. $S_3 \xrightarrow{m_4} S_3$	4. $S_3 \xrightarrow{m_8} S_4 \xrightarrow{m_1} S_5 \xrightarrow{m_7} S_0$	3. $S_0 \xrightarrow{m_4} S_0$	2. $S_0 \xrightarrow{m_{10}} S_0$
7. $S_1 \xrightarrow{m_3} S_1$	15. $S_3 \xrightarrow{m_1} S_5 \xrightarrow{m_6} S_6$		4. $S_0 \xrightarrow{m_2} S_0$	3. $S_0 \xrightarrow{m_5} S_0$
8. $S_1 \xrightarrow{m_1} S_0$	16. $S_3 \xrightarrow{m_2} S_1$			4. $S_0 \xrightarrow{m_6} S_0$

3.2 Test Sequence Generation

The input of the `TestSequenceGeneration` is a message m and two sets which identify the already explored state machines and related traces, while the output is a set of traces representing the different paths that conduct the system to exchange message m . The second and the third inputs are used for recursive purposes, propagating the state reached for each state machine and the traces under construction. In the first invocation the two sets are empty.

Let C_s (C_r) the component which sends (receives) m . The algorithm initially generates two sets of scenarios from the linearized traces of C_s and C_r , and eventually merges them (through the `merge` method) generating the outSD scenario.

`TestSequenceGeneration` makes use of the linearized state machines and can be synthesized as follow:

1. Let is the state of C when m must be exchanged.

The `TestSequenceGeneration` method selects the linearized traces containing m . For each selected trace t we check if the trace contains the state is . If it contains it, the trace t is accepted. Otherwise we must check if concatenating several traces it is possible to reach the state is . The selection of those traces uses some rules: *i*) if a trace contains twice the message m , the trace is rejected. *ii*) if a trace adds a loop (an already visited state is reached), then it is rejected.

This procedure is implemented by the `Validation` method and identifies sup-traces which may contain new messages that are required by the system to reach the configuration able to exchange the message m .

2. Any added message to the sup-trace can require the introduction of other messages. In fact, for each new message contained into the sup-trace, the `TestSequenceGeneration` algorithm is invoked with the set of the updated state machine and the already constructed traces as parameters.

For a complete description of the algorithm please refer to Appendix A. In the following, the `Validation` and the `Merge` algorithms are detailed.

Validation : the input of this algorithm are a trace t , a message m , the current state cs for trace t , the initial state is for component C (where component C contains trace t), and a set of traces tr used to avoid infinite loops. The output of the algorithm is a set of sup-traces ST , of a valid trace.

It performs also validity checks in the sense that returns an empty set if it is impossible to reach the initial state is .

In the algorithm we make use of the following methods:

- `append(Trace tr1, Trace tr2)`: is a method that assigns $tr2.tr1$ to $tr1$.
- `Trace subtrace(t, s1, s2)`: this method returns the sub-trace of t that starts with s_1 and ends with s_2 ;
- `tracesSet tracesToValidate(s)`: this method returns the set of the traces with target state s that must be verified;

- boolean `contains(t,m)`: is a boolean method that returns *true* iff the trace t contains the transition m .
- State `initialState(Trace t)`: this method returns the initial state of the trace t .
- State `finalState(Trace t)`: this method returns the final state of the trace t .

The code for function `Validation` can be divided into three parts: *i*) the current state cs coincides with the initial state is , lines 5-9, *ii*) we reach the start state of trace t , lines 10-14, *iii*) we iterate on the other traces identified thanks to the function `tracesToValidate`, lines 19-29. In the recursive construction of the sup-trace, traces containing the considered message m , loop generated through nested invocations of validation, and traces with loops on one state are not considered (lines 19-27).

Validation Pseudo-Code

```

ST Validation(Trace t, Transition m, State cs, State is, StateSet sSet){
2  ST Ist = ∅;
   Traces tdv;
4  /* Basic step */
   if (cs == is){
6     st = subtrace(t,is,cs);
     add(Ist ,st);
     return Ist;
8  }
   if (initialState(t)==is){
10    st = subtrace(t,is,cs);
     add(Ist ,st);
     return Ist;
12  }
   add(sSet,cs);
14  /* If the start state of the trace is not equals to is */
   tdv = tracesToValidate(cs);
16  foreach t' in tdv{
18     if (!contains(t',m)&&
20         !(initialState(t') ∈ sSet)&&
           !(initialState(t')==finalState(t'))){
22         ST Istc = Validation(t',m,cs,is,sSet);
         foreach st ∈ Istc{
24             append(st,subtrace(t,is,cs));
             add(Ist ,st);
26         }
28     }
   }
   return Ist;
30 }

```

Merge : This method gets in input two sets of sequence diagrams and tries to merge them in order to obtain a single sequence diagram. Each sequence diagram contains the set of messages that are contained into the inSD but each one of them can be enriched with messages required by some components in order to respect the inSD. Then the **Merge** method tries to merge each sequence diagram into the first set with every sequence diagram contained into the second set.

For every message contained into the inSD the following reasoning is iterated: let m the current message and m' its predecessor. We are sure that m and m' (if m' exists) are contained into the pair of sequence diagram. We remember that each one of the sequence is complete, in the sense that no additional messages

are required to allow the system to exchange the message m . Then let $setS$ the set of messages contained into the first set, between the message m and m' , and $setR$ the set of messages contained into the second set, between the message m and m' . The algorithm checks if there exists an ordering of messages that allows the system to exchange all the messages contained into $setS$ and $setR$ in according to the state machines that dictate the behavior of the system.

Test Sequence Generation applied to the Optical Session System : in the following, we show how the TESTOR algorithm works by making use of the case study. We take into consideration only few steps in the scenario in Figure 1. As showed in Figure 1.e, the first message in inSD is $m9$. The traces (in Section 3.1) which make use of $m9$ are those related to components C3 and C2 where the sender component is C3 (with associated state machine sm_s) and the receiver component is C2 (with associated state machine sm_r).

Analyzing the traces generated for the sender and the receiver, the traces of interest are $itc_s = \{1\}$ and $itc_r = \{2, 3\}$. Trace 3 in itc_r is not of interest since it does not allow to reach the start state without loops. So far, the obtained $sdSet$ contains only one sequence diagram: by exchanging message $m9$, component C3 reaches state $S0$ and component C2 reaches state $S1$.

The second message in the inSD in Figure 1.e is $m3$. $sm_s = C2$ and $sm_r = C1$. $itc_s = \{2\}$ and $itc_r = \{3, 7, 10, 13\}$. The traces that must be considered are: trace 2 for component C2 and traces 3 and 13 for component C1. Trace 3 is selected since allows to reach state $S0$. Trace 7 is not selected for the following reasons: the initial state of trace 7 is $S1$; the traces analyzed to reach state $S0$ are traces 3, 4, and 16, where traces 3 and 4 are deleted because represent a loop (trace 3 contains also the message $m3$) and trace 16 requires to consider traces 9, 10, 13, and 14 which represent a loop or contain message $m3$. Trace 13 is selected since it can be combined with trace 5 to reach state $S0$ obtaining the path $S_0 \xrightarrow{m4} S_2 \xrightarrow{m6} S_6 \xrightarrow{m3} S_7$.

Traces 3 and 13 then cause the construction of two different sequence diagrams. The first scenario simply requires the exchange of message $m3$ that conducts component C2 to state $S2$ and component C1 to state $S1$. The second scenario contains, before the message $m3$, the messages $m4$ and $m6$ as required by the sup-trace obtained concatenating traces 13 and 5.

The states reached are the following: $C1_{is} = S1$, $C2_{is} = S2$, $C3_{is} = S0$, and $C4_{is} = S0$ for the sequence diagram 1 and $C1_{is} = S7$, $C2_{is} = S2$, $C3_{is} = S0$, and $C4_{is} = S0$ for the sequence diagram 2.

The same reasoning is applied to the remaining messages. Only another message, $m1$, causes the generation of two different sequence diagrams. Thus, the obtained sequence diagrams are four, as illustrated in Figure 2.

4 Tool Support

The TESTOR algorithm has been implemented has a plugin component for CHARMY, a validation framework for architectural analysis. A beta version of the TESTOR plugin implementation is currently available in [4].

While the CHARMY (standard) editing capabilities allows to edit the SA topology, the components' behavior and the inSD, the TESTOR plugin takes in input such information and produces the outSD scenarios.

Figure 3 shows a couple of screenshots of CHARMY and its TESTOR plugin. In Figure 3.a) the *sequence editor* pane is shown which allows to draw inSDs and to show resulting outSDs, while Figure 3.b) graphically depicts the *thread editor* which allows to specify the behavior of each component.

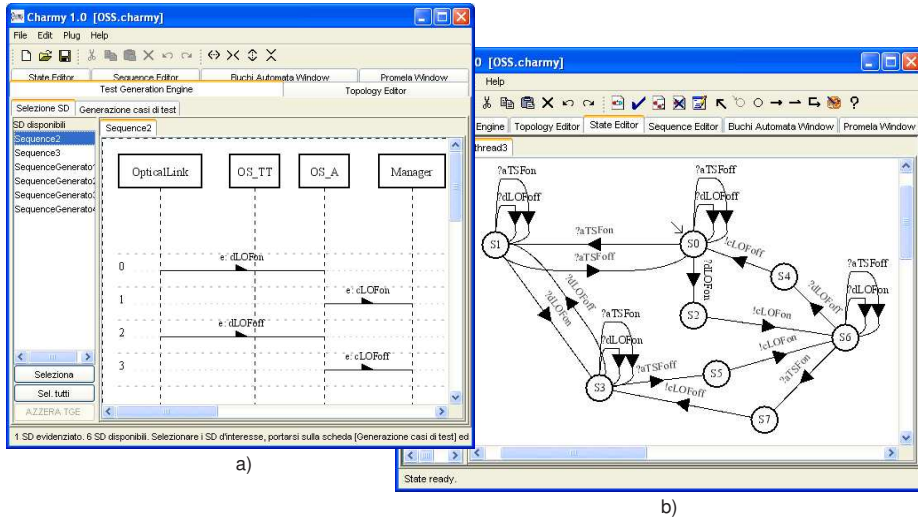


Fig. 3. Some screenshots of TESTOR and CHARMY

More details on CHARMY, the TESTOR plugin and the CHARMY model of the case study may be found in [4].

5 Some initial Considerations

As presented in Section 2, there are up to now many model-based testing approaches which make use of UML as the input notation. However, we may distinguish TESTOR from such approaches from many reasons: *i)* TESTOR does not require the parallel composition of component state machines. While other techniques usually require to create a global model representing the assembled system, TESTOR analyzes the components behavior in isolation. The state machine linearization process allows to synthesize traces which are successively traversed in order to reproduce the output scenarios. Since only a subset of the generated traces are traversed by an inSD, this solution allows to strongly reduce the state explosion problem; *ii)* TESTOR does not require any formal notation. State machine and scenarios are the only input required by the approach and the

tool; *iii*) TESTOR is conceived to meet some important (and often conflicting) requirements imposed by industry: low testing effort and high testing accuracy. Effort reduction imposes that models are used as they are, i.e., without requiring extra information and imposing model completeness. Testing accuracy, on the other side, requires to generate test cases from models that are as much informative as possible. The use of both state machine and scenario specifications allows to perceive this goal; *iv*) TESTOR is completely tool supported.

Here we report some initial considerations about the proposed algorithm, in terms of *completeness*, *correctness*, and *complexity*. Future work will improve such informal notes.

Regarding completeness, TESTOR does not generate all the possible traces out of the SA behavioral model. In fact, since the `TestSequenceGeneration` algorithm creates loops-free traces, completeness cannot be guaranteed (neither it would be). However, the TESTOR algorithm guarantees to cover at least once any occurrence of inSD messages.

Talking about correctness, instead, we may empirically prove that the TESTOR is correct (i.e.: the traces generated from the algorithm are real traces in the state machine models). The linearization algorithm, for definition, contains traces which are contained into the state machines. The `TestSequenceGeneration` algorithm initially selects such linearized traces which allow to reach m , and then combine such traces based on how state machines synchronize. The algorithm `merge` assures, finally, that the traces produced are behaviors contained into the system.

Another important point to be evaluated regards the algorithm computational complexity, in terms of time and space occupied. The linearization algorithm requires to visit each arc of the state machine graph, thus it has a complexity equal to the number of arcs. The `TestSequenceGeneration` algorithm complexity depends on the number of traces generated by the linearization algorithm, on the size of the inSD, and on the “granularity” of the inSD (i.e., how much the inSD scenario is incomplete, with respect to the state machines). In future work we plan to formally discuss the algorithm computational and time complexity.

6 Conclusions and Future Work

TESTOR is an algorithm, which taking in input state machines and scenarios, generates test sequences in the form of scenarios. The algorithm poses its basis on the idea that scenarios are usually incomplete specifications and represent important and expected system interaction. Such incomplete specifications may be “completed” by recovering, from state machines, the missing information. TESTOR has been implemented as a plugin component of the CHARMY analysis framework.

In an ongoing work we are integrating the test case selection approach implemented in TESTOR in our model-checking-based testing methodology proposed in [9]. We are starting applying the resulting tool supported methodology to

the Siemens C.N.X. system described in [20], and evaluating the efficacy of our approach with respect to Siemens C.N.X. design and analysis processes. Moreover, we are currently analyzing how TESTOR generated test sequences can be converted into executable tests.

Another interesting integration we have in mind is to use the TESTOR output as input for the Use Interaction Test (UIT) method [6]. Largely inspired to the Category Partition Method [22], UIT systematically constructs and defines a set of test cases for the Integration Testing phase, by using UML sequence diagrams as its exclusive reference model. We believe in this way we may combine the UIT main advantage of effort reduction with a more effective set of derived test cases.

Other improvements we have in mind are to analyze the possibility to produce linear traces and generate test sequences all at once, and to provide a richer formalism to express inSDs. In particular, we may wish to extract all such outSDs which correctly implement inSDs such as those in Figure 4: inSD1 (i.e., any path which includes *m1* followed by *m3*, excluding *m2* in between the two) and inSD2 (i.e., any path which includes *m1* immediately followed by *m2*, and eventually followed by *m3*).

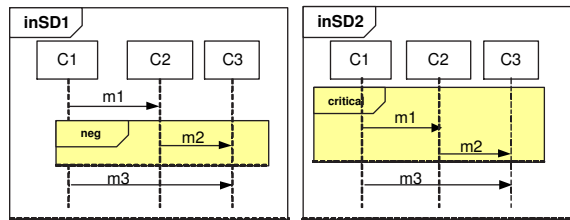


Fig. 4. inSD1 and inSD2 in the UML 2 formalism

Acknowledgment

The authors acknowledge Siemens C.N.X. who provided the case study and supports this research, Pierluigi Pierini for his contribution on a previous version of the paper, and Antonia Bertolino and Eda Marchetti which contributed on a previous paper with similar goals. We are indebted with the anonymous reviewers too which suggested relevant changes.

References

1. AGEDIS Project. <http://www.agedis.de/index.shtml>.
2. Model-based Testing Home Page. http://www.geocities.com/model_based_testing/. Maintained by Harry Robinson.

3. UMLAUT Project. <http://www.irisa.fr/UMLAUT/>.
4. CHARMY Project. Charmy Web Site. <http://www.di.univaq.it/charmy>, 2004.
5. G. Antoniol, L. C. Briand, M. D. Penta, and Y. Labiche. A Case Study Using the Round-Trip Strategy for State-Based Class Testing. In *Proc. IEEE ISSRE2002*, 2002.
6. F. Basanieri, A. Bertolino, and E. Marchetti. The Cow_Suite Approach to Planning and Deriving Test Suites in UML Project . In *Fifth International Conference on the Unified Modeling Language - the Language and its applications(UML 2002)*, pages 383–397, Dresden, Germany, September 2002.
7. A. Bertolino, E. Marchetti, and H. Muccini. Introducing a Reasonably Complete and Coherent Approach for Model-based Testing. In *In Testing and Analysis of Component-Based Systems Workshop, Tacos 2004*. To be published in Electronic Notes of Theoretical Computer Science, 2004.
8. L. Briand and Y. Labiche. A UML-Based Approach to System Testing. *Journal of Software and System Modelling (SoSyM)*, 1(1):10–42, 2002.
9. A. Bucchiarone, H. Muccini, P. Pelliccione, and P. Pierini. Model-Checking plus Testing: from Software Architecture Analysis to Code Testing. In *Proc. International Testing Methodology workshop*, Lecture Notes in Computer Science, LNCS, vol. 3236, pp. 351 - 365 (2004), October 2004.
10. I. Crnkovic and M. Larsson, editors. *Building Reliable Component-based Software Systems*. Artech House, July 2002.
11. F. Fraikin and T. Leonhardt. Seditec - testing based on sequence diagrams. In *Proc. IEEE CASE 02*, Edingburgh, September 2002.
12. P. Graubmann and E. Rudolph. HyperMSCs and Sequence Diagrams for use case modeling and testing. In *Proc. UML 2000*, volume LNCS Vol.1939, pages 32–46, 2000.
13. D. Harel and R. Marelly. Specifying and Executing Behavioural Requirements: The Play In/Play-Out Approach. *Journal of Software and System Modelling (SoSyM)*, 2003.
14. A. Hartman. Model Based Test Generation Tools. Technical report, AGEDIS project Downloads, 2002.
15. J. Hartmann, C. Imoberdof, and M. Meisenger. UML-Based Integration Testing. In *ACM Proc. ISSTA 2000, Portland*, 2000.
16. C. Jard and T. Jéron. TGV: Theory, Principles and Algorithms. In *Conf. IDPT 2002, Pasadena (USA)*, 2002.
17. G. Kim, H. S. Hong, D. H. Bae, and S. Cha. Test Cases Generation from UML State Diagram. *IEEE Proceedings - Software*, 146(4):187–192, August 1999.
18. L. Liuying and Q. Zhichang. Test Selection from UML Statecharts. In *Proc. of 31st Int. Conf. on Technology of Object-Oriented Language and System*, China, 22-25 September 1999.
19. E. Marchetti. *Software Testing in the XXI Century: Methods, Tools and New Approaches to Manage, Control and Evaluate This Critical Phase*. PhD thesis, University of Pisa, September 2003.
20. H. Muccini, P. Pelliccione, A. Bucchiarone, and P. Pierini. Software Architecture-driven System Testing through Model-Checking. Technical Report TRCS 035/2004, University of L'Aquila, 2004.
21. J. Offutt and A. Abdurazik. Generating Test from UML Specifications. In *Proc. UML 99, Fort Collins, CO, October 1999*.
22. T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 31(6):676–686, June 1988.

23. P. Pelliccione, H. Muccini, A. Bucchiarone, and F. Facchini. Deriving Test Sequences from Model-based Specifications. Technical Report TRCS 002/2005, University of L'Aquila, 2005.
24. J. Ryser and M. Glinz. Using Dependency Charts to Improve Scenario-Based Testing, 2000.
25. J. Wittevrongel and F. Maurer. Using UML to Partially Automate Generation of Scenario-Based Test Drivers. In Springer, editor, *OOS 2001*, 2001.

Appendix A: Test Sequence Generation

In this appendix we report a more complete description of the algorithm Test Sequence Generation, in the form of Pseudo-Code. It makes use of the following methods:

- **TracesSet** `extractTraces(SMId sm, Message m)`: this method extracts all traces containing the message m ;
- **SMId** `senderSM(Message m)`: this method returns the ID of the state machine that sends the message m
- **SMId** `receiverSM(Message m)`: this method returns the ID of the state machine that receives the message m
- **SDSet** `merge(SDSet sdSenderSet, SDSet sdReceiverSet)`: this method makes the merge of two sets of sequence diagrams. The two sets represent the actions required by the send and receive state machine to reproduce the behavior in the input sequence diagram. The algorithm attempts to construct for each element in the first set and for each element in the second one, a sequence that the system is able to execute. If it is impossible to obtain a correct behavior, the algorithm does not introduce any sequence in the output set. If the resulting set is empty, then it is impossible to obtain a correct behavior starting from the two input sets.
- **msgSet** `messages(Trace t)`: this method returns the set of messages contained into the input trace.
- **State** `startState(Trace t, Message m)`: this method returns the start state for a message contained into a trace.
- **Int** `size(Set s)`: this method returns the size of the set s .

The algorithm starts identifying such component which may send/receive m and selects the linearized traces containing m (lines 2-11). The trace selection is performed by means of the `extractTraces` method.

Test Sequence Generation Pseudo-Code

```

SDSet TestSequenceGeneration(Message m, SMSet smSet,
2 TraceSet trSet){
    sm_s = senderSM(m);
4    sm_r = receiverSM(m);
    itc_s = trSet(sm_s, smSet);
6    itc_r = trSet(sm_r, smSet);
    if (isEmpty(itc_s)){
8        itc_s = extractTraces(sm_s, m);
    }
10    if (isEmpty(itc_r)){
        itc_r = extractTraces(sm_r, m);
12    }

```

The next part of algorithm is composed of two equal parts: one for the sender component state machine (lines 13-40) and one for the receiver component state

machine (lines 41-68). A sender component is one which may send message m . Symmetrically, a receiver component is one which may receive message m . Each of them constructs the set of traces representing what the system must do to allow the exchanging of the message m . Focusing only on the sender part, we note that the algorithm makes use of the method `Validation` (line 15). This method is detailed in the following and intuitively identifies each trace which allows the system to exchange message m .

For each trace identified and for each new message introduced, a recursive call to `TestSequenceGeneration` is performed (line 23).

The `TestSequenceGeneration` is invoked with the sets $smSet$ and $trSet$ increased with the current state machine sm_s and with the current trace t , respectively. In fact the sense of the recursive invocation is to fill the sequence under construction with details needed for all the SM, except for the already considered ones. At each nested invocation, another SM with the related trace is added.

The rest of the code, lines 13-34, represents the construction of the set of sequence concatenating the results of the recursive invocations.

```

SDSet sdSets;
14  foreach t ∈ itcs{
    sms.cs = startState(t,m);
16  ist = Validation(t,m,sms.cs,sms.is,-);
    if (!isEmpty(ist)){
18      foreach el ∈ ist{
          if (size(ist)==1){
20          sdSets = ist;
          } else{
22          foreach message m' ∈ messages(el){
              /* ordered from the first to the last */
24          SDSet sdSetMsg = TestSequenceGeneration(m',smSet+sms,trSet+t);
              foreach sdMsg ∈ sdSetMsg{
26                  SDSet sdSetTmp;
                  if (isEmpty(sdSets)){
28                      add(sdSetTmp,sdMsg);
                  } else{
30                      foreach sd ∈ sdSets{
                          add(sdSetTmp,sd.sdMsg);
32                      }
34                  }
                  sdSets = sdSetTmp;
36              }
            }
38        }
40    }
42  /* ... */
    /* Code for the receiver */

```

The last part of the algorithm is the construction of the output set of sequence diagrams by merging the set of sequence diagrams required by the sender and the receiver state machines.

```

44  /* Merge of sdSets and sdSetr */
    SDSet sdSet = merge(sdSets,sdSetr);
    return sdSet;
46 }

```