Formal Test-case Generation for UML Statecharts *

Stefania Gnesi, Diego Latella and Mieke Massink
CNR/ISTI, Via Moruzzi 1, I56124 Pisa, ITALY
{Stefania.Gnesi,Diego.Latella,Mieke.Massink}@isti.cnr.it

Abstract

The Unified Modeling Language has been introduced as a notation for modeling and reasoning about large and complex systems, and their design, across a wide range of application domains. System modeling and analysis techniques, especially those based on formal methods, are more and more used for enhancing traditional System Engineering techniques for improving system quality. In particular this holds for model-based formal test case derivation using formal conformance testing. The contribution of the present paper is to provide a solid mathematical basis for conformance testing and automatic test case generation for UML Statecharts (UMLSCs). We propose a formal conformance-testing relation for input-enabled *transition systems with transitions labeled by input/output-pairs (IOLTSs). IOLTSs provide a suitable semantic model for a behavioral subset of UMLSCs. We also provide an algorithm which, for a UMLSC specification and the alphabet of implementations, generates a test suite. The algorithm is proven exhaustive and sound w.r.t. the conformance relation.*

1. Introduction and Related Work

Modern societies strongly depend, for their functioning as well as for the protection of their citizens, on systems of highly interconnected and interdependent infrastructures, which are increasingly based on computer systems. The complexity of such systems, and those of the near future, will be higher than that of any artifact which has been built so far. In recent years, the Unified Modeling Language (UML) [18] has been introduced as a notation for modeling and reasoning about large and complex systems, and their design, across a wide range of application domains. Moreover system modeling and analysis techniques, especially those based on formal methods, are more and more used for enhancing traditional System Engineering techniques for improving system quality. In particular this holds for model-based formal test case derivation using formal conformance testing, of which the present paper addresses the theoretical foundations.

Testing and conformance relations in the context of labeled transition systems (LTSs) have been thoroughly investigated in the literature. Broadly speaking, conformance testing refers to a field of theory, methodology and applications for testing that a given implementation of a system *conforms* to its abstract specification, where a proper conformance relation is defined using the formal semantics of the notation(s) at hand. An account of the major results in the area of testing and conformance relations can be found in [9, 22]. The theory has been developed mainly in the context of process algebras and input/output transition systems.

In [13, 14] and in this paper we set the theoretical basis for testing and conformance theories for UML Statecharts (UMLSCs, for short), thus making them available for practitioners in industry where the Unified Modeling Language has become a de facto standard, in particular for the development of complex systems[1]. The UML consists of a number of diagrammatic specification languages, among which UMLSCs, that are intended for the specification of behavioral aspects of software systems. This diagrammatic notation differs considerably from process algebraic notations. In UMLSCs, transitions are labeled by input/output-*pairs* (i/o-pairs), where the relation between input and output is maintained at the level of the *single* transitions. This is neither the case in traditional testing theories, like [9], where no distinction is made between input and output, nor for the input/output transition systems used in standard conformance testing theory [22]. In our approach we use LTSs labelled over i/o-pairs where a generic transition models a *step* of the associated statechart (*step*-transition). Preserving the atomicity of input acquisition and output generation in a single step has two important advantages. First of all, this reflects in a more direct way the semantics of UMLSCs

* This work has been carried out under the Agreement between CNR CNUCE/IEIand GMD in the frame of the Project "Formal Test Cases Derivation for UML Statechart Diagrams Specifications". It has been partially funded by projects EU-IST IST-2001-32747 (AGILE) and MIUR/SP4.

1 Although we refer to UML 1.5, the main features of the notation of interest for our work have not changed in later versions.

steps; each *step*, according to [18], is triggered by an input event and causes *both* a change in the current configuration *and* the execution of certain actions such as those that generate output events. The use of LTSs with separate input and output events [22] would require the introduction of additional, intermediate, global states and transitions at the semantics level, thus breaking the neat correspondence between the notion of *step* of the statechart and that of *step*-transition of its associated LTS. Secondly, a testing theory based on i/o-pairs preserves the compatibility of the testing models with the rest of the semantics framework that we have developed for UMLSCs (e.g. [11, 8]) and that is based on re-use of a basic set of deduction rules (the "core semantics") leading to a high degree of homogeneity, modularity and re-use.

Our LTSs labelled over i/o-pairs are very similar to Finite State Machines (FSMs), in particular Mealy Machines. A considerable number of studies in the field of testing FSMs are available in the literature. An excellent survey can be found in [16]. Many such proposals deal with test case generation but mainly in the context of *deterministic* machines. In some proposals, like the one in [3], further restrictions on the machines are introduced, requiring that they must be strongly connected.

To our knowledge, the study of conformance relations, and of testing theory in general, in the context of *non-deterministic* machines, or LTSs over i/o-pairs, has received scant attention. On the other hand, non-determinism is a key notion in the area of formal approaches to system modeling and verification and, in fact, it is a central notion in traditional testing theories for LTSs [9] and their variants for systems with inputs and outputs [13, 14, 17].

In this paper we propose a formal conformance testing relation and a test case generation algorithm for *input enabled* labeled transition systems over i/o-pairs (IOLTSs). IOLTSs are LTSs where each state has (at least) one outgoing transition for each element of the input alphabet of the transition system. Intuitively, such transition systems cannot refuse any of the specified input events, in the sense that they cannot deadlock when such events are offered to them by the external environment. Whenever a machine, in a given state, *does not* react on a given input, its modeling IOLTS has a specific loop-transition from the corresponding state to itself, labelled by that input and a special "stuttering" output-label.

IOLTSs have been used as semantic model for a *behavioral subset* of UMLSCs [7]. In this paper, we will assume that system specifications are given as UMLSCs and we will concentrate on their associated IOLTSs. IOLTSs are suitable also for modeling implementations of systems specified by such diagrams. Modeling implementations as input enabled transition systems is common practice in the context of formal conformance testing - see e.g. [22]. We focus on conformance testing and the soundness and exhaustiveness properties of a test case generation algorithm relative to a conformance relation for IOLTSs. The conformance relation we define is similar to the one of Tretmans [22], with adaptations which take care of our semantic framework for UMLSCs. As a by-product of our work, we also define and propose a specific test case language.

In [13] we defined a general testing theory for UMLSCs, using a framework similar to that proposed in [17], which was in turn inspired by the work of Hennessy for traditional LTSs [9]. The general approach of the above mentioned theories is based on the well known notions of *MAY* and *MUST* preorders and related equivalences. Intuitively, for systems A and B, $A \sqsubseteq_{MAY} B$ means that if a generic experimenter (i.e. test case) E has a successful test run while testing A, then E has also a successful test run when testing B. On the other hand, $A \sqsubseteq_{MUST} B$ means that if all test runs of a generic experimenter E are successful when testing A, then it must be the case that all test runs of E are successful when testing B. It can be shown that $\sqsubseteq_{MAY}$ coincides with trace inclusion and that $A \sqsubseteq_{MUST} B$ implies $B \sqsubseteq_{MAY} A$. Thus, the testing preorders focus essentially on the observable behaviour of systems and are strongly related to their internal non-determinism and deadlock capabilities; intuitively, if both $A \sqsubseteq_{MAY} B$ and $A \sqsubseteq_{MUST} B$ hold, then A is "more non-deterministic" than B and can generate more deadlocks than B can, when tested by an experimenter. Finally, if also the reverse preorders hold, i.e. $B \sqsubseteq_{MAY} A$ and $B \sqsubseteq_{MUST} A$ as well, then A and B are *testing equivalent* since no experimenter can distinguish them. The main semantic assumptions in [9] are that (i) system interaction is modeled by action-synchronization rather than input/output exchanges, and (ii) absence of reaction from a system to a stimulus presented by an experimenter results in a deadlock affecting both the system and the experimenter. In [17], and later in [13] specifically for UMLSCs, assumption (i) has been replaced by modeling system interaction as input/output exchanges, but assumption (ii) remained unchanged. In particular, in [13], absence of reaction of a given state s on a given input i is represented by the absence of *any* transition with such an input i from s, in a way which is typical of the process-algebraic approach. We refer to the resulting semantic model as the "non-stuttering" one, as opposed to input enabled IOLTS, i.e. the "stuttering" semantics, used in the present paper. The above two different ways of dealing with absence of reaction, and in particular, the ability for experimenters to explicitly *detect* absence of reaction turns out to be of major importance for determining the relative expressive power of the various semantics. More specifically, in [14] we defined *MAY* and *MUST* preorders also for the stuttering semantics and we provided a formal comparison between the Hennessy-like, non-stuttering semantics [13, 9], and the stuttering seman-

tics w.r.t. testing and conformance ordering relations; we showed that if two UMLSCs, say A and B, are in conformance relation (i.e. A conforms to B) in the stuttering semantics, then they are also in *MAY* and in the reverse-*MUST* relations (i.e. $A \precsim_{MAY} B$ and $B \precsim_{MUST} A$) in the non-stuttering semantics, *but not vice-versa*. This shows that the Hennessy-like, non-stuttering, semantics [13, 9] is not adequate for reasoning about issues of conformance, since the detection of absence of reaction, explicitly modeled only in the stuttering semantics, plays a major role when dealing with conformance. Accordingly, the following results have been proven: the conformance relation coincides with the *MAY* preorder *in the stuttering semantics*. Moreover, in the stuttering semantics, nice substitutivity properties hold; for instance, testing equivalent implementations conform to the same specifications and implementations conform to testing equivalent specifications.

Related work on automatic test generation based on UMLSCs is being developed in the context of the Agedis project [20]. In that approach a system model, composed of class, object and statechart diagrams is translated into a model expressed in an intermediate format suitable as input for model checking and test generation tools. It follows a pragmatic, industrial approach with a clear focus on the test selection problem, but with less emphasis on UML formal semantics. In contrast, we follow a 'Semantics-first' approach (also) with respect to conformance testing. Similarly, in [15] emphasis is put primarily on support tool implementation. Other approaches to automatic test generation include [19] that describes the use of the CASE tool AutoFocus. The authors emphasize the need for a formally defined semantics and use state transition diagrams that resemble a subset of the UML-RT, but it seems there is no formal relation between their diagrams and the subset of the UML-RT. Automated test generation has been developed also for classical Harel statechart diagrams, e.g. [2], which semantically differ considerably from UMLSCs (e.g., a different priority schema as well as a different semantics for the input queues are used).

The paper is organized as follows: in Sect. 2 IOLTSs are defined and a running example is introduced showing how IOLTSs can be used as semantic model for UMLSCs. The notion of conformance and the formal definition of the conformance relation are given in Sect. 3, together with the formal definition of the notion of test case and an account of what it formally means for a system to pass a test case and/or a test suite. The test case generation algorithm is defined in Sect. 4 where its completeness theorem is also provided. The application of the algorithm is illustrated by the derivation of some test cases for the example of Sect. 2. Some conclusions and lines for future research are discussed in Sect. 5. The proofs of the results presented in this paper can be found in [7], where all technical details of the operational semantics definition are given as well.

2. IOLTSs and UMLSCs

In this section we summarize the basic definitions concerning IOLTSs, which are necessary for developing the notions of conformance and conformance testing. The example of Fig. 1 shows the IOLTS (b) of a simple UMLSC (a). It will be briefly discussed in Sect. 2.1 and will be used as the running example throughout the paper. The formal definition of the operational semantics of UMLSCs based on IOLTSs is outside the scope of this paper. The interested reader is referred to [7]. Here we only point out that the IOLTSs semantics of UMLSCs is essentially the same as that proposed in [13]. The only difference from [13] is the way *stuttering* is dealt with. In the context of UMLSCs, *stuttering* occurs when no transition of the UMLSC is enabled by the current event e in the current (global) state σ of the underlying state-machine. In the semantics proposed in [13] no step-transition with input label e leaves σ. Thus, the absence of reaction on input e is modeled *implicitly* by the absence of corresponding transitions. This approach is quite standard in the context of general testing theory. In the IOLTSs semantics proposed in [7], instead, stuttering is modeled *explicitly*: in the above situation, a *step*-transition, with input label e and a special output symbol Σ, denoting stuttering, leaves σ and points back to σ[2].

As in [13], we consider a subset of UMLSCs, which includes all the interesting conceptual issues related to concurrency in dynamic behavior—like sequentialisation, nondeterminism and parallelism—as well as UMLSCs specific issues—like state refinement, transition priorities, interlevel/join/fork transitions. More specifically, we do not consider history, action and activity states; we restrict events to signals without parameters (actually we do not interpret events at all); time and change events, object creation and destruction events, and deferred events are not considered neither are branch transitions; also variables and data are not allowed so that actions are required to be just (sequences of) events. We also abstract from entry and exit actions of states. The definition of a sound "basic" kernel of a notation, to be extended only after its main features have been investigated, has already proven to be a valuable and fruitful methodology and is often standard practice in many fields of concurrency theory, like process-algebra. We refer to e.g. [11] for a deeper discussion on such "basic-notation-first" and "semantics-first" versus "full-notation-first" issue.

2 This notion of stuttering is the UMLSCs analogous of *quiescence* in the context of LTSs with separate input and output label sets, like in [21]. Explicit representation of quiescence is common practice in the study of formal conformance relations.

2.1. Basic definitions

In the following we give the basic definitions of LTS and IOLTS and we briefly discuss the example of Fig. 1.

Definition 2.1 (LTS) *A LTS* $\mathcal{S}$ *is a tuple* $(S, \sigma^0, L, \longrightarrow)$ *where* S *is the* set of states *with* $\sigma^0 \in S$ *being the* initial *state,* L *is the* set of (transition) labels *and* $\longrightarrow \subseteq S \times L \times S$ *is the* transition relation *of the LTS.* ∘

For $(\sigma, l, \sigma') \in \longrightarrow$ we write $\sigma \xrightarrow{l} \sigma'$. The notation $\sigma \xrightarrow{l}$ will be a shorthand for $\exists \sigma'.\ \sigma \xrightarrow{l} \sigma'$. Some standard definitions are given below [3].

Definition 2.2 *For LTS* $\mathcal{S} = (S, \sigma^0, L, \longrightarrow)$, $\sigma, \sigma', \sigma'' \in S$, $l \in L$, $\gamma \in L^*$

- *The transition relation* $\xrightarrow{\gamma}$ *over finite sequences is defined in the obvious way: (a)* $\sigma \xrightarrow{\epsilon} \sigma$ *and (b) if* $\sigma \xrightarrow{\gamma} \sigma'$ *and* $\sigma' \xrightarrow{l} \sigma''$*, then* $\sigma \xrightarrow{\gamma l} \sigma''$*;*
- *The* language *of* $\mathcal{S}$ *is the set of all its* traces*:* $Lan\ \mathcal{S} = \{\gamma \in L^* \mid \exists \sigma'.\ \sigma^0 \xrightarrow{\gamma} \sigma'\}$*;*
- *The states of* σ after γ *is the set* $(\sigma\ \texttt{after}\ \gamma) = \{\sigma' \mid \sigma \xrightarrow{\gamma} \sigma'\}$. ∘

In this paper we will use LTSs where the labels in L are i/o-pairs, i.e. $L = L_I \times L_U$, for some input set L_I and output set L_U. For such LTSs the following auxiliary definitions apply:

Definition 2.3 *For LTS* $\mathcal{S} = (S, \sigma^0, L_I \times L_U, \longrightarrow)$, $\sigma \in S$, $i \in L_I$, $\gamma \in L^*$ *and set* $Z \subseteq S$*:*

- *the* output *of* Z *on* i *is the set* $(\texttt{out}\ Z\ i) = \bigcup_{\sigma' \in Z} \{u \in L_U \mid \sigma' \xrightarrow{(i,u)}\}$*; we let* $(\texttt{OUT}\ \sigma\ \gamma\ i)$ *be the set* $(\texttt{out}\ (\sigma\ \texttt{after}\ \gamma)\ i)$*;*
- $\mathcal{S}$ *is* input enabled *iff* $\forall \sigma' \in S, i \in L_I.\ \exists u \in L_U.\ \sigma' \xrightarrow{(i,u)}$.

We need similar operators as `after` and `OUT` also for sets of traces over $(L_I \times L_U)^*$.

Definition 2.4 *For* $\mathcal{F} \subseteq L^*$, $i \in L_I$, $\gamma \in L^*$

- *the traces of* $\mathcal{F}$ after γ *is the set* $(\mathcal{F}\ \underline{\texttt{after}}\ \gamma) = \{\gamma' \mid \gamma\gamma' \in \mathcal{F}\}$*;*
- *the* output *of* $\mathcal{F}$ *on* i *is the set* $(\underline{\texttt{out}}\ \mathcal{F}\ i) = \{u \in L_U \mid \exists \gamma.\ (i,u)\gamma \in \mathcal{F}\}$*; we let* $(\underline{\texttt{OUT}}\ \mathcal{F}\ \gamma\ i)$ *be the set* $(\underline{\texttt{out}}\ (\mathcal{F}\ \underline{\texttt{after}}\ \gamma)\ i)$. ∘

3 In this paper we will freely use a functional programming like notation where currying will be used in function application, i.e. $f\ a_1\ a_2 \ldots\ a_n$ will be used instead of $f(a_1, a_2, \ldots, a_n)$ and function application will be considered left-associative. Moreover, for set X, the set of finite sequences over X will be denoted by X^*; for $x \in X$ we let x denote also the sequence in X^* composed by the single element x, while for $\gamma, \gamma' \in X^*$ we let the juxtaposition $\gamma\gamma'$ of γ with γ' denote their concatenation.

Definition 2.5 (IOLTS) *An* IOLTS *labeled over* $L_I \times L_U$ *is a LTS* $(S, \sigma^0, L_I \times L_U, \longrightarrow)$ *which is* input enabled. ∘

The operational semantics of a UMLSC is defined in [7] as a IOLTS, where transitions are characterized by the *step*-relation. Every step-transition models the collective firing of a maximal set of enabled non-conflicting transitions of the UMLSC which do not violate transition priority constraints [18, 12, 8].

The input component of the i/o-pair of a step-transition is a single event which represents the stimulus for the transitions to fire while the output component is a collection of events that the UMLSC returns to the environment as (part of) the reaction to the stimulus (the other part being represented by the change in its global state). When stuttering occurs, the output component is the special symbol Σ. Thus, in the remainder of this paper we will focus on IOLTSs labeled over $L_I \times L_U$ where Σ, with $\Sigma \notin L_I$, may belong to L_U. In the following we will often use the word 'transition' both for those of UMLSCs and for the step-transitions of their associated semantics.

In the official definition of the UML [18], the dispatching policy of events to state machines by their external environment is not specified. In our proposals for the formal semantics of UMLSCs we used a parametric abstract data type approach for modeling the environment policy. As a consequence, also the collections of events generated by (the parallel execution of) more than one transition within a step have been represented by instances of such data-types.

For the sake of simplicity, in the following examples we will model such collections of events simply as sets, since the dispatching policy is of no conceptual influence for formal test case generation.

A sample UMLSC, H, is shown in Fig. 1 (a). The *set of input events* of H is $L_I = \{a1, a2, e1, e2, f1, f2, r1, r2\}$. The IOLTS of H, as obtained by applying the formal operational semantics definition presented in [7], is shown in Fig. 1 (b); labels (i, u) are drawn as i/u in the picture. For simplicity, several stuttering loops from/to the same state, labeled by $i_1/\Sigma, \ldots, i_k/\Sigma$ have been collapsed to a single loop labeled by $i_1, \ldots, i_k/\Sigma$. Notice that more than one output event can be generated by a single step, as a consequence of internal parallelism in H. For instance, in the transition from state 2 to state 6 of the IOLTS events $e1$ and $e2$ are both generated as a reaction to receiving $e2$. State 2 of the IOLTS corresponds to configuration $\{s0, s1, s7, s8, s10\}$ of H. The reader can easily check that the LTS is indeed input enabled over $L_I \times L_U$, where set L_U is $\{\Sigma, \emptyset, \{a1\}, \{a2\}, \{e1\}, \{e2\}, \{f1\}, \{r1\}, \{r2\}, \{e1, e2\}\}$ We close this section remarking that the LTSs generated according to the UMLSCs semantics definition proposed in [7] are *finite*: the number of their states is finite as well as their number of transitions.

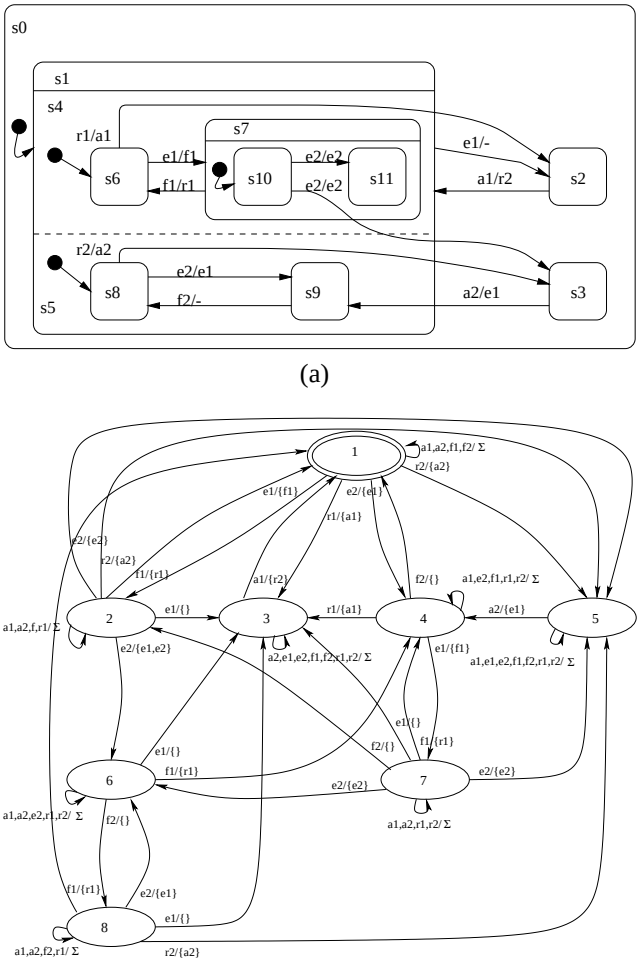

Figure 1. A UMLSC and its IOLTS

2.2. Pragmatics

In the context of the present work, we assume that a *specification* of system behavior is given in the form of a UMLSC H and we make reference mainly to its semantics, namely a IOLTS labeled over $L = L_I \times L_U$ for proper L_I and L_U, which we denote by $\mathbf{IOLTS}\ H$. An *implementation* for H will be modeled by an IOLTS labeled over $L' = L'_I \times L'_U$ (with L'_I not necessarily equal to L_I). Under the above assumptions, for simplicity, we often speak of specifications over L and implementations over L'. We remind the reader that $\Sigma \notin L_I \cup L'_I$ is assumed while $\Sigma \in L_U$ (resp. $\Sigma \in L'_U$) represents stuttering of the specification (resp. implementation). Notice that we do not require that IOLTSs modeling implementations are necessarily generated from UMLSCs. Any such a model can be obtained by any means, obviously including, but not limited to the case in which the implementation is itself a UMLSC. The above assumptions are quite standard in the context of formal conformance theory and its application [21].

3. Conformance Testing and the Conformance Relation

As we briefly discussed in the previous section, from our point of view, both specifications and implementations are *modeled* as IOLTSs. A discussion on the adequacy of LTSs as models for specifications and implementations is outside the scope of the present paper. The interested reader is referred to [22].

Under the above modeling assumption, one of the most successful formal conformance relations is the **ioco** relation proposed by Tretmans [22]. Informally, for specification $\mathcal{S}$ and implementation $\mathcal{S}'$, $\mathcal{S}'$ **ioco** $\mathcal{S}$ means that $\mathcal{S}'$ can never produce an output which could not be produced by $\mathcal{S}$ "in the same situation", i.e. after the same sequence of steps.

In [22], inputs and outputs are "irregularly" scattered throughout the LTS, and a "quiescence" transition from a state means that in this particular state no output is produced by the system. We remark that, in such an approach, input is not (always) required in order to produce some output. In our setting, there is a clear causal relation between input and related output. They both appear in the same transition. A stuttering transition in a given state—actually a stuttering loop—is labelled by (i, Σ), which means that in that state the system produces no output, or better, does not react at all, *on input* i.

On the basis of the above considerations, with particular reference to the role played by the *input* events of transitions, we give the following definition of our conformance relation. We define it for generic LTSs over i/o-pairs, although we will use it only for input-enabled ones. Finally, we point out that we actually define a *class* of conformance relations, in a similar way as in [21]. The class is indexed by a set $\mathcal{F}$ of traces which determines the discriminatory power of the relation. Such a parametric definition turns out to be of technical help in the definition of the test case generation algorithm in the next section and in the proof of its properties. The definition of the *Conformance Relation* $\sqsubseteq_{\mathtt{co}}^{\mathcal{F}}$ follows:

Definition 3.1 *For LTSs* $\mathcal{S} = (S, \sigma^0, L_I \times L_U, \longrightarrow)$, $\mathcal{S}' = (S', \sigma^{0\prime}, L'_I \times L'_U, \longrightarrow')$ *and* $\mathcal{F} \subseteq (L_I \times L_U)^*$*:* $\mathcal{S}' \sqsubseteq_{\mathtt{co}}^{\mathcal{F}} \mathcal{S}$ *iff* $\forall \gamma \in \mathcal{F}, i \in L_I.\ \mathtt{OUT}\ \sigma^{0\prime}\ \gamma\ i \subseteq \mathtt{OUT}\ \sigma^0\ \gamma\ i$ ∘

In the following we will let $\sqsubseteq_{\mathtt{co}}$ (i.e. "conforms to") denote $\sqsubseteq_{\mathtt{co}}^{(Lan\ \mathcal{S})}$—we remind that $Lan\ \mathcal{S}$ denotes the language of $\mathcal{S}$—and we will mainly focus on $\sqsubseteq_{\mathtt{co}}$. Intuitively, $\mathcal{S}' \sqsubseteq_{\mathtt{co}} \mathcal{S}$ means that $\mathcal{S}'$ can never produce an output which could not be produced by $\mathcal{S}$ in the same situation, i.e. after the same i/o sequence *and the same input*. In general, it is not required that $L_I = L'_I$: for partial specifications we have that

$L_I \subseteq L'_I$, while for incomplete implementations we have that $L'_I \subseteq L_I$; The case that $L_I \cap L'_I = \emptyset$ does not make so much sense. Notice that when $\Sigma \in L_U$ the above definition implies that $\mathcal{S}'$ may produce no output at all due to stuttering only if $\mathcal{S}$ can do so. This is also the case in [21, 22] but its technical definition has been adapted here for UMLSCs.

In the next section we will define the test case generation algorithm. Before we can proceed, however, we need to define precisely what a test case is and what testing an implementation against such a test case means. The remainder of this section will be devoted to these definitions, which are inspired by those given in [13]. The basic notions behind them have been introduced in [9] and [17].

Intuitively, a test case is a specially customized 'environment' which interacts with the implementation under test by providing it with an event, collecting all the output generated by the implementation as a reaction to that event, analyzing its output and behaving accordingly: in particular it may (i) report success and/or (ii) provide the implementation with a new event and wait for the new related output and so on, or (iii) decide to stop testing. It is important to point out that, after providing the implementation with an event, the test case *must* be prepared to receive *any* possible outcome of the machine. If the implementation is an IOLTS over $L_I \times L_U$, such outcome can be any element of L_U.

Definition 3.2 (Test Case) *A* Test Case $\mathcal{T}$ *over* $L_I \times L_U$ *is a tuple* $(T_U, \upsilon^0, T_I, L_I, L_U, \longrightarrow)$ *where* T_U *is the set of* output *states, with* $\upsilon^0 \in T_U$ *being the* initial *(output) state,* $T_I \subseteq L_U \mapsto T_U$ *is the set of* input *states, each input state being a* total *function from* L_U *to output states. Finally* $\longrightarrow \subseteq (T_U \times L_I \times T_I) \cup (T_U \times \{\tau, \mathbf{W}\} \times T_U)$ *is the* transition relation*, with* $(L_I \cup L_U) \cap \{\tau, \mathbf{W}\} = \emptyset$. ∘

A test case is similar to a transition system where some states—namely the *input* states, i.e. states in which the test case is supposed to get some output from the implementation—are actually *total functions* from L_U to output states. Totality guarantees that *any* output of the implementation is accepted by the test case for analysis in that state and, on the basis of the particular value received, the test case will move to the next output state. Notice that $\Sigma \in L_U$ makes test cases able to detect stuttering. It is also worth pointing out here that, although for generality in the above definition an input state is a function in $L_U \mapsto T_U$, for any practical purposes it is sufficient to consider finite functions [13]. Output states are those in which the test case can (i) produce specific events to be delivered to the implementation, or (ii) silently move, via τ, to other (output) states—thus a test case can be internally non-deterministic—or (iii) produce the special action $\mathbf{W}$ by which the test case reports success. We say that a test case $\mathcal{T}$ is *finite* whenever T_U, T_I and $\longrightarrow$ are finite sets.

Testing an IOLTS over $L_I \times L_U$ against test case $\mathcal{T}$ amounts to the *Experimental System* they characterize:

Definition 3.3 (Experimental System) *For IOLTS* $\mathcal{S} = (S, \sigma^0, L_I \times L_U, \longrightarrow)$ *and test case* $\mathcal{T} = (T_U, \upsilon^0, T_I, L_I, L_U, \longrightarrow)$ *, the experimental system* $< \mathcal{T}, \mathcal{S} >$ *is the transition system* $(T_U \times S, (\upsilon^0, \sigma^0), \rightsquigarrow)$*. The transition relation* $\rightsquigarrow \subseteq (T_U \times S) \times (T_U \times S)$ *is the smallest relation induced by the deduction system below where* $\sigma, \sigma' \in S$*,* $\upsilon, \upsilon' \in T_U$*,* $\iota \in T_I$ *and for* $((\upsilon, \sigma), (\upsilon', \sigma')) \in \rightsquigarrow$ *we write* $\upsilon \,||\, \sigma \rightsquigarrow \upsilon' \,||\, \sigma'$

$$\frac{\upsilon \xrightarrow{\ i\ } \iota,\ \sigma \xrightarrow{(i,u)} \sigma',\ (\iota\ u) = \upsilon'}{\upsilon \,||\, \sigma \rightsquigarrow \upsilon' \,||\, \sigma'} \qquad \frac{\upsilon \xrightarrow{\ \tau\ } \upsilon'}{\upsilon \,||\, \sigma \rightsquigarrow \upsilon' \,||\, \sigma}$$

The Success *set of the experimental system is the set* $\{\upsilon \in T_U \mid \exists \upsilon' \in T_U.\ \upsilon \xrightarrow{\mathbf{W}} \upsilon'\}$ ∘

Notice that in the first rule in the above definition function ι is applied to u to obtain the next (output) state of υ, namely υ'. The effect of silent moves of test cases is defined by the second rule. Test runs are modeled by *computations*:

Definition 3.4 (Computations) *A* computation *of experimental system* $< \mathcal{T}, \mathcal{S} >$ *is a sequence of the form:*

$$\upsilon_0 \,||\, \sigma_0 \rightsquigarrow \upsilon_1 \,||\, \sigma_1 \rightsquigarrow \upsilon_2 \,||\, \sigma_2 \rightsquigarrow \ldots \upsilon_k \,||\, \sigma_k \rightsquigarrow \ldots$$

which is maximal*, i.e. either it is infinite or it is finite with terminal element* $\upsilon_n \,||\, \sigma_n$ *which has the property that* $\upsilon_n \,||\, \sigma_n \rightsquigarrow \upsilon' \,||\, \sigma'$ *for no pair* υ'*,* σ'*.* υ_0 *and* σ_0 *are the initial states of* $\mathcal{T}$ *and* $\mathcal{S}$*.* ∘

We let $\mathrm{Comp}(\mathcal{T}, \mathcal{S})$ denote the set of all computations of $< \mathcal{T}, \mathcal{S} >$. A computation is *successful* iff $\upsilon_k \in$ *Success* for some $k \geq 0$.

A *verdict* is the result of testing a system $\mathcal{S}$ against a test case $\mathcal{T}$. The test is passed if all computations are successful:

Definition 3.5 (Verdict) *The* verdict $\mathcal{V}$ *of* $\mathcal{T}$ *on* $\mathcal{S}$ *is defined as follows:*

$$\mathcal{V}\ \mathcal{T}\ \mathcal{S} = \begin{cases} \textbf{pass} & \textit{if } \mathrm{Comp}(\mathcal{T}, \mathcal{S}) \textit{ contains only successful computations} \\ \textbf{fail} & \textit{otherwise} \end{cases}$$

∘

A *test suite* is a set of test cases. The *verdict* function is extended to test suites in the obvious way; for test suite TS

$$\mathcal{V}\ TS\ \mathcal{S} = \begin{cases} \textbf{pass} & \text{if } \forall \mathcal{T} \in TS.\ \mathcal{V}\ \mathcal{T}\ \mathcal{S} = \textbf{pass} \\ \textbf{fail} & \text{otherwise} \end{cases}$$

The following definition relates test suites to specifications using conformance relations and introduces the notions of sound and exhaustive test suites.

Definition 3.6 (Soundness and Completeness) *Given specification $\mathcal{S}$ and test suite TS*

- *TS is* sound *w.r.t. $\mathcal{S}$ and $\sqsubseteq_{\mathtt{co}}^{\mathcal{F}}$ iff $\mathcal{S}' \sqsubseteq_{\mathtt{co}}^{\mathcal{F}} \mathcal{S}$ implies $\mathcal{V}\ TS\ \mathcal{S}' =$ **pass**, for all implementations $\mathcal{S}'$;*
- *TS is* exhaustive *w.r.t. $\mathcal{S}$ and $\sqsubseteq_{\mathtt{co}}^{\mathcal{F}}$ iff $\mathcal{V}\ TS\ \mathcal{S}' =$ **pass** implies $\mathcal{S}' \sqsubseteq_{\mathtt{co}}^{\mathcal{F}} \mathcal{S}$, for all implementations $\mathcal{S}'$.*

We say that a test suite is complete if it is both exhaustive and sound. ∘

4. Automatic Test Case Generation

In this section we define the test case generation algorithm. The algorithm generates test cases written in a language introduced in [13], which is a mix of process algebra (guarded action prefix, choice, and process definition/instantiation) and a simplified version of the lambda-calculus.

4.1. The test language

Let IE be a set of *events* and OS be a set of *possible outputs* such that $\Sigma \in OS \setminus IE$ and $(IE \cup OS) \cap \{\tau, \mathbf{W}\} = \emptyset$. The abstract syntax of *output* test expressions $\mathcal{U}$, resp. *input* test expressions $\mathcal{I}$, of the language is given below, where $e \in IE$ is an event, $\alpha \in \{\tau, \mathbf{W}\}$, $U \subseteq OS$, P and x are test and input variables respectively, g is a boolean expression of the form "$x = \Gamma$" for $\Gamma \in OS$ or "$x \notin X$" for $X \subseteq OS$. The notion of free (input) variable is the same as in lambda-calculus. Brackets as well as proper indentation will be used whenever necessary.

$$\mathcal{U} ::= \delta \mid e;\mathcal{I} \mid \alpha;\mathcal{U} \mid g \Rightarrow \mathcal{U} \mid \mathcal{U} + \mathcal{U} \mid P$$

$$\mathcal{I} ::= \lambda\, x : U.\mathcal{U}$$

A test case specification consists of a pair $(\mathcal{U}, U)$ where $\mathcal{U}$ is an output test expression and $U \subseteq OS$. We will require that no input variable occurs free in $\mathcal{U}$ and that a proper test definition is associated with any test variable occurring in $\mathcal{U}$ in the context where the test case specification is used. Moreover, all input test sub-expressions of $\mathcal{U}$ must use the same set U in their defining lambda-expression.
The test δ performs no action. Expression $e;\mathcal{I}$ offers event e and then behaves like $\mathcal{I}$ which is an input test expression, namely a function. Such a function will be applied to the output produced by an implementation under test in an experimental system (see Def.3.3 in Sect. 3). The specific (output) state resulting from the application is obtained according to the semantics of *input* test expressions, as given by the following rewrite rule for function application:

$$(\lambda\, x : U.\mathcal{U})\ \Gamma = \mathcal{U}[\Gamma/x]$$

$$e;\mathcal{I} \xrightarrow{e} \mathcal{I} \qquad \alpha;\mathcal{U} \xrightarrow{\alpha} \mathcal{U}$$

$$\frac{\mathcal{U} \xrightarrow{\mu} \mathcal{E}}{\mathcal{U} + \mathcal{U}' \xrightarrow{\mu} \mathcal{E}} \qquad \frac{\mathcal{U} \xrightarrow{\mu} \mathcal{E}}{\mathcal{U}' + \mathcal{U} \xrightarrow{\mu} \mathcal{E}}$$

$$\frac{\mathcal{U} \xrightarrow{\mu} \mathcal{E}}{TRUE \Rightarrow \mathcal{U} \xrightarrow{\mu} \mathcal{E}} \qquad \frac{P := \mathcal{U}, \mathcal{U} \xrightarrow{\mu} \mathcal{E}}{P \xrightarrow{\mu} \mathcal{E}}$$

Figure 2. Test Expressions Operational Semantics

where $\mathcal{U}[\Gamma/x]$ denotes $\mathcal{U}$ where all free occurrences of x are simultaneously replaced by Γ. Notice that the above rule is a simplification of β-reduction of the lambda-calculus since Γ is just an element of OS: It cannot contain variables or lambda-expressions. Expression $\alpha;\mathcal{U}$ produces α and then behaves like $\mathcal{U}$. Notice that α can be either τ or the success action $\mathbf{W}$, so no interaction with implementations can take place (see again Def.3.3 in Sect. 3). In order for a guarded action prefix to proceed it is necessary that the guard evaluates to true. The choice expression $\mathcal{U}_1 + \mathcal{U}_2$ behaves as $\mathcal{U}_1$ or $\mathcal{U}_2$. Finally, if $P := \mathcal{U}$ is the definition for P, P behaves like $\mathcal{U}$.

The operational semantics of test case specifications is given in a similar way as for process algebra, by means of the Structural Operational Semantics rules of Fig. 2 where $\mu \in IE \cup \{\tau, \mathbf{W}\}$ and $\mathcal{E}$ stands both for output and for input test expressions.

In order to formally derive the test case denoted by a test case specification we first need a couple of auxiliary definitions where by $\vdash \mathcal{U} \xrightarrow{\mu} \mathcal{E}$ we mean that $\mathcal{U} \xrightarrow{\mu} \mathcal{E}$ is derivable using the rules of Fig. 2,

Definition 4.1 (Derivatives) *The derivatives of test case specification $(\mathcal{U}, U)$ is the smallest set $\mathcal{D}_{(\mathcal{U},\, U)}$ of test expressions which satisfies the following three conditions:*

1. *$\mathcal{U} \in \mathcal{D}_{(\mathcal{U},\, U)}$;*
2. *if output test expression $\mathcal{U}'$ is in $\mathcal{D}_{(\mathcal{U},\, U)}$ and $\vdash \mathcal{U}' \xrightarrow{\mu} \mathcal{E}$ then also $\mathcal{E}$ is in $\mathcal{D}_{(\mathcal{U},\, U)}$;*
3. *if input test expression $\mathcal{I}$ is in $\mathcal{D}_{(\mathcal{U},\, U)}$ then $(\mathcal{I}\ u)$ is in $\mathcal{D}_{(\mathcal{U},\, U)}$ for all $u \in U$.* ∘

Definition 4.2 (Labels) *The labels of test case specification $(\mathcal{U}, U)$, $Lab(\mathcal{U})$ is defined recursively as follows:*
$Lab(\delta) = \emptyset$
$Lab(e;\mathcal{I}) = \{e\}$
$Lab(\alpha;\mathcal{U}) = \{\alpha\} \cup Lab(\mathcal{U})$
$Lab(g \Rightarrow \mathcal{U}) = Lab(\mathcal{U})$
$Lab(\mathcal{U}_1 + \mathcal{U}_2) = Lab(\mathcal{U}_1) \cup Lab(\mathcal{U}_2)$
$Lab(P) = Lab(\mathcal{U})$ *where $P := \mathcal{U}$ is the definition for P* ∘

We can now formally define the test case associated with test case specification $(\mathcal{U}, U)$:

Definition 4.3 *The test case associated with test case specification $(\mathcal{U}, U)$ is the test case over $L = Lab(\mathcal{U}) \times U$ with* output states *the output test expressions in $\mathcal{D}_{(\mathcal{U},\, U)}$, the initial state being $\mathcal{U}$,* input states *the input test expressions in $\mathcal{D}_{(\mathcal{U},\, U)}$ and transition relation $\{(\mathcal{U}', \mu, \mathcal{E}) \mid \mathcal{U}', \mathcal{E} \in \mathcal{D}_{(\mathcal{U},\, U)}, \vdash \mathcal{U}' \xrightarrow{\mu} \mathcal{E}\}$* ∘

In the sequel we will omit set U in test case specification $(\mathcal{U}, U)$ when U is clear from the context. Moreover we will identify $(\mathcal{U}, U)$ with the test case it denotes.

The following is an example of a very simple test case over $I \times U$, where $I = \{r_1\}$ and $U = \{\Sigma, \{a_1\}, \{e_1\}, \{r_2\}\}$ which starts by sending r_1 to the implementation under test and then, if the latter responds with $\{a_1\}$ it reports success, otherwise it stops without reporting success:

$$r_1; \lambda x : U.\ \ x = \{a_1\} \Rightarrow \tau; \mathbf{W}; \delta \ + \ x \notin \{\{a_1\}\} \Rightarrow \delta$$

4.2. The Test Case Generation Algorithm

The definition of the test case generation algorithm TD is given in Fig.3. Note that TD is non-deterministic. Given $L = L_I \times L_U$ and $L' = L'_I \times L'_U$ and $\mathcal{F} \subseteq L^*$, after a *finite* number of recursive calls, TD returns a test case $\mathcal{U}$ in the test case language. The intuitive behaviour of the algorithm is rather simple; at each call, the algorithm generates a single test case. In particular, at each call, it may (non-deterministically) either generate the test which always reports success ($\tau; \mathbf{W}; \delta$), after which it terminates, or generate a test case as follows. An event e is (non-deterministically) chosen which belongs both to the input alphabet of the specification (L_I) and to that of the implementation (L'_I) and such that the set $\underline{\texttt{out}}\ \mathcal{F}\ e = \{\Gamma_1, \ldots, \Gamma_k\}$ is non-empty (notice that such an e exists when dealing with IOLTSs associated to UMLSCs, due to input-enabledness; see detailed proofs in [7]). Intuitively, $\Gamma_1, \ldots, \Gamma_k$ are the expected correct values for the output of the implementation under test as reaction to input e. Consequently, a test case is generated which first sends e to the implementation and then, if the output of the implementation does not match any of the expected values $\Gamma_1, \ldots, \Gamma_k$, it stops without reporting success, otherwise, assuming that the output of the implementation is Γ_j, it continues as $\mathcal{U}_j$. Notice that test case $\mathcal{U}_j$ is generated by a recursive call of the algorithm.

The set of all test cases which can be generated from $\mathcal{F}$, L and L' by repeated application of TD is denoted by $(TD_{L,L'}\ \mathcal{F})$.

For $L = L_I \times L_U$ and $L' = L'_I \times L'_U$ we define the following non-deterministic algorithm which, given set $\mathcal{F} \subseteq L^*$, after a *finite* number of recursive calls, returns a test case in the test language.

$TD_{L,L'}\ \mathcal{F} =$
Non-deterministically choose between options (1) and (2) below
1) generate "$\tau; \mathbf{W}; \delta$"
2) generate "$e; \lambda x : L'_U .\ x = \Gamma_1 \ \Rightarrow\ \mathcal{U}_1$
$+$
$\vdots$
$+$
$x = \Gamma_k \ \Rightarrow\ \mathcal{U}_k$
$+$
$x \notin \{\Gamma_1, \ldots, \Gamma_k\} \ \Rightarrow\ \delta$"

where:
e is non-deterministically chosen in $L_I \cap L'_I$
such that $\underline{\texttt{out}}\ \mathcal{F}\ e = \{\Gamma_1, \ldots, \Gamma_k\} \neq \emptyset$, and
$\mathcal{U}_j \in TD_{L,L'}\ (\mathcal{F}\ \underline{\texttt{after}}\ (e, \Gamma_j))$ for $j = 1, \ldots, k$

Figure 3. The Test Case Generation Algorithm

Notice that, by construction, test cases generated by TD have a tree-like structure; there is no looping possibility in their execution.
The following lemma easily follows from the definition of the algorithm, and the above remark, by observing that sets $\{\Gamma_1, \ldots, \Gamma_k\}$ are finite when $\mathcal{F}$ is the language of the IOLTS of a UMLSC:

Lemma 4.1 *For every UMLSC H with* **IOLTS** *H over i/o-pair set L, and i/o-pair set L', every test case $\mathcal{U} \in TD_{L,L'}\ (Lan(\mathbf{IOLTS}\ H))$ is finite.* ◇

Typically $Lan(\mathbf{IOLTS}\ H)$ is an infinite set. This does not affect the effectiveness of TD since, at each recursive step, it uses *only* the first elements of the traces in the set, postponing the use of their tails to the next recursive calls. Thus, proper lazy techniques can be used for the evaluation of $Lan(\mathbf{IOLTS}\ H)$. Notice also that the set of all test cases generated using $TD_{L,L'}$ on $Lan(\mathbf{IOLTS}\ H)$ is infinite. Each individual test case is however finite. As an immediate consequence of the above lemma and the fact that the test cases generated by the algorithm do not contain loops, we have that all computations involving test cases in $TD_{L,L'}\ (Lan(\mathbf{IOLTS}\ H))$ are finite.

The following theorem establishes completeness of the test case generation algorithm, when applied to (the language of) a specification **IOLTS** H:

Theorem 4.1 *For every UMLSC H with* **IOLTS** *H over i/o-pair set L, and i/o-pair set L', the test suite*

$$TD_{L,L'}\ (Lan(\mathbf{IOLTS}\ H))$$

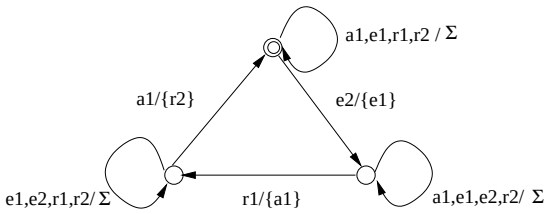

Figure 4. An implementation

$$\mathcal{U}_1 := e_2; \lambda x : U'.((x = \{e_1\} \Rightarrow \mathcal{U}_2) + (x \notin \{\{e_1\}\} \Rightarrow \delta))$$
$$\mathcal{U}_2 := r_1; \lambda x : U'.\ x = \{a_1\} \Rightarrow \mathcal{U}_3 + x = \Sigma \Rightarrow \mathcal{U}_4 + x \notin \{\{a_1\}, \Sigma\} \Rightarrow \delta$$
$$\mathcal{U}_3 := a_1; \lambda x : U'.((x = \{r_2\} \Rightarrow \mathcal{U}_4) + (x \notin \{\{r_2\}\} \Rightarrow \delta))$$
$$\mathcal{U}_4 := \tau; \mathbf{W}; \delta$$

Figure 5. A test case generated from the running example

is complete *w.r.t.* **IOLTS** H *and* $\sqsubseteq_{\mathtt{co}}$. $\diamond$

The above important result means that if a test case generated by the algorithm for a certain specification H reports a failure when running against an implementation, then we can be sure that the latter does not conform to the specification H; moreover, if an implementation does not conform to specification H, then a test case can be generated by the algorithm which will report failure when executed against such an implementation.

We close this section with an application of test cases derivation to our running example. Let us consider again the specification $\mathcal{S}$ of Fig. 1 and the implementation $\mathcal{S}'$ over $L'_I \times L'_U$ with $L'_I = \{a_1, e_1, e_2, r_1, r_2\}$ and $L'_U = \{\Sigma, \{a_1\}, \{e_1\}, \{r_2\}\}$ given in Fig.4, which is obviously an incomplete one. We can apply the algorithm in order to obtain, among others, the test case $\mathcal{U}_1$ shown in Fig. 5.

It is easy to see that $\mathcal{V}\,\mathcal{U}_1\,\mathcal{S}' =$ **pass**. On the other hand, $\mathcal{S}' \not\sqsubseteq_{\mathtt{co}} \mathcal{S}$, and this can be checked using the test case $\mathcal{U}_5$ shown in Fig. 6, which is also derived using the algorithm. Clearly $\mathcal{V}\,\mathcal{U}_5\mathcal{S}' =$ **fail**.

5. Conclusions and Future Work

In this paper we proposed a formal conformance testing relation for UMLSCs and an automatic test case generation algorithm. The algorithm has been proven complete, i.e. sound and exhaustive.

$$\mathcal{U}_5 := r_1; \lambda x : U'.((x = \{a_1\} \Rightarrow \mathcal{U}_4) + (x \notin \{\{a_1\}\} \Rightarrow \delta))$$

Figure 6. Another test case generated from the running example

The conformance relation and its test case generation algorithm are based on an operational semantics for UMLSCs which has been proven to fulfill major behavioral requirements stated in the official UML definition [8, 7]. As we already pointed out, the main contribution of the present paper is to set the theoretical basis for test case generation in a conformance testing setting. In order to use the test generation algorithm in practice proper *test selection* strategies are needed which will be a subject of our future work. Some work on test selection in a formal test derivation framework is already present in the literature (see, e.g. [4, 1, 6]), and in particular random test case selection seems to be a promising option. In fact it nicely fits with the structure of our algorithm; what is needed is to replace non-deterministic choices with random, coin-flipping, ones. Moreover, random test selection is receiving more and more attention due to the high coverage that it can provide, using efficient automated tools. Another promising line of research is the use of model-checking techniques for enhancing automatic test case generation, which we are currently investigating [5]. Tightly connected to the above research lines is the area of efficient implementation of test generation and selection algorithms. There are already tools available to that purpose, e.g. AutoFocus [19] and TGV/AGEDIS [20], and one of our next steps will be an investigation on the possibility of connecting our work with such tools.

In the present paper we made no assumption on how test cases are "implemented", i.e. on their actual presentation. They might be represented again as UMLSCs or as UML Sequence Diagrams or just as code in a proper programming language. This last possibility could allow for the implementation of test runs using proper automatic tools, to be integrated with the test case generation tools, which is our ultimate goal.

Another line of future research deals with the extension of the subset of UMLSCs we take into consideration. One necessary extension consists in allowing the use of UML specifications consisting of *collections* of UMLSCs interacting via queues, which brings to *distributed* testing. The use of a test language like the one proposed in the present paper, which is easy to extend in order to allow control communication between the experimenters to take place, greatly facilitates the task of specifying complex distributed test cases and to developing a suitable extension of testing theory to the distributed case.

Another useful extension is the introduction of data values and variables in UMLSCs. We have already a semantics definition for such an extension, fully developed in the context of the the PRIDE project [10]. Of course (infinite) data sets pose further problems in the test selection procedures.

References

[1] J. Alilovic-Curgus and S. Vuong. A metric based theory of test selection and coverage. In A. Danthine, G. Leduc, and W. P., editors, *Protocol Specification, Testing, and Verification, XII*, pages 289–304. IFIP WG 6.1, North-Holland Publishing Company, 1993.

[2] K. Bogdanov, M. Holcombe, and H. Singh. Automated test set generation for Statecharts. In D. Hutter, W. Stephan, P. Traverso, and M. Ullmann, editors, *Applied Formal Methods*, volume 1641 of *Lecture Notes in Computer Science*, pages 107–121. Springer-Verlag, 1998.

[3] B. Bosik and M. Ümit Uyar. Finite state machines based formal methods in protocol conformance testing: from theory to implementation. *Computer Networks and ISDN Systems. North-Holland*, 22:7–33, 1991.

[4] E. Brinksma, J. Tretmans, and L. Verhaard. A framework for test selection. In B. Jonsson, J. Parrow, and B. Pehrson, editors, *Protocol Specification, Testing, and Verification, XI*, pages 289–304. IFIP WG 6.1, North-Holland Publishing Company, 1991.

[5] A. Fantechi, S. Gnesi, and A. Maggiore. Enhancing test coverage by back-tracing model-checker counterexamples, 2004. (submitted for publication).

[6] L. Feijs, N. Goga, S. Mauw, and J. Tretmans. Test selection, trace distance and heuristics. In *IFIP 14th International Conference on Testing of Communicating Systems*, 2002.

[7] S. Gnesi, D. Latella, and M. Massink. Formal conformance testing UML Statechart Diagrams Behaviours: From theory to automatic test generation. Technical Report CNUCE-B04-2001-16, Consiglio Nazionale delle Ricerche, Istituto CNUCE, 2001. (Full version).

[8] S. Gnesi, D. Latella, and M. Massink. Modular semantics for a UML Statechart Diagrams kernel and its extension to Multicharts and Branching Time Model Checking. *The Journal of Logic and Algebraic Programming. Elsevier Science*, 51(1):43–75, 2002.

[9] M. Hennessy. *Algebraic Theory of Processes*. MIT Press, Boston, 1988.

[10] Intecs and CNR-CNUCE. PRIDE Definition of Changes in UML Notation. Technical Report PRIDE Deliverable 1.2, PRIDE, 02.

[11] D. Latella, I. Majzik, and M. Massink. Automatic verification of a behavioural subset of UML statechart diagrams using the SPIN model-checker. *Formal Aspects of Computing. The International Journal of Formal Methods. Springer-Verlag*, 11(6):637–664, 1999.

[12] D. Latella, I. Majzik, and M. Massink. Towards a formal operational semantics of UML statechart diagrams. In P. Ciancarini, A. Fantechi, and R. Gorrieri, editors, *IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Oriented Distributed Systems*, pages 331–347. Kluwer Academic Publishers, 1999. ISBN 0-7923-8429-6.

[13] D. Latella and M. Massink. A formal testing framework for UML Statechart Diagrams behaviours: From theory to automatic verification. In A. Jacobs, editor, *Sixth IEEE International High-Assurance Systems Engineering Symposium*, pages 11–22. IEEE Computer Society Press, 2001. ISBN0-7695-1275-5.

[14] D. Latella and M. Massink. On testing and conformance relations of UML Statechart Diagrams Behaviours. In P. G. Frankl, editor, *Proceedings of the ACM SIGSOFT 2002 International Symposium on Software Testing and Analysis*, pages 144–153. Association for Computing Machinery - ACM, 2002. ACM Software Engineering Notes 27(4), ISBN 1-58113-562-9.

[15] J. Le Traon, T. Jeron, J. Jezequel, S. Pickin, C. Jard, and A. Le Guennec. System test synthesis from UML models of distributed software. In D. Peled and M. Vardi, editors, *Formal Techniques for Networked and Distributed Systems - FORTE 2002*, volume 2529 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[16] D. Lee and M. Yannakakis. Principles and Methods of Testing Finite State Machines - A Survey. *Proceedings of the IEEE*, 84(8):1090–1123, 1996.

[17] M. Massink. *Functional Techniques in Concurrency*. PhD thesis, University of Nijmegen, Feb. 1996. ISBN 90-9008940-3.

[18] Object Management Group, Inc. *OMG Unified Modeling Language Specification - version 1.5*, 2003. http://www.omg.org/cgi-bin/doc?formal/03-03-01.

[19] A. Pretschner, O. Slotosh, H. Lötzbeyer, and E. Aiglstorfer. Model based testing for real: The Inhouse Card study. In *6th International ERCIM Workshop on Formal Methods for Industrial Critical Systems, Paris*, pages 79–94, 2001.

[20] The Agedis Project. The Agedis Home Page, 2003. `http://www.agedis.de/index.shtml`.

[21] J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software - Concepts and Tools*, 17(3):103–120, 1996.

[22] J. Tretmans. Testing concurrent systems: A formal approach. In J. Baeten and S. Mauw, editors, *Concur '99*, volume 1664 of *Lecture Notes in Computer Science*, pages 46–65. Springer-Verlag, 1999.