Testing UML Statecharts *

Mieke Massink Diego Latella† Stefania Gnesi

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

Abstract

Statechart diagrams provide a graphical notation to model dynamic aspects of system behaviour within the Unified Modelling Language (UML). In this paper we present a formal framework for notions related to testing and model based test generation for a behavioural subset of UML Statecharts (UMLSCs). This framework builds, on one hand, upon formal testing and conformance theory that has originally been developed in the context of process algebras and Labeled Transition Systems (LTSs), and, on the other hand, upon previous work of ours on formal semantics for UMLSCs. The paper covers the development of proper extensional testing preorders and equivalence for UMLSCs. An algorithm for testing equivalence verification is presented which is based on an intensional characterization of the testing relations. Testing equivalence verification is reduced to bisimulation equivalence verification. The paper also addresses the issue of conformance testing and presents a formal conformance relation together with a test case generation algorithm which is proved sound and exhaustive w.r.t. the conformance relation. We show results also on the formal relationship of the testing relations with the conformance one. The comprehensive and uniform approach presented in this paper sets the theoretical basis for UMLSCs testing frameworks and makes them available for practitioners in industry where the UML has become a *de facto* standard, in particular there where it is used for the development of compex concurrent systems.

Key words: UML Statecharts, Formal Testing Theories, Testing Equivalence Mechanical Verification, Formal Conformace Testing, Test Case Mechanical Generation, Formal Semantics

MSC: 68N30, 68Q10, 68Q60, 68Q85

1 Introduction

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 and software. 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) [34] 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 testing and model-based formal test case derivation using formal conformance testing.

*The work presented in the present paper has been partially funded by projects EU-IST IST-2001-32747 (AGILE) and MIUR/SP4

†Corresponding Author; phone: +39 0503152982

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, 17, 40, 41, 42]. The theory has been developed mainly in the context of process algebras and input/output transition systems.

In this paper we present a uniform, formal, approach to a *testing theory* and *equivalence* as well as *conformance testing* and *test case generation* for UML Statecharts[1] (UMLSCs, for short), based on previous work of ours presented in [25, 26, 16].

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 [17], where no distinction is made between input and output, nor for the input/output transition systems used in standard conformance testing theory [42]. In our approach we use transition systems labeled over i/o-pairs where a generic transition models a *step* of the associated statechart (*step*-transition), thus preserving the atomicity of input acquisition and output generation in a single step. The advantages of sush a semantic model choice are discussed in [16], where the interested reader is referred to.

In [25] a general testing theory for UMLSCs has been defined using a framework similar to that proposed in [31], which was in turn inspired by the work of Hennessy for traditional LTSs [17]. 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 behavior 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 [17] 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 [31], and later in [25] specifically for UMLSCs, assumption (i) has been replaced by modeling system interaction as input/output exchanges, but assumption (ii) remained unchanged. In particular, in [25], 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. The testing equivalence verification algorithm has originally been developed on the non-stuttering semantics and exploits an intensional characterization of testing preorders/equivalence. More specifically, Functional Acceptance Automata (FAAs), a compact representation of the LTSs associated to UMLSCs by their semantics, is defined using the intensional characterization of testing relations. It is shown that testing equivalence of UMLSCs coincides with bisimulation equivalence of (a variant of) their associated FAAs. The algorithm translates the LTSs associated to the UMLSCs into FAAs and checks their bisimulation equivalence after proper manipulation of their labels. It is partially based on results proposed in [4].

[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.

In [16] we proposed a formal conformance testing relation and a test case generation algorithm for *input enabled* labeled transition systems over i/o-pairs, i.e. 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 LTS has a specific loop-transition from the corresponding state to itself, labeled by that input and a special "stuttering" output-label.

Input enabled LTSs over i/o-pairs have been used as semantic model, which we call the "stuttering semantics", for a behavioral subset of UMLSCs [14], which can be seen as system specifications. Moreover, input enabled LTSs over i/o-pairs are also suitable for modeling implementations of systems specified by such diagrams. Modeling implementations as input enabled LTSs is common practice in the context of formal conformance testing —see e.g. [42]. The conformance relation we defined in [14] is similar to the one of Tretmans [42], with adaptations which take care of our semantic framework for UMLSCs.

The test case generation algorithm we present is both *exhaustive* and *sound* with respect to the conformance relation. Exhaustiveness ensures that if an implementation passes all test cases generated by the algorithm from a given specification, then it conforms to the specification. Conversely, soundness ensures that if an implementation conforms to a specification, then it passes all test cases generated by the algorithm from such a specification. The testing equivalence verification algorithm naturally extends to the case of stuttering semantics.

The 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, we define *MAY* and *MUST* preorders also for the stuttering semantics and we provide a formal comparison between the Hennessy-like, non-stuttering semantics [25, 17], and the stuttering semantics w.r.t. testing and conformance ordering relations; we show 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. $A \underset{\sim}{\sqsubset}_{MAY} B$ and $B \underset{\sim}{\sqsubset}_{MUST} A$) in the non-stuttering semantics, *but not vice-versa*. This shows that the Hennessy-like, non-stuttering, semantics [25, 17] 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 are proved: *in the stuttering semantics*, the conformance relation essentially coincides with the *MAY* preorder, and is strictly weaker than the testing preorder. 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. The above results have been originally presented in [26] and in the present paper we include all related proofs.

As an additional result our work, we also defined a specific test case language which we use uniformly in the present paper both for what concerns the testing preorders/equivalence and for automatic test case generation as well as what concerns conformance.

The present paper is organized as follows. Sect.2 discusses the relationship of the work presented in the present paper with the literature. In Sect. 3 the major background notions, necessary for the development of the testing and conformance theories are recalled. Sect. 4 addresses the testing preorders and the equivalence verification algorithm. Sect. 5 addresses conformance testing and the test case generation algorithm. Sect. 6 studies the relationships between the two views at the semantics of UMLSCs presented and used in Sections 4 and 5—namely the "non-stuttering" and the "stuttering" semantics—and compares the testing and the conformance relations. Technical details and results on the dynamic semantics of UMLSCs on which our work is based, and in particular their "core semantics" are given in Appendix A. Appendix B contains the detailed formal proofs of all results presented in this paper.

2 Related Work

As briefly mentioned in Sect. 1, the basic work on formal theories for testing, mainly in the context of Process Algebra and LTSs, has been proposed by De Nicola and Hennessy (see e.g. [9, 17]). Tretmans addressed more the issues related to conformance testing in a formal, LTS-based, framework [40, 41, 42].

The results addressed in the present paper have been originally proposed in [25, 26, 16], although in isolation, while in the present paper they are dealt with in a uniform framework and notation. Moreover all proofs, some of which were omitted in the above mentioned papers, are provided in the present paper.

Our LTSs labeled 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 [28]. Many such proposals deal with test case generation but mainly in the context of *deterministic* machines, as, e.g., in the seminal work of Chow [38], or in [11], where practical applicablity of model-based test case generation is addressed. In some proposals, like the one in [5], further restrictions on the machines are introduced, requiring e.g. that they must be strongly connected. Non-determinism in the context of conformance testing FSMs is addressed in [35], where specifications may be non-deterministic, while implementations are required to be deterministic. Specifications and implementations are required to share the same input alphabet. Moreover, only so called *observable* non-deterministic FSMs are considered. Observable non-deterministic FSMs are FSMs which cannot produce the same output on the same input while moving to different next states, i.e. if they move from the current state to different next states they must also produce different outputs. Specification FSMs are not required to be completely specified, i.e. there can be states which have no outgoing transition for some input event (the notion of complete specification is the FSMs analogous of input-enabled in the context of LTSs). A conformance relation on FSMs, called *reduction*, is given which is very similar to the conformance relation we use in the present paper, and a finite test suites generation algorithm is proposed which is complete in the class of all implementation with a given upper bound on the number of states (completeness in the context of FSMs corresponds to eshaustiveness in the context of LTSs). In [18] the methods are extended in such a way that *adaptive* testing is possible, i.e. information is gathered from the output of an implementation under test that can be used for guiding future testing. Neither [35] nor [18] adresses the issue of linking the testing methodology and algorithms they propose to a general framework where the very notion of testing computing devices, its formalization and the equivalence it induces are addressed, as e.g. in [9, 17]. Moreover, we think that the restriction to observable FSMs is a rather strong one, if non-deterministic behaviour is to be addressed. In fact, although for each (completely specified) non-observable FSM there exists an equivalent observable one [18], such an equivalence takes into account only the language defined by the machine and not its deadlock properties, which can be of major interest in specific contexts, possibly different from conformance testing (e.g. non-stuttering semantics of UMLSCs). Furthermore UMLSCs can easily violate the observability constraint. Restricting testing theories to deterministic implementations seems also a rather strong limitation, especially in the context of distributed or concurrent implementations. In such a context, non-determinism arising from concurrency cannot be avoided and, in fact, non-determinism is a key notion in the area of formal approaches to system modeling and verification and it is a central notion in traditional concurrency and testing theories for LTSs [19, 33, 9, 17]. Consequently we use generic LTSs over i/o-pairs without any limitation on the form of non-determinism they may exhibit. The restriction to input-enabled LTSs, when we address conformance, adequately models the UMLSCs stuttering phenomenon. Furthermore, the link we provide to testing equivalence, rather than language equivalence, and in particular its definition in terms of experimenters, in the sense of [9, 17], brings in—without renouncing to a solid mathematical framework—a strong intuitive support which is sometimes missing in the above mentioned works on FSMs testing.

In [21] algorithms for test case generation from UML statecharts are presented which cover both control flow issues and data flow ones. As far as flow control is concerned, statecharts are mapped to (extended) FSMs. The semantic framework on which the presented work is based

is a flat one, i.e. the hierarchical structure of UMLSCs is not exploited in the definition of the operational semantics. Moreover, the model presented does not take transition priorities into consideration. The relationship with general testing theories for state/transition structures is not addressed.

Further related work on automatic test generation based on UMLSCs has been developed in the context of the Agedis project [39, 8]. 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 [36] emphasis is put primarily on support tool implementation. The semantics of UMLSCs is defined by means of the tool umlout—which generates LTSs with inputs and outputs events, in the style of [42]. In particular, no formal definition of such semantics is given in [36]. We already addressed the issues related to the use of LTSs with separate input and output events as a model for UMLSCs semantics.

Other approaches to automatic test generation include [37] 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. [3], which semantically differ considerably from UMLSCs (e.g., a different priority schema as well as a different semantics for the input queues are used).

3 Basic Notions

In this section we summarize the definitions concerning LTSs, with particular reference to LTSs over input/output-pairs (i/o-pairs, for short), hierarchical automata, experimenters, experimental systems and their computations, which form the basis for the formal semantics of UMLSCs and related testing and conformance notions as presented in [15, 25, 26, 16].

The definition of a sound "basic" kernel of a notation, to be extended only after its main features have been investigated, has already proved 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. [23] for a deeper discussion on such "basic-notation-first" and "semantics-first" versus "full-notation-first" issue. In line with this approach, in the present paper, we consider a subset of UMLSCs, which includes all the interesting conceptual issues related to concurrency in dynamic behavior—like sequentialization, non-determinism and parallelism—as well as UMLSCs specific issues—like state refinement, transition priorities, interlevel/join/fork transitions. 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. We refer to [27] for object-based extensions of our basic model which include, among others, object management, e.g. object creation/destruction.

In Sect. 3.1 basic notions related to Labeled Transition Systems are briefly recalled; Hierarchical Automata are shortly described in Sect. 3.2 while Sect. 3.3 recalls the major notions related to testing theories.

3.1 Labeled Transition Systems

The notion of Labeled Transition System (LTS) is central in the present paper:

Definition 3.1 (LTS)
A Labeled Transition System (LTS) $\mathcal{S}$ *is a tuple* $(S, s_{in}, L, \rightarrow)$ *where* S *is the* set of states *with*

$s_{in} \in S$ *being the* initial *state,* L *is the* set of (transition) labels *and* $\rightarrow \subseteq S \times L \times S$ *is the* transition relation *of the LTS.* ∘

For $(s, l, s') \in \rightarrow$ we write $s \xrightarrow{l} s'$. The notation $s \xrightarrow{l}$ will be a shorthand for $\exists s'.\ s \xrightarrow{l} s'$. Some standard definitions are given below [2].

Definition 3.2
For LTS $\mathcal{S} = (S, s_{in}, L, \rightarrow)$, $s, s', s'' \in S$, $l \in L$, $\gamma \in L^*$, $\omega \in L^\infty$

- *The transition relation* $\xrightarrow{\gamma}$ *over finite sequences is defined in the obvious way: (a)* $s \xrightarrow{\epsilon} s$ *and (b) if* $s \xrightarrow{\gamma} s'$ *and* $s' \xrightarrow{l} s''$, *then* $s \xrightarrow{\gamma l} s''$;
- *By* $s \xrightarrow{\omega}$ *we mean that there exists an infinite sequence* $s_0 s_1 s_2 \ldots$ *of states in* S, *with* $s = s_0, \omega = l_0 l_1 \ldots$, *such that for all* $n \geq 0$ *we have* $s_n \xrightarrow{l_n} s_{n+1}$
- *The* language *of* $\mathcal{S}$ *is the set of all its finite* traces: $\mathsf{lan}\,\mathcal{S} =_{df} \{\gamma \in L^* \mid \exists s'.\ s_{in} \xrightarrow{\gamma} s'\}$;
- *The labels of* $\mathcal{S}$ after γ *is the set* $\mathsf{S}\,\mathcal{S}\,\gamma =_{df} \{l \in L \mid \exists s, s' \in S.\ s_{in} \xrightarrow{\gamma} s \wedge s \xrightarrow{l} s'\}$
- *The* acceptance sets *of* $\mathcal{S}$ after γ *is the set* $\mathsf{AS}\,\mathcal{S}\,\gamma =_{df} \{\mathsf{S}\,s\,\epsilon \mid \exists s.\ s_{in} \xrightarrow{\gamma} s\}$ ∘

Notice that in the definition of acceptance sets we have treated state $s \in S$ of LTS $\mathcal{S}$ as a LTS in turn. The set S_s of states of such LTS contains all and only those elements of S which are reachable from s via $\rightarrow$ (i.e. $S_s =_{df} \{s' \in S \mid \exists \gamma.\ s \xrightarrow{\gamma} s'\}$), the initial state being s; the transition relation of the LTS is $\rightarrow \cap (S_s \times S_s)$. We will often treat states of LTSs as LTSs in turn, as above.

In the rest of 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; we will refer to such LTSs as LTSs *over* $L_I \times L_U$.

3.2 Hierarchical Automata

As briefly mentioned in Sect. 1 we use hierarchical automata (HAs) [32] as an abstract syntax for UMLSCs. HAs for UMLSCs have been introduced in [24, 15]. The relevant definitions concerning HAs, like their dynamic semantics, are recalled in Appendix A. In this section we recall, informally, only the main notions which are necessary for the understanding of the paper.

Let us consider, as a small example, the UMLSC of Fig.1 and its corresponding HA in Fig. 2. Roughly speaking, each OR-state of the UMLSC is mapped into a sequential automaton of the HA while basic states and AND-states are mapped into states of the sequential automaton corresponding to the OR-state immediately containing them. Moreover, a refinement function maps each state in the HA corresponding to an AND-state into the set of the sequential automata corresponding to its component OR-states. In the example OR-states $s0, s4, s5$ and $s7$ are mapped to sequential automata $A0, A1, A2$ and $A3$, while state $s1$ of $A0$, corresponding to AND-state $s1$ of the UMLSC, is refined into $\{A1, A2\}$. Non-interlevel transitions are represented in the obvious way: for instance transition $t8$ of the HA represents the transition from state $s8$ to state $s9$ of the UMLSC. The labels of transitions are collected in Table 1; for example the *trigger event* of $t8$, namely $EV\ t8$, is $e2$ while its associated *output event*, namely $AC\ t8$ is $e1$. An interlevel transition is represented as a transition t departing from (the HA state corresponding to) its highest source and pointing to (the HA state corresponding to) its highest target. The set of the other sources, resp., targets, are recorded in the *source restriction* - $SR\ t$, resp. *target determinator* $TD\ t$, of t. So, for instance, $SR\ t1 = \{s6\}$ means that a necessary condition for $t1$ to be enabled is that the current state configuration contains not only $s1$ (the source of $t1$), but *also* $s6$. Similarly, when

[2] 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 (resp. infinite) sequences over X will be denoted by X^* (resp. X^∞); for $x \in X$ we let x denote also the sequence in X^* consisting of the single element x, while for $\gamma, \gamma' \in X^*$ we let the juxtaposition $\gamma\gamma'$ of γ with γ' denote their concatenation. Concatenation is extended to infinite sequences in with $\gamma\gamma' = \gamma$ when γ is infinite, and $\gamma\gamma'$ defined in the usual way otherwise.

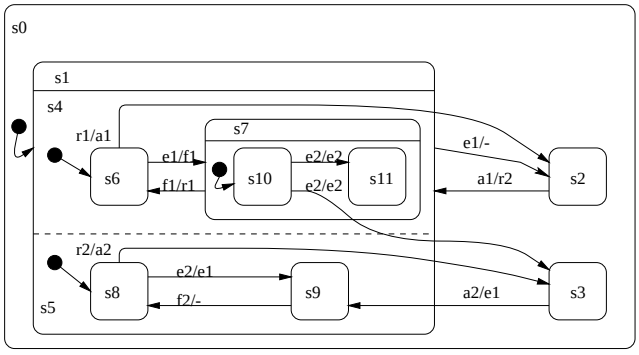

Figure 1: A sample UMLSC

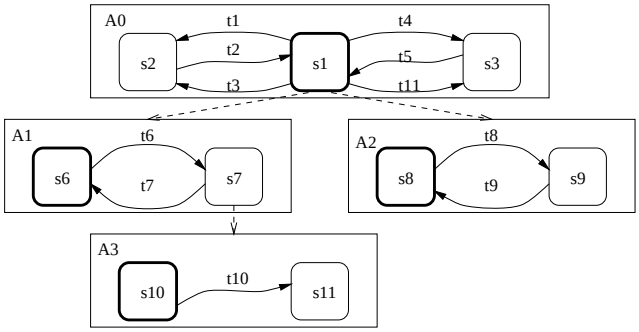

Figure 2: The HA representing the sample UMLSC of Fig. 1

t	$t1$	$t2$	$t3$	$t4$	$t5$	$t6$	$t7$	$t8$	$t9$	$t10$	$t11$
$SR\ t$	$\{s6\}$	$\emptyset$	$\emptyset$	$\{s8\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\{s10\}$
$EV\ t$	$r1$	$a1$	$e1$	$r2$	$a2$	$e1$	$f1$	$e2$	$f2$	$e2$	$e2$
$AC\ t$	$a1$	$r2$	ϵ	$a2$	$e1$	$f1$	$r1$	$e1$	ϵ	$e2$	$e2$
$TD\ t$	$\emptyset$	$\{s6, s8\}$	$\emptyset$	$\emptyset$	$\{s6, s9\}$	$\{s10\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

Table 1: Transition Labels for the HA of Fig. 2

firing $t2$ the new state configuration will contain $s6$ and $s8$, besides $s1$. Finally, each transition has a guard $G\ t$, not shown in this example.

Summarizing, basically a HA $H = (F, E, \rho)$ is composed of a finite (non-empty) collection F of sequential automata related by a *refinement function* ρ which imposes on H the hierarchical state nesting-structure of the associated statechart: ρ maps every state s of each automaton in F into a (possibly empty) set of elements of F which refine s. The automata in F are finite, i.e. they have a finite set of states and a finite number of transitions. E is the finite set of events labeling the transitions of the elements of F. Inter-level transitions are encoded by means of proper annotations in transition labels. Global states of H, called state *configurations* are *sets* of states of the automata in F which respect the tree-like structure imposed by ρ. We let Conf_H denote the set of all configurations of H and $\mathcal{C}_{\mathrm{in}}$ its initial configuration, i.e. the configuration composed only of initial states of automata in F.

An issue which deserves to be briefly addressed here is the way in which we deal with the so called *input-queue* of UMLSCs, i.e. their "external environment". In the standard definition of UML statecharts semantics [34], a *scheduler* is in charge of selecting an event from the input-queue of an object, feeding it into the associated state-machine, and letting such a machine produce a step transition. Such a step transition corresponds to the firing of a maximal set of enabled non-conflicting transitions of the statechart associated to the object, provided that certain transition priority constraints are not violated. After such transitions are fired and when the execution of all the actions labeling them is completed, the step itself is completed and the scheduler can choose another event from a queue and start the next cycle. While in classical statecharts the external environment is modeled by a set, in the definition of UML statecharts, the nature of the input-queue of a statechart is not specified; in particular, the management policy of such a queue is not defined. In our overall approach to UMLSCs semantics definition, we choose *not* to fix any particular semantics, such as set, or multi-set or FIFO-queue etc., but to model the input queue in a policy-independent way, freely using a notion of abstract data types. In the following we assume that for set D, Θ_D denotes the class of all structures of a certain kind (like FIFO queues, or multi-sets, or sets) over D and we assume to have basic operations for manipulating such structures. In particular, in the present paper, we let $\mathsf{Add}\ d\ \mathcal{D}$ denote the structure obtained inserting element d in structure $\mathcal{D}$ and the predicate $(\mathsf{Sel}\ \mathcal{D}\ d\ \mathcal{D}')$ states that $\mathcal{D}'$ is the structure resulting from selecting d from $\mathcal{D}$; of course, the selection policy depends on the choice for the particular semantics. Similarly, $(\mathsf{Join}\ \mathcal{D}\ \mathcal{D}')$ denotes the structure obtained by merging $\mathcal{D}$ with $\mathcal{D}'$. We assume that if $\mathcal{D}$ is the empty structure, denoted by $\langle\rangle$, then $(\mathsf{Sel}\ \mathcal{D}\ d\ \mathcal{D}')$ yields FALSE for all d and $\mathcal{D}'$. We shall often speak of the *input queue*, or simply *queue*, by that meaning a structure in Θ_D, abstracting from the particular choice for the semantics of Θ_D.

The operational semantics of HA H characterizes the relation $(\mathcal{C}, \mathcal{E}) \xrightarrow{\mathcal{L}} (\mathcal{C}', \mathcal{E}')$ between *statuses* and transitions fired during a step. A status is a pair $(\mathcal{C}, \mathcal{E})$ where $\mathcal{C}$ is the current configuration and $\mathcal{E}$ is the current input queue. $(\mathcal{C}, \mathcal{E}) \xrightarrow{\mathcal{L}} (\mathcal{C}', \mathcal{E}')$ means that a step-transition of H can be performed in the current status $(\mathcal{C}, \mathcal{E})$ by firing the transitions belonging to set $\mathcal{L}$ and reaching the new status $(\mathcal{C}', \mathcal{E}')$. The new configuration (resp. input-queue) of H after the step will be $\mathcal{C}'$ (resp. $\mathcal{E}'$). The definition of the step-transition relation is given below:

Definition 3.3 (Transition Deduction System)

$$\frac{(\mathsf{Sel}\ \mathcal{E}\ e\ \mathcal{E}'') \qquad H \uparrow \emptyset :: (\mathcal{C}, \{e\}) \xrightarrow{\mathcal{L}} (\mathcal{C}', \mathcal{E}')}{(\mathcal{C}, \mathcal{E}) \xrightarrow{\mathcal{L}} (\mathcal{C}', (\mathsf{Join}\ \mathcal{E}''\mathcal{E}'))}$$

The above definition makes use of the so called *Core Semantics*, i.e. the relation

$$A \uparrow P :: (\mathcal{C}, \mathcal{E}) \xrightarrow{\mathcal{L}} (\mathcal{C}', \mathcal{E}').$$

The role of the Core Semantics is the characterization of the set $\mathcal{L}$ of transitions to be fired, their related output-events, $\mathcal{E}'$ and the resulting configuration $\mathcal{C}'$, when HA A is in status $(\mathcal{C}, \mathcal{E})$, under specific constraints P related to transition priorities. All issues of (event) ordering, concurrency,

and non-determinism within single statecharts are dealt with by the Core Semantics. Although essential for the definition of the formal semantics, all the above issues are concerned with an intensional view of the statechart behavior, thus they are technically quite orthogonal to the testing and conformance issues which we address in the present paper and which are intrinsically extensional, and, therefore, the details of the Core Semantics are given in Appendix A.

3.3 Definitions Related to Testing

In order to model how test cases are performed over systems represented by LTSs over $L_I \times L_U$ we first of all need to formalize the notion of experimenters. An *experimenter* is similar to a transition system where some states—namely the *input* states, i.e. states in which the experimenter is supposed to get some output generated by the system to experiment with—are actually *total functions* from output labels L_U to output states. Totality guarantees that *any* output of the system under test is accepted by the experimenter, in that state. On the basis of the particular value received, the experimenter will move to the next output state. Output states are those in which the experimenter can produce specific events to be delivered to the system, or silently move, via τ, to other (output) states, or produce the special action **W** by which it reports that the experiment has been successful. As it clearly appears from the above description, an experimenter can be non-deterministic. The formal definition of experimenters follows:

Definition 3.4 (Experimenter)
An Experimenter $\mathcal{T}$ *over* $L_I \times L_U$ *is a tuple* $(T_U, \upsilon_{in}, T_I, L_I, L_U, \rightarrow)$ *where* T_U *is the set of* output *states, with* $\upsilon_{in} \in T_U$ *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* $\rightarrow \subseteq (T_U \times L_I \times T_I) \cup (T_U \times \{\tau, \mathbf{W}\} \times T_U)$ *is the* transition relation*, with* $(L_I \cup L_U) \cap \{\tau, \mathbf{W}\} = \emptyset$. ∘

We say that an experimenter $\mathcal{T}$ is *finite* whenever T_U, T_I and $\rightarrow$ are finite sets.

It is 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 [25].

Experimentation of a LTS over $L_I \times L_U$ against an experimenter $\mathcal{T}$ is modeled by the *Experimental System* they characterize:

Definition 3.5 (Experimental System)
For LTS $\mathcal{S} = (S, s_{in}, L_I \times L_U, \rightarrow)$ *and experimenter* $\mathcal{T} = (T_U, \upsilon_{in}, T_I, L_I, L_U, \rightarrow)$ *, the experimental system* $< \mathcal{T}, \mathcal{S} >$ *is the transition system* $(T_U \times S, (\upsilon_{in}, s_{in}), \rightsquigarrow)$*. 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* $s, s' \in S$*,* $\upsilon, \upsilon' \in T_U$*,* $\iota \in T_I$ *and for* $((\upsilon, s), (\upsilon', s')) \in \rightsquigarrow$ *we write* $\upsilon \mid\mid s \rightsquigarrow \upsilon' \mid\mid s'$

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

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 experimenters is defined by the second rule. Single experiments are modeled by *computations*:

Definition 3.6 (Computations)
A computation *of experimental system* $< \mathcal{T}, \mathcal{S} >$ *is a sequence of the form:*

$$\upsilon_0 \mid\mid s_0 \rightsquigarrow \upsilon_1 \mid\mid s_1 \rightsquigarrow \upsilon_2 \mid\mid s_2 \rightsquigarrow \ldots \upsilon_k \mid\mid s_k \rightsquigarrow \ldots$$

which is maximal*, i.e. either it is infinite or it is finite with terminal element* $\upsilon_n \mid\mid s_n$ *which has the property that* $\upsilon_n \mid\mid s_n \rightsquigarrow \upsilon' \mid\mid s'$ *for no pair* υ'*,* s'*.* υ_0 *and* s_0 *are the initial states* υ_{in} *and* s_{in} *of* $\mathcal{T}$ *and* $\mathcal{S}$*.*
A computation is successful *iff* $\upsilon_k \in$ Success *for some* $k \geq 0$*, otherwise it is* unsuccessful. ∘

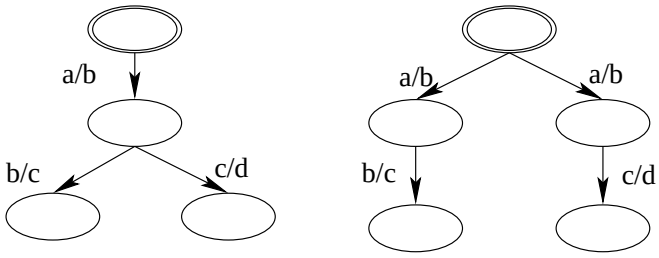

Figure 3: Two language-equivalent but not testing equivalent LTSs

We let $\mathrm{Comp}(\mathcal{T},\mathcal{S})$ denote the set of all computations of $< \mathcal{T},\mathcal{S} >$. From the definition of experimental system we know that every computation $\eta \in \mathrm{Comp}(\mathcal{T},\mathcal{S})$ gives rise to a transition $s_{\mathrm{in}} \xrightarrow{\gamma}$ over finite or infinite sequence γ on the side of the LTS. In this case, we say that η *runs over* γ.

Definition 3.7 (Testing Equivalence)
For experimental system $< \mathcal{T},\mathcal{S} >$ *we let the set* $\mathrm{Result}(\mathcal{T},\mathcal{S}) \subseteq \{\top,\bot\}$ *be defined as follows:*

$\top \in \mathrm{Result}(\mathcal{T},\mathcal{S})$ *iff* $\mathrm{Comp}(\mathcal{T},\mathcal{S})$ *contains a successful computation*

$\bot \in \mathrm{Result}(\mathcal{T},\mathcal{S})$ *iff* $\mathrm{Comp}(\mathcal{T},\mathcal{S})$ *contains an unsuccessful computation*

We say $\mathcal{S}$ *and* $\mathcal{S}'$ *are* testing equivalent*, written* $\mathcal{S} \sim \mathcal{S}'$ *iff for all experimenters* $\mathcal{T}$ $\mathrm{Result}(\mathcal{T},\mathcal{S}) = \mathrm{Result}(\mathcal{T},\mathcal{S}')$ ∘

Intuitively, the above definition establishes that we can consider two systems $\mathcal{S}$ and $\mathcal{S}'$ equivalent if and only if no experimenter $\mathcal{T}$ can distinguish them on the basis of the fact that its computations have reported success or not. This notion completely captures the idea of equivalence based on the *externally observable* behavior of the two systems. Notice that testing equivalence for LTSs is strictly stronger than language equivalence for FSMs, in the sense that, as we shall see in Sect. 4, testing equivalent LTSs characterize the same language, while the converse does not hold. This can be easily seen using the exammple of Fig. 3. We leave it to the reader to prove that the two LTSs in the figure are not testing equivalent but they have the same language.

In the rest of the paper it will often be useful to define specific experimenters. To that purpose we define below a language of experimenter specifications. Let IE and OS be countable sets such that $(IE \cup OS) \cap \{\tau, \mathbf{W}\} = \emptyset$—we call IE the set of *events* and OS the set of *possible outputs*. The abstract syntax of *output* experimenter expressions $\mathcal{U}$—resp. *input* experimenter 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 (resp. x) is an experimenter (resp. input) variable, X is a parameter of type ΘD, g is a boolean expression of the form " $x = u$" or "$x \neq u$", for $u \in OS$, 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.

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

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

An experimenter specification consists of a pair $(\mathcal{U},U)$ where $\mathcal{U}$ is an output experimenter expression and $U \subseteq OS$. We will require that no input variable occurs free in $\mathcal{U}$ and that a unique experimenter definition $P(X) \triangleq \mathcal{U}'$ is associated with any experimenter variable P occurring in $\mathcal{U}$ in the context where the experimenter specification is used. Moreover, all input experimenter sub-expressions of $\mathcal{U}$ must use the same set U in their defining lambda-expression.

Let us briefly describe the semantics of the language of experimenter specifications. The experimenter δ performs no action. Expression $e;\mathcal{I}$ offers event e and then behaves like $\mathcal{I}$ which is an input experimenter expression, namely a function. Such a function will be applied to the output produced by the system to experiment with in an experimental system (see Def.3.5). The

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

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

$$\frac{\mathcal{U} \xrightarrow{\mu} \mathcal{O}}{TRUE \Rightarrow \mathcal{U} \xrightarrow{\mu} \mathcal{O}} \qquad \frac{P(X) \triangleq \mathcal{U}, \mathcal{U}[\mathcal{D}/X] \xrightarrow{\mu} \mathcal{O}}{P(\mathcal{D}) \xrightarrow{\mu} \mathcal{O}}$$

Figure 4: Experimenter Expressions Operational Semantics

specific (output) state resulting from the application is obtained according to the semantics of *input* experimenter expressions, as given by the following rewrite rule for function application:

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

where $\mathcal{U}[u/x]$ denotes $\mathcal{U}$ where all free occurrences of x are simultaneously replaced by u. Notice that the above rule is a simplification of β-reduction of the lambda-calculus since u 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 $\mathbf{W}$. 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 $P \triangleq \mathcal{U}$ is the definition for P, P behaves like $\mathcal{U}$. If the optional parameter X is used in the definition of P, then $P(\mathcal{D})$ behaves as $\mathcal{U}[\mathcal{D}/X]$ where again we use substitution. In the following, we will assume $\mathcal{D}$ be an element of L, or L^*, 2^L, or Θ_D, the particular case being clear from the context. Finally, we will often use an extended form of parametrized experimenter $P(X_1,\ldots,X_k)$ with the obvious meaning.

The operational semantics of experimenter specifications is given in a similar way as for process algebra, by means of the Structural Operational Semantics rules of Fig. 4 where $\mu \in IE \cup \{\tau, \mathbf{W}\}$ and $\mathcal{O}$ stands both for output and for input experimenter expressions.

In order to formally derive the experimenter denoted by an experimenter specification we first need a couple of auxiliary definitions where by $\vdash \mathcal{U} \xrightarrow{\mu} \mathcal{O}$ we mean that $\mathcal{U} \xrightarrow{\mu} \mathcal{O}$ can be deduced using the rules of Fig. 4.

Definition 3.8 (Derivatives)
The derivatives of experimenter specification $(\mathcal{U},U)$ is the smallest set $\mathcal{D}_{(\mathcal{U},\,U)}$ of experimenter expressions which satisfies the following three conditions:

1. *$\mathcal{U} \in \mathcal{D}_{(\mathcal{U},\,U)}$;*
2. *if output experimenter expression $\mathcal{U}'$ is in $\mathcal{D}_{(\mathcal{U},\,U)}$ and $\vdash \mathcal{U}' \xrightarrow{\mu} \mathcal{O}$ then also $\mathcal{O}$ is in $\mathcal{D}_{(\mathcal{U},\,U)}$;*
3. *if input experimenter 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 3.9 (Labels)
The labels of experimenter specification $(\mathcal{U},U)$, $Lab(\mathcal{U})$ is defined recursively as follows:
$Lab(\delta) =_{df} \emptyset$
$Lab(e;\mathcal{I}) =_{df} \{e\}$
$Lab(\alpha;\mathcal{U}) =_{df} \{\alpha\} \cup Lab(\mathcal{U})$
$Lab(g \Rightarrow \mathcal{U}) =_{df} Lab(\mathcal{U})$
$Lab(\mathcal{U}_1+\mathcal{U}_2) =_{df} Lab(\mathcal{U}_1) \cup Lab(\mathcal{U}_2)$
$Lab(P(\mathcal{D})) =_{df} Lab(\mathcal{U}[\mathcal{D}/X])$ *where $P(X) \triangleq \mathcal{U}$ is the definition for P* ∘

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

Definition 3.10
The experimenter associated with experimenter specification $(\mathcal{U}, U)$ *is the experimenter over* $L = Lab(\mathcal{U}) \times U$ *with* output states *the output experimenter expressions in* $\mathcal{D}_{(\mathcal{U},\, U)}$*, the initial state being* $\mathcal{U}$*,* input states *the input experimenter expressions in* $\mathcal{D}_{(\mathcal{U},\, U)}$ *and transition relation* $\{(\mathcal{U}', \mu, \mathcal{O}) \mid \mathcal{U}', \mathcal{O} \in \mathcal{D}_{(\mathcal{U},\, U)}, \vdash \mathcal{U}' \xrightarrow{\mu} \mathcal{O}\}$ ∘

In the sequel we will omit set U in experimenter specification $(\mathcal{U}, U)$ when U is clear from the context. Moreover we will identify $(\mathcal{U}, U)$ with the experimenter it denotes. The following is an example of a very simple experimenter over $I \times U$, where $I = \{r_1\}$ and $U = \{\{a_1\}, \{e_1\}, \{r_2\}\}$ which starts by sending r_1 to the system to experiment with and then, if the latter responds with $\{a_1\}$ it reports success, otherwise it stops without reporting success:

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

4 Testing Relations

In this section we develop a general testing theory for UMLSCs, originally proposed in [25], using a framework similar to that proposed in [31], which was in turn inspired by the work of Hennessy for traditional LTSs [17]. The general approach is based on the well known notions of *MAY* and *MUST* preorders and related equivalences. The main semantic assumptions in [17] 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 [31], and later in [25] specifically for UMLSCs, assumption (i) has been replaced by modeling system interaction as input/output exchanges, but assumption (ii) remains unchanged. In the following we shall first recall the semantic interpretation of HAs as proposed in [25], which we call the *non-stuttering* semantics for HAs, for reasons which will be clear in the sequel, and we show its formal relation with the original semantics for HAs recalled in Sect. 3.2. In the rest of this section we will develop the above mentioned testing theory based on the non-stuttering semantics.

The non-stuttering semantics is recalled in Sect. 4.1 while its relationship with the original semantics of UMLSCs proposed in [15] is addressed in Sect. 4.2. In Sect. 4.3 relevant testing preorders are given which bring to the notion of testing equivalence. In Sect.4.4 an alternative, intensional, characterization of such preorders/equivalence is addressed which serves as a link to a finite representation for (the LTSs denoted by) UMLSCs used for automatic verification in Sect. 4.6.

4.1 Non-stuttering Semantics

The non-stuttering semantics associates a LTS to each HA.

Definition 4.1 (Non-stuttering semantics)
The non-stuttering semantics of an HA $H = (F, E, \rho)$ *is the LTS over* $E \times \Theta_E$ $\mathcal{LTS}(H) =_{df}$ $(\mathrm{Conf}_H, \mathcal{C}_{in}, \rightarrow)$ *where (i)* Conf_H *is the set of configurations, (ii)* $\mathcal{C}_{in} \in \mathrm{Conf}_H$ *is the initial configuration, (iii)* $\rightarrow \subseteq \mathrm{Conf}_H \times (E \times \Theta_E) \times \mathrm{Conf}_H$ *is the* step*-transition relation defined below.* ∘

We write $\mathcal{C} \xrightarrow{e/\mathcal{E}} \mathcal{C}'$ for $(\mathcal{C}, (e, \mathcal{E}), \mathcal{C}') \in \rightarrow$. Any such transition denotes the result of firing a maximal set of non-conflicting transitions of the sequential automata of H which respect priorities when the state machine associated to H is given event e as an input. $\mathcal{E}$ is the collection of output events generated by the transition which have been fired. Relation $\rightarrow$ is the smallest relation which satisfies the rule below:

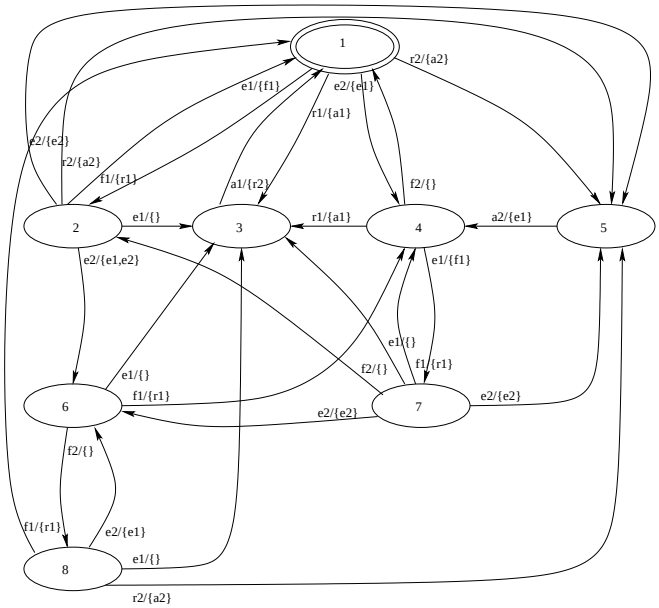

Figure 5: LTS(H), for H of Fig. 2

Definition 4.2 (Non-stuttering Transition Deduction System)

$$\frac{e \in E,\ \mathcal{L} \neq \emptyset,\ H \uparrow \emptyset :: (\mathcal{C}, \{e\}) \xrightarrow{\mathcal{L}} (\mathcal{C}', \mathcal{E})}{\mathcal{C} \xrightarrow{e/\mathcal{E}} \mathcal{C}'}$$

∘

Also in the above rule we make use of the Core Semantics $A \uparrow P :: (\mathcal{C}, \mathcal{E}) \xrightarrow{\mathcal{L}} (\mathcal{C}', \mathcal{E}')$. It is worth pointing out that the LTS associated by non-stuttering semantics to a generic HA is *finite*. This nice property can be easily understood by considering that each HA has a finite set of events, a finite set of configurations and that the total set of transitions is finite, so that there is a finite number of subsets of transitions, i.e. there is a finite number of possible step-transitions. For HA H as in Fig. 2, the corresponding LTS(H) is shown in Fig.5[3].

4.2 Correctness

The original operational semantics proposed in [24] was proved correct w.r.t. the official UML Statechart Diagrams semantics, although the latter are defined only informally. The correctness theorem in [24] essentially states that the set of transitions fired during an arbitrary step is a *maximal* set $\mathcal{L}$ such that (a) all transitions in $\mathcal{L}$ are enabled, (b) they are non-conflicting and (c) there is no transition outside $\mathcal{L}$ which is enabled in the current status and which has higher priority than a transition in $\mathcal{L}$.

In this section we shall provide a correctness result for the non-stuttering semantics, showing its formal relation with the original semantics, in the form presented in [15] and recalled in Sect. 3.2 (Def. 3.3), which differs from that presented in [24] only in that each step-transition is explicitly labeled by set $\mathcal{L}$, which is omitted in [24].

In order to present the correctness result it is convenient to model the UML input-queue as a specific experimenter, namely the experimenter which simulates the data structure used for the queue. The experimenter we are interested in is $Queue(\mathcal{E}_0)$ where $\mathcal{E}_0$ is the initial content of the input-queue and $Queue$ is recursively defined, as shown in Fig. 6.

[3]For the sake of simplicity, in the examples in the present paper, the events generated as outputs are collected as *sets*, i.e. Θ_E is chosen to be 2^E; moreover, a singleton set $\{e\}$ is denoted by the element e it contains, when this cannot cause confusion.

Intuitively, for structure $\mathcal{E}$ in Θ_E, $Queue(\mathcal{E})$ will produce a transition if and only if there exist event e and structure $\mathcal{E}'$ such that $(\mathsf{Sel}\ \mathcal{E}\ e\ \mathcal{E}')$ holds, i.e. $\mathcal{E}$ is not empty. Moreover, f is bound to e and $\mathcal{F}'$ is bound to $\mathcal{E}'$. The expression $(f; \lambda\mathcal{F}''.Queue(join\ \mathcal{F}'\ \mathcal{F}''))$ is an *action prefix* which performs the event (action) currently bound to f, say e, and then behaves like the abstraction $\lambda\mathcal{F}''.Queue(join\ \mathcal{F}'\ \mathcal{F}'')$. The latter is a function which, when applied to a structure, say $\mathcal{E}''$, will behave again like a queue but with a different argument, i.e. $Queue(join\ \mathcal{E}'\ \mathcal{E}'')$.

The following proposition, proved in Appendix B, establishes the correctness of the new semantics definition. The transition relation in the operational semantics given in [15] is denoted by $\xrightarrow{\mathcal{L}}$.

Proposition 4.1
For HA $H = (F, E, \rho), \mathcal{C}, \mathcal{C}' \in \mathrm{Conf}_H, \mathcal{E}, \mathcal{E}', \mathcal{E}'' \in \Theta_E$, *the following holds:* $\exists\mathcal{L}.\ \mathcal{L} \neq \emptyset\ \wedge\ (\mathcal{C}, \mathcal{E}) \xrightarrow{\mathcal{L}} (\mathcal{C}', (join\ \mathcal{E}''\ \mathcal{E}'))$ *if and only if* $(Queue(\mathcal{E}) \mid\mid \mathcal{C}) \rightsquigarrow (Queue(join\ \mathcal{E}''\ \mathcal{E}') \mid\mid \mathcal{C}')$ $\diamond$

So the two semantic models generate the same step-transitions, except for stuttering. We remind here that a HA H stutters on input event e when there is *no* transition of *any* sequential automaton of H enabled by e in the current status. In other words, stuttering happens when the machine does not accept e in the current state. This refusal is modeled in the new semantics by not generating a step-transition at all. This last behavior is in line with traditional testing theories as developed e.g. in [17].

4.3 Testing Preorders

Below we define preorders which will allow us to "order" non-deterministic i/o-pair-LTSs, like HAs, according to their "amount of non-determinism" and to recollect testing equivalence as the equivalence induced by such preorders.

Definition 4.3 (Testing Preorders)
For $\mathcal{S}, \mathcal{S}'$ *i/o-pair LTSs we let*

i) $\mathcal{S} \sqsubseteq_{\sim MAY} \mathcal{S}'$ *iff for every* $\mathcal{T}$: *if* $\top \in \mathrm{Result}(\mathcal{T}, \mathcal{S})$ *then* $\top \in \mathrm{Result}(\mathcal{T}, \mathcal{S}')$

ii) $\mathcal{S} \sqsubseteq_{\sim MUST} \mathcal{S}'$ *iff for every* $\mathcal{T}$: *if* $\bot \notin \mathrm{Result}(\mathcal{T}, \mathcal{S})$ *then* $\bot \notin \mathrm{Result}(\mathcal{T}, \mathcal{S}')$

iii) $\mathcal{S} \sqsubseteq_{\sim} \mathcal{S}'$ *iff* $(\mathcal{S} \sqsubseteq_{\sim MAY} \mathcal{S}') \wedge (\mathcal{S} \sqsubseteq_{\sim MUST} \mathcal{S}')$ $\circ$

So $\mathcal{S} \sqsubseteq_{\sim MAY} \mathcal{S}'$ means that if a generic experimenter $\mathcal{T}$ *may* report success when experimenting with $\mathcal{S}$ it must be the case that $\mathcal{T}$ *may* report success also when experimenting with $\mathcal{S}'$. Symmetrically, $\mathcal{S} \sqsubseteq_{\sim MUST} \mathcal{S}'$ means that if a generic experimenter $\mathcal{T}$ *must* report success when experimenting with $\mathcal{S}$ it must be the case that $\mathcal{T}$ *must* report success also when experimenting with $\mathcal{S}'$. In other words if we know that $\mathcal{S}$ *may* pass a generic test $\mathcal{T}$ and $\mathcal{S} \sqsubseteq_{\sim MAY} \mathcal{S}'$ then we know also that $\mathcal{S}'$ *may* pass the test, where "*may* pass the test" is the informal equivalent of $\top \in \mathrm{Result}(\mathcal{T}, \mathcal{S})$, with the intuitive meaning that there may be a successful computation starting from the initial state of the experimental system $< \mathcal{T}, \mathcal{S} >$. Similarly if we know that $\mathcal{S}$ *must* pass a generic test $\mathcal{T}$ and $\mathcal{S} \sqsubseteq_{\sim MUST} \mathcal{S}'$ then we know also that $\mathcal{S}'$ *must* pass the test, where "*must* pass the test" is the informal equivalent of $\bot \notin \mathrm{Result}(\mathcal{T}, \mathcal{S})$, with the intuitive meaning that all computations starting from the initial state of the experimental system $< \mathcal{T}, \mathcal{S} >$ must be successful.

We let $\simeq$ denote the equivalence induced by the testing preorders, i.e. $\mathcal{S} \simeq \mathcal{S}'$ iff $\mathcal{S} \sqsubseteq_{\sim} \mathcal{S}'$ and $\mathcal{S} \sqsubseteq_{\sim} \mathcal{S}$. The proposition below easily follows from the relevant definitions and allows to identify $\sim$ with $\simeq$.

$Queue(\mathcal{F} : \Theta E) \triangleq$
$(\mathsf{Sel}\ \mathcal{F}\ f\ \mathcal{F}') \Rightarrow (f; \lambda\mathcal{F}''.Queue(\mathsf{Join}\ \mathcal{F}'\ \mathcal{F}''))$

Figure 6: Definition of $Queue(\mathcal{F})$

Proposition 4.2
$\mathcal{S} \sim \mathcal{S}'$ *iff* $\mathcal{S} \simeq \mathcal{S}'$ $\diamond$

Finally, we let $\sqsubset^{\aleph}_{\sim MAY}$, $\sqsubset^{\aleph}_{\sim MUST}$, $\sqsubset^{\aleph}_{\sim}$ denote the relativized preorders. For example $\mathcal{S} \sqsubset^{\aleph}_{\sim MUST} \mathcal{S}'$ iff for every $\mathcal{T} \in \aleph$, $\bot \notin \mathrm{Result}(\mathcal{T}, \mathcal{S}) \Rightarrow \bot \notin \mathrm{Result}(\mathcal{T}, \mathcal{S}')$.

4.4 Alternative characterization of testing preorders

The characterization of testing preoprders by means of the concepts of *MAY* and *MUST* is intuitively appealing, but is problematic when it comes to automatic verification of testing equivalence of LTSs. Such an aoutomatic verification can be performed based on so called *Acceptance Automata*, which are a variant of Acceptance Trees originally proposed by Hennessy. In order to be able to show the correspondence between testing equivalence defined in terms of the *MAY* and *MUST* preorders and the *Acceptance Automata*—which will be introduced in the next section—we give an intermediate alternative characterization of testing preorders in this section. First we introduce two auxiliary notions; *set closure* and *maximal functional subsets.*

Definition 4.4 (Set Closure)
For X a finite set of finite subsets of L, the closure *of X, $\mathbf{c}\ X$ is the smallest set such that:*

i) $X \subseteq \mathbf{c}\ X$

ii) if $x_1, x_2 \in \mathbf{c}\ X$ *then also* $x_1 \cup x_2 \in \mathbf{c}\ X$

iii) if $x_1, x_2 \in \mathbf{c}\ X$ *and* $x_1 \subseteq x \subseteq x_2$ *then also* $x \in \mathbf{c}\ X$. $\circ$

The following definition is necessary for identifying the *functional* subsets of subsets of L whenever L is a set of i/o-pairs. Functional sets, which are in fact (finite) functions, are used for modeling single steps of input/output behavior.

Definition 4.5 (mfs)
For X finite set of finite subsets of L we let

$$mfs\ X =_{df} \bigcup_{x \in X} (mf\ x)$$

where
$mf\ x =_{df} \{y \in (func\ x) \mid \nexists y' \in (func\ x).\ y \subset y'\}$ *and*
$func\ x =_{df} \{y \subseteq x \mid \forall (i_1, u_1), (i_2, u_2) \in y.\ i_1 = i_2 \Rightarrow u_1 = u_2\}$ $\circ$

For finite set X of finite subsets of L, $mfs\ X$ generates the *maximal functional subsets* of the elements of X, by applying function mf to each of them. Function mf splits each set into its maximal functional subsets. Each functional set is indeed a *function* from input-events to output-events. As we will see, intuitively, every such a set represents an instance of *external* non-determinism relative to a single step of the machine. Similarly, but in a complementary way, *internal* non-determinism relative to a single step of the machine is coded by means of having different functional sets as elements of the same set associated to such a step. Notice that $func\ \emptyset = \{\emptyset\} = mf\ \emptyset$ and $mfs\ \{\emptyset\} = \{\emptyset\}$. The following definitions introduce a preorder on finite LTSs which will be used for defining the intermediate equivalence $\approx$ and which will be proved to coincide with the testing preorder.

Definition 4.6
For finite LTSs $\mathcal{S}, \mathcal{S}'$ over $L = L_I \times L_U$

i) $\mathcal{S} <<_{MAY} \mathcal{S}'$ *iff* $(\mathsf{Ian}\ \mathcal{S}) \subseteq (\mathsf{Ian}\ \mathcal{S}')$

ii) $\mathcal{S} <<_{MUST} \mathcal{S}'$ *iff* $\forall \gamma \in L^*.\ mfs(\mathbf{c}\ (\mathsf{AS}\ \mathcal{S}'\ \gamma)) \subset\subset mfs(\mathbf{c}\ (\mathsf{AS}\ \mathcal{S}\ \gamma))$

iii) $\mathcal{S} << \mathcal{S}'$ *iff* $\mathcal{S} <<_{MAY} \mathcal{S}' \wedge \mathcal{S} <<_{MUST} \mathcal{S}'$

where $X \subset\subset Y$ *iff* $\forall x \in X.\ \exists y \in Y.\ y \subseteq x$ ∘

It is easy to show that $<<$ is indeed a preorder so that it induces the following equivalence:

Definition 4.7
For finite LTSs $\mathcal{S}, \mathcal{S}'$ *over* L $\mathcal{S} \approx \mathcal{S}'$ *iff* $\mathcal{S} << \mathcal{S}' \wedge \mathcal{S}' << \mathcal{S}$ ∘

The following theorem establishes the first correspondence result, namely the correspondence between the testing preorders (Def. 4.3) and the preorders defined in Def. 4.6:

Theorem 4.1
For all finite LTSs $\mathcal{S} = (S, s_{in}, L, \rightarrow)$ $\mathcal{S}' = (S', s'_{in}, L, \rightarrow)$ *over* $L = L_I \times L_U$ *the following holds:*

a) $\mathcal{S} \sqsubseteq_{\sim MAY} \mathcal{S}'$ *iff* $\mathcal{S} <<_{MAY} \mathcal{S}'$

b) $\mathcal{S} \sqsubseteq_{\sim MUST} \mathcal{S}'$ *iff* $\mathcal{S} <<_{MUST} \mathcal{S}'$

c) $\mathcal{S} \sqsubseteq_{\sim} \mathcal{S}'$ *iff* $\mathcal{S} << \mathcal{S}'$ ◇

As a corollary we have the link between testing equivalence and the relation $\approx$ defined on LTSs.

Corollary 4.1
For finite LTSs $\mathcal{S}, \mathcal{S}'$ *over* $L = L_I \times L_U$, $\mathcal{S} \simeq \mathcal{S}'$ *iff* $\mathcal{S} \approx \mathcal{S}'$ ◇

At this point we can already say something about the exact nature of this notion of testing LTSs over L. Essentially it has to do with internal non-determinism, as it can be detected by means of "black-box" testing. Intuitively $\mathcal{S} \sqsubseteq_{\sim} \mathcal{S}'$ if they have the same set of traces but in some sense $\mathcal{S}$ is "more non-deterministic", or equivalently, "more chaotic" then $\mathcal{S}'$. In other words, although the sequences of input/output interactions of both systems are the same, an experimenter $\mathcal{T}$ may experience failures with $\mathcal{S}$ "more often" than with $\mathcal{S}'$. In this sense, $\mathcal{S}$ has a"higher degree of non-determinism" than $\mathcal{S}'$.

4.5 Finite Acceptance Automata

In this section we introduce the model of *Finite Acceptance Automata* (FAAs), equipped with a preorder $\leq_{FAA}$ and the equivalence relation $\equiv_{FAA}$ it induces. FAAs are a natural extension of Finite Acceptance Trees to the case of i/o-pairs LTS. Finite Acceptance Trees have been originally introduced by Hennessy [17]; they have been adapted to the case of systems with explicit input/output behavior in [31]. Both in [17] and [31] Acceptance Trees form a semantic domain within a denotational approach. So, Acceptance Trees modeling systems with traces of unlimited length—even finite state systems—are characterized by infinite fixpoints. In this paper, instead, we are using an operational—rather than denotational—approach and we shall use FAAs for modeling the behavior of finite state systems with i/o-labels, including those with traces of unlimited length. In other words, our acceptance structures are generic finite graphs and not simply finite trees or directed acyclic graphs.

The reason why we introduce FAAs is fairly simple: they can easily be mapped into finite *deterministic* LTSs on which strong bisimulation equivalence can be automatically checked, and such a mapping preserves the equivalence $\equiv_{FAA}$ on FAAs. On the other hand, we can map finite LTSs over i/o-pairs into FAAs in such a way that *testing equivalence* is preserved, i.e. two finite LTSs over i/o-pairs are testing equivalent if and only if their images via such a mapping are equivalent according to $\equiv_{FAA}$. In conclusion, FAAs represent an effective model for performing automatic verification of testing equivalence over UMLSDs.
Before defining the FAA model, we need to define the notion of *saturated sets.*

Definition 4.8 (Saturated sets)
For finite subset S *of* L*, an* S*-set* A *is a finite, non-empty set of subsets of* L *which satisfies:*

For $\mathcal{S}$ finite LTS over $L = L_I \times L_U$, let $(\mathsf{T}_{\mathsf{FAA}}\ \mathcal{S})$ be the FAA computed as follows:

1. Apply the Aho-Ullman "Subset Construction" Algorithm ([1], pag. 93) to $\mathcal{S}$, getting deterministic automaton $d = (D, d_{\text{in}}, \rightarrow_d)$ over L;
2. Let $(\mathsf{T}_{\mathsf{FAA}}\ \mathcal{S})$ be the FAA $\alpha = (A, \alpha_{\text{in}}, \rightarrow_\alpha, \mathsf{AS}_{\text{FAA}}\ \alpha)$ defined as follows:
   (a) $A = D$
   (b) $\alpha_{\text{in}} = d_{\text{in}}$
   (c) For all $\alpha', \alpha'' \in A, (i,u) \in L$, $\alpha' \xrightarrow{(i,u)}_\alpha \alpha''$ iff $\alpha' \xrightarrow{(i,u)}_d \alpha''$
   (d) For all $\gamma \in L^*$ let
   - $\mathsf{AS}_{\text{FAA}}\ \alpha\ \gamma = mfs\ (\mathbf{c}\ (\mathsf{AS}\ \mathcal{S}\ \gamma))$, if $\gamma \in \mathsf{lan}\ \mathcal{S}$
   - $\mathsf{AS}_{\text{FAA}}\ \alpha\ \gamma = \emptyset$, if $\gamma \notin \mathsf{lan}\ \mathcal{S}$

Figure 7: The algorithm for mapping $\mathsf{T}_{\mathsf{FAA}}$

i) $\forall X \in A.\ X \subseteq S$

ii) $\forall x \in S.\ \exists X \in A.\ x \in X$

iii) $\forall X_1, X_2 \in A.\ X_1 \cup X_2 \in A$

iv) $\forall X_1, X_2 \in A, X.\ X_1 \subseteq X \subseteq X_2 \Rightarrow X \in A$

A finite set A of finite subsets of L is saturated *if it is an S-set for some S.* ∘

Finite Acceptance Automata (FAAs) are defined below

Definition 4.9 (FAA)
A finite acceptance automaton *α over L is a deterministic finite LTS over L, where also nodes are labeled*[4]*. The node of α identified by sequence $\gamma \in \mathsf{lan}\ \alpha$ is labeled by $(mfs\ A)$ for some $(\mathsf{S}\ \alpha\ \gamma)$-set A. Such a label will be denoted by $\mathsf{AS}_{\text{FAA}}\ \alpha\ \gamma$, and is assumed equal to $\emptyset$ whenever $\gamma \notin \mathsf{lan}\ \alpha$.* ∘

It is easy to see that the relation on FAA defined below is a preorder (but not a partial order).

Definition 4.10 ($\leq_{FAA}$ and $\equiv_{FAA}$)
For FAAs α, α' over L, $\alpha \leq_{FAA} \alpha'$ iff

- $\mathsf{lan}\ \alpha = \mathsf{lan}\ \alpha'$, *and*
- $\forall \gamma \in \mathsf{lan}\ \alpha.\ \mathsf{AS}_{\text{FAA}}\ \alpha'\ \gamma \subseteq \mathsf{AS}_{\text{FAA}}\ \alpha\ \gamma$

The equivalence induced by $\leq_{FAA}$ is denoted by $\equiv_{FAA}$. ∘

Intuitively, $\alpha \leq_{FAA} \alpha'$ if they have the same set of traces but α represents "more non-deterministic" systems. Such non-determinism is coded in the acceptance sets. In order to compute the FAA $(\mathsf{T}_{\mathsf{FAA}}\ \mathcal{S})$ associated to any $\mathcal{S}$, finite LTS over L, we proceed in a similar way as in [4]. The algorithm is defined in Fig.7. Fig.9 (a) shows the result of applying mapping $\mathsf{T}_{\mathsf{FAA}}$ to the LTS of Fig.8 (b) which is the semantics of the UMLSD of Fig.8 (a).

The following proposition easily derives from the fact that, by construction, $\mathsf{T}_{\mathsf{FAA}}\ \mathcal{S}$ is deterministic, finite and its language is the same as $\mathsf{lan}\ \mathcal{S}$; moreover, for the node of $\mathsf{T}_{\mathsf{FAA}}\ \mathcal{S}$ identified by sequence γ, $\mathbf{c}\ (\mathsf{AS}\ \mathcal{S}\ \gamma)$ is an $(\mathsf{S}\ \mathcal{S}\ \gamma)$-set as it can easily be seen from the definitions of $\mathbf{c}$ and AS.

[4] All definitions for LTS are thus valid also for FAAs.

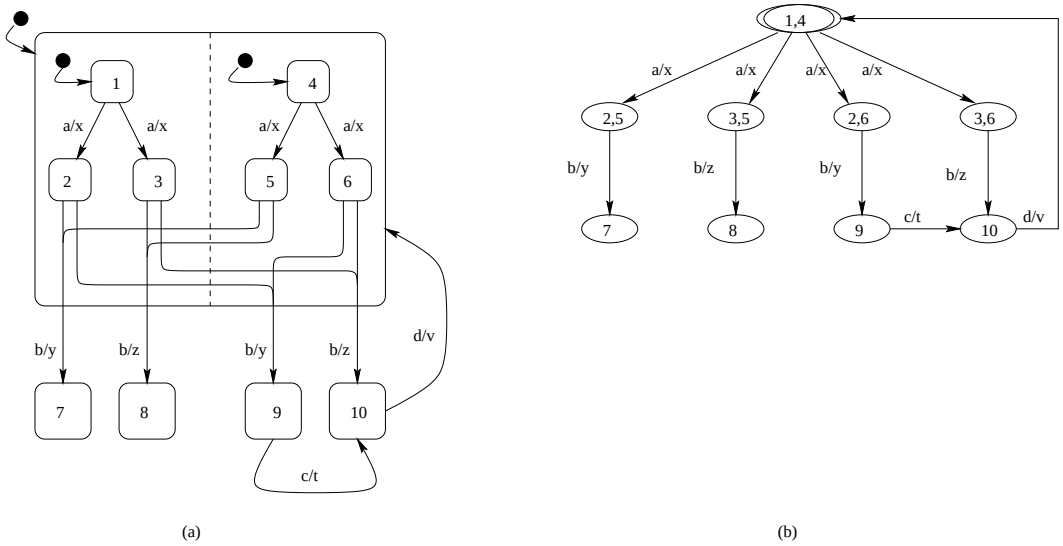

Figure 8: A simple UMLSD (a) and its LTS (b)

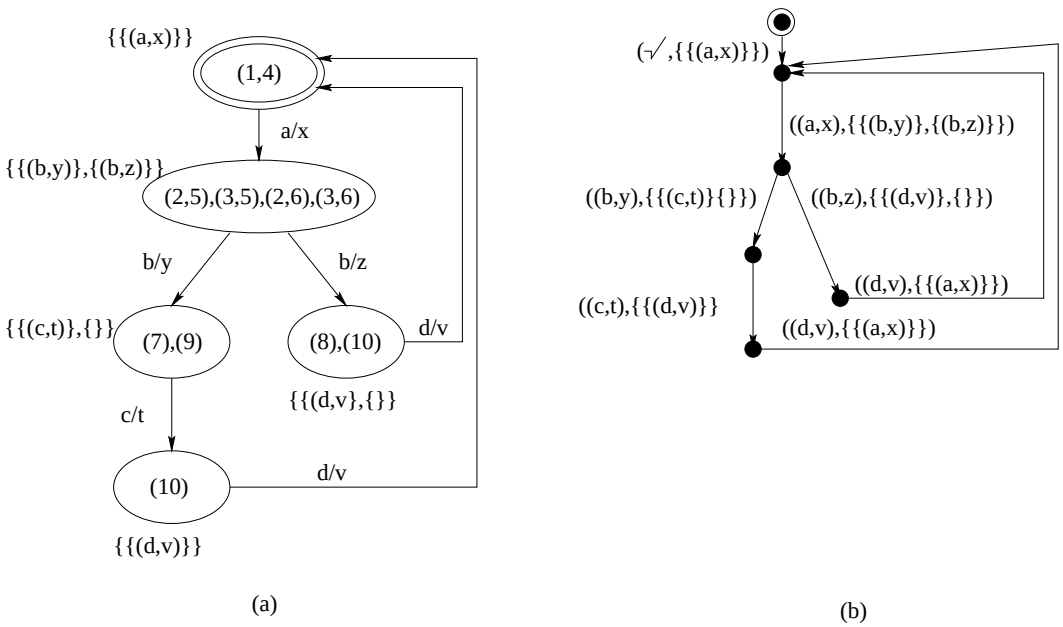

Figure 9: FAA (a) and related deterministic LTS (b) associated to the LTS of Fig.8

Proposition 4.3
For finite LTS $\mathcal{S}$ over $L = L_I \times L_U$, $\mathcal{T}_{FAA}\,\mathcal{S}$ is a FAA over L. $\diamond$

We are now ready for giving the second correspondence theorem

Theorem 4.2
For all finite LTSs $\mathcal{S}$, $\mathcal{S}'$ over $L = L_I \times L_U$ the following holds: $\mathcal{S} << \mathcal{S}'$ iff $(\mathcal{T}_{FAA}\,\mathcal{S}) \leq_{FAA} (\mathcal{T}_{FAA}\,\mathcal{S}')$ $\diamond$

As a corollary of Theorem 4.2 we have the link between the relation $\approx$ on LTSs and the equivalence of FAAs.

Corollary 4.2
For finite LTSs $\mathcal{S}, \mathcal{S}'$ over L, $\mathcal{S} \approx \mathcal{S}'$ iff $(\mathcal{T}_{FAA}\,\mathcal{S}) \equiv_{FAA} (\mathcal{T}_{FAA}\,\mathcal{S}')$. $\diamond$

4.6 Testing Equivalence Verification

In this section we show how the results given in the previous section can be used for effective verification of testing equivalence over UMLSDs. We start by stating the final correspondence result, which easily follows from Corollaries 4.1 and 4.2.

Corollary 4.3
For all finite LTSs $\mathcal{S}$, $\mathcal{S}'$ over L, $\mathcal{S} \simeq \mathcal{S}'$ iff $(\mathcal{T}_{FAA}\,\mathcal{S}) \equiv_{FAA} (\mathcal{T}_{FAA}\,\mathcal{S}')$ $\diamond$

The above result allows us to reduce the problem of checking whether two UMLSDs S and S' are testing equivalent to the problem of checking whether $(\mathsf{T}_{\mathsf{FAA}}\,(\mathsf{LTS}(H))) \equiv_{FAA} (\mathsf{T}_{\mathsf{FAA}}\,(\mathsf{LTS}(H')))$ where H (resp. H') is the HA representing S (resp. S'). In the remainder of the paper we show that checking $\alpha \equiv_{FAA} \alpha'$, for FAAs α, α', can be reduced in turn to checking (strong) bisimulation equivalence. Below we recall the definition of bisimulation equivalence [33].

Definition 4.11
A binary relation R on states of LTSs over label set L is a (strong) bisimulation iff for all $l \in L$ and s_1, s_2 with $s_1\ R\ s_2$

- *whenever $s_1 \xrightarrow{l} s_1'$ for some s_1' also $s_2 \xrightarrow{l} s_2'$ for some s_2' such that $s_1'\ R\ s_2'$, and*
- *whenever $s_2 \xrightarrow{l} s_2'$ for some s_2' also $s_1 \xrightarrow{l} s_1'$ for some s_1' such that $s_1'\ R\ s_2'$* $\circ$

We say that s_1 and s_2 are (strong) bisimulation equivalent, written $s_1 \approx_{bis} s_2$ in this paper, iff there exists a bisimulation R such that $s_1\ R\ s_2$.

Two LTSs $\mathcal{S}$ and $\mathcal{S}'$ are bisimulation equivalent, written $\mathcal{S} \approx_{bis} \mathcal{S}'$, if and only if their initial states are so. It is important to point out that there are tools available nowadays for automatic verification of bisimulation equivalence for finite LTSs (see, e.g. [13]). In order to reduce our problem to bisimulation equivalence checking we first build the LTS Up α for FAA α according to the algorithm shown in Fig.10. The algorithm simply moves node labels up to the transitions pointing to such nodes, introducing a new node and a new transition for the label of the initial node. Fig.9 (b) shows the result of applying the algorithm to the FAA of Fig.9 (a). It is easy to see that the two lemmas below directly follow from the definitions of Up and $\equiv_{FAA}$.

Lemma 4.1
For FAA α over L, $(Up\ \alpha)$ is a deterministic, finite LTS. $\diamond$

Lemma 4.2
For FAAs α and α' over L the following holds: $\alpha \equiv_{FAA} \alpha'$ iff $\mathsf{lan}\,(Up\ \alpha) = \mathsf{lan}\,(Up\ \alpha')$ $\diamond$

But then, since strong bisimulation equivalence coincides with trace equivalence for deterministic LTSs (see e.g. [22]), from the above two lemmas we can easily prove the following

For FAA over L $\alpha = (A, \alpha_{\text{in}}, \rightarrow_\alpha, \mathsf{AS}_{\text{FAA}}\ \alpha)$, let $\mathsf{Up}\ \alpha$ be the finite LTS over $L \times 2^{2^L}$ $d = (D, d_{\text{in}}, \rightarrow_d)$ defined as follows:

1. $D = A \cup \{d_{\text{in}}\}$, with $d_{\text{in}} \notin A$
2. $\rightarrow_d$ is the smallest relation over $D \times (L \times 2^{2^L}) \times D$ such that the following holds:
   - $d_{\text{in}} \xrightarrow{v}_d \alpha_{\text{in}}$ with $v = (\surd, \mathsf{AS}_{\text{FAA}}\ \alpha\ \epsilon)$ and $\surd \notin L$
   - For all $d', d'' \in D$, $x \in L$ and $\gamma \in L^*$ such that γ identifies d' in α and γx identifies d'' in α, $d' \xrightarrow{v}_d d''$ iff $d' \xrightarrow{x}_\alpha d''$ and $v = (x, \mathsf{AS}_{\text{FAA}}\ \alpha\ \gamma x)$.

Figure 10: The algorithm for mapping Up

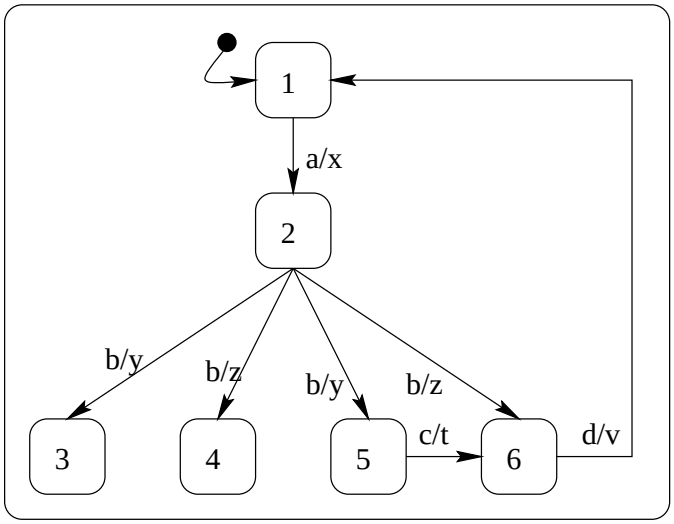

Figure 11: A UMLSD testing equivalent to that of Fig.8.

Theorem 4.3
For all finite LTSs $\mathcal{S}$, $\mathcal{S}'$ over $L = L_I \times L_U$ the following holds: $\mathcal{S} \simeq \mathcal{S}'$ if and only if $\mathsf{Up}(\mathcal{T}_{FAA}\ \mathcal{S}) \approx_{bis} \mathsf{Up}(\mathcal{T}_{FAA}\ \mathcal{S}')$ $\diamond$

We leave it to the reader to verify that the UMLSD of Fig.11 is testing equivalent[5] to that of Fig.8 (a).

5 Conformance Testing

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 conformance relations and conformance testing can be found in [42]. The theory has been developed mainly in the context of process algebras and input/output LTSs. Input/output LTSs[6] [29] are LTSs where the set of labels is *partitioned* into two separate sets, i.e. input labels and output ones. Moreover, in the context of conformance testing theories, such LTSs are required to be *input enabled*, i.e. for each label of the input set, in

[5] Actually, the associated deterministic LTSs turn out to be not only bisimulation equivalent, but also isomorphic. Obviously this does not need to be the case in general.

[6] Strictly speaking "Automata".

each state of the LTS there must be at least one outgoing transition labeled by such a label. Finally, the situation in which, in a given state, an LTS does not generate any output at all is modeled by an outgoing transition labeled by a special label denoting "quiescence". Both specifications and implementations are modeled by input enabled LTSs with quiescence.

Under the above modeling assumptions, one of the most successful formal conformance relations is the **ioco** relation proposed by Tretmans in the above mentioned work [42]. 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 the previous section we have developed a model of system behavior based on the assumption that absence of reaction by a system to a stimulus presented by an experimenter results in a deadlock affecting both the system and the experimenter. In this section, instead, we define a slightly richer semantics for HAs, which we call the *stuttering semantics*, where absence of reaction is represented *explicitly* in the associated LTSs, in a way which is similar to quiescence and which naturally represents the notion of *stuttering* in the context of UMLSCs. A HA H (or equivalently the UMLSC it represents) experiences a stuttering step on input event e whenever, in the current configuration $\mathcal{C}$ *no* transition is enabled on such input e. The input event e is consumed anyway but no state change occurs in H[7]. As we will see, in the stuttering semantics when stuttering occurs, the output component of the label of the involved step-transition is the special symbol Σ. Thus, in the remainder of this paper we will focus on input enabled LTSs over $L_I \times L_U$ where Σ, with $\Sigma \notin L_I$, may belong to L_U. Moreover we will let ${}^{\Sigma}\Theta_E$ denote $\Theta_E \cup \{\Sigma\}$.

In Sect. 5.1 the stuttering semantics is given and its relation with the original semantics of UMLSCs proposed in [15] is addressed in Sect. 5.2. In Sect. 5.3, the Conformance Relation is introduced, on which the test case generation algorithm (Sect. 5.4) is based.

Before proceeding with the definition of the stuttering semantics we need some further auxiliary definitions related to the Conformance Relation and to test case generation:

Definition 5.1
For LTS $\mathcal{S} = (S, s_{in}, L, \rightarrow)$, *with* $L = L_I \times L_U$, $i \in L_I$, $s \in S$, $Z \subseteq S$, $\gamma \in L^*$, $\mathcal{F} \subseteq L^*$:

- *the* states *of* s after γ *is the set* $(s\ \mathit{after}\ \gamma) =_{df} \{s' \mid s \xrightarrow{\gamma} s'\}$.
- *the* output *of* Z *on* i *is the set* $(\mathit{out}\ Z\ i) =_{df} \bigcup_{s \in Z} \{u \in L_U \mid s \xrightarrow{(i,u)}\}$; *we let* $(\mathit{OUT}\ s\ \gamma\ i)$ *be the set* $(\mathit{out}\ (s\ \mathit{after}\ \gamma)\ i)$; *moreover, we will often denote* $(\mathit{OUT}\ s_{in}\ \gamma\ i)$ *by* $(\mathit{OUT}\ \mathcal{S}\ \gamma\ i)$;
- *the traces of* $\mathcal{F}$ after γ *is the set* $(\mathcal{F}\ \mathit{after}^*\ \gamma) =_{df} \{\gamma' \mid \gamma\gamma' \in \mathcal{F}\}$;
- *the* output *of* $\mathcal{F}$ *on* i *is the set* $(\mathit{out}^*\ \mathcal{F}\ i) =_{df} \{u \in L_U \mid \exists \gamma.\ (i,u)\gamma \in \mathcal{F}\}$; *we let* $(\mathit{OUT}^*\ \mathcal{F}\ \gamma\ i)$ *be the set* $(\mathit{out}^*\ (\mathcal{F}\ \mathit{after}^*\ \gamma)\ i)$
- $\mathcal{S}$ *is* input enabled *iff* $\forall s \in S, i \in L_I.\ \exists u \in L_U.\ s \xrightarrow{(i,u)}$. ∘

5.1 Stuttering Semantics

Definition 5.2
The stuttering semantics of an HA $H = (F, E, \rho)$ *is the LTS over* $E \times {}^{\Sigma}\Theta_E$, ${}^{\Sigma}LTS(H) =_{df}$ $(\mathrm{Conf}_H, \mathcal{C}_{in}, \rightarrow_{\Sigma})$ *where (i)* Conf_H *is the set of configurations, (ii)* $\mathcal{C}_{in} \in \mathrm{Conf}_H$ *is the initial configuration, (iii)* $\rightarrow_{\Sigma} \subseteq \mathrm{Conf}_H \times (E \times {}^{\Sigma}\Theta_E) \times \mathrm{Conf}_H$ *is the* step-*transition relation defined below.* ∘

As usual, we write $\mathcal{C} \xrightarrow{e/\mathcal{E}}_{\Sigma} \mathcal{C}'$ for $(\mathcal{C}, (e, \mathcal{E}), \mathcal{C}') \in \rightarrow_{\Sigma}$.

[7] In fact in UML the notion of *deferred events* is introduced in order not to loose events as a consequence of stuttering. In our work we do not take deferred events into consideration.

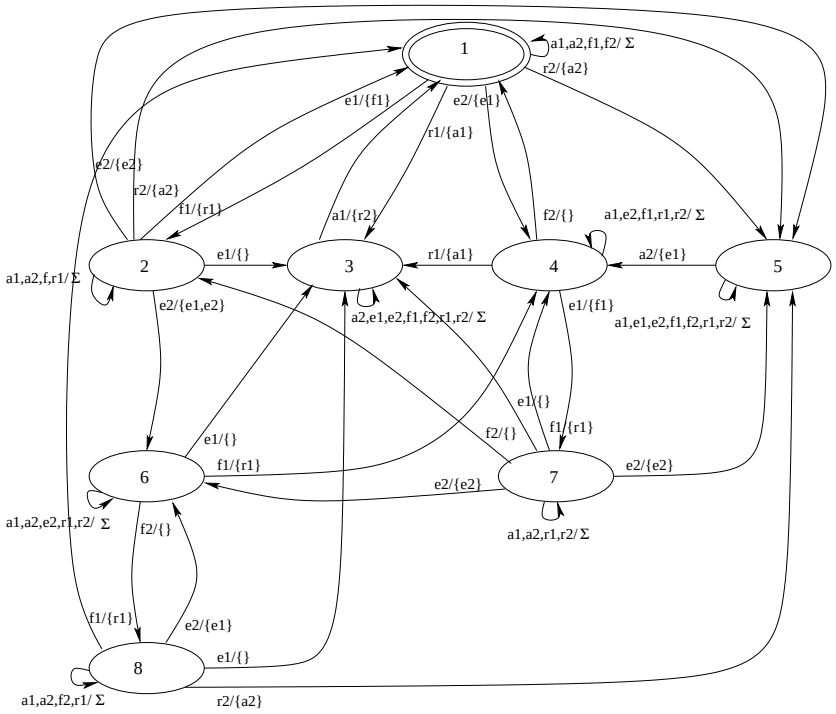

Figure 12: ${}^{\Sigma}\mathrm{LTS}(H)$, for H of Fig. 2

Definition 5.3 (Stuttering Transition Deduction System)

$$\frac{e \in E,\ \mathcal{L} \neq \emptyset,\ H \upharpoonright \emptyset :: (\mathcal{C}, \{e\}) \xrightarrow{\mathcal{L}}_{\Sigma} (\mathcal{C}', \mathcal{E})}{\mathcal{C} \xrightarrow{e/\mathcal{E}}_{\Sigma} \mathcal{C}'} \quad (1)$$

$$\frac{e \in E, ; H \upharpoonright \emptyset :: (\mathcal{C}, \{e\}) \xrightarrow{\emptyset}_{\Sigma} (\mathcal{C}', \mathcal{E})}{\mathcal{C} \xrightarrow{e/\Sigma}_{\Sigma} \mathcal{C}'} \quad (2)$$

∘

Fig. 12 shows ${}^{\Sigma}\mathrm{LTS}(H)$, for H as in Fig. 2. 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 $i_1, \ldots, i_k/\Sigma$.

The following lemma shows some interesting features of the stuttering semantics:

Lemma 5.1
For HA $H = (F, E, \rho)$, all $\mathcal{C} \in \mathrm{Conf}_H$, and $e \in E$ the following holds:

i) $\exists u \in {}^{\Sigma}\Theta_E, \mathcal{C}' \in \mathrm{Conf}_H . \mathcal{C} \xrightarrow{e/u}_{\Sigma} \mathcal{C}'$, *i.e.* ${}^{\Sigma}LTS(H)$ *is* input enabled *over* $E \times {}^{\Sigma}\Theta_E$.

ii) $\mathcal{C} \xrightarrow{e/\Sigma}_{\Sigma} \mathcal{C}'$ *for some* $\mathcal{C}' \in \mathrm{Conf}_H$ *implies* $\mathcal{C} = \mathcal{C}'$.

iii) $\mathcal{C} \xrightarrow{e/\Sigma}_{\Sigma} \mathcal{C}$ *implies* $\nexists \mathcal{E} \in \Theta_E, \mathcal{C}' \in \mathrm{Conf}_H . \mathcal{C} \xrightarrow{e/\mathcal{E}}_{\Sigma} \mathcal{C}'$.

◇

Finally, it is easy to see that also the stuttering semantics associates a *finite* LTS to each HA.

5.2 Correctness

As in the case of the non-stuttering semantics, also for the stuttering semantics we show the formal link to and the original semantics, presented in [15], and recalled in Sect. 3.2.

We recursively define a specific experimenter which behaves like a queue and is the same as that used in Sect. 4.2, except that it has to deal also with Σ. In particular, when receiving Σ from the HA, it disregards it, as it can be seen in Fig. 13.

The following proposition establishes the correctness of the stuttering semantics definition. The transition relation in the operational semantics given in [15] is denoted by $\xrightarrow{\mathcal{L}}$.

Proposition 5.1
For hierarchical automaton $H = (F, E, \rho), \mathcal{C}, \mathcal{C}' \in \mathrm{Conf}_H$, $\mathcal{E}$, $\mathcal{E}'$, $\mathcal{E}'' \in \Theta_E$, *the following holds:*
$(\Sigma Queue(\mathcal{E}) \mid\mid \mathcal{C}) \rightsquigarrow (\Sigma Queue(\mathsf{Join}\ \mathcal{E}''\ \mathcal{E}') \mid\mid \mathcal{C}')$ *iff* $\exists \mathcal{L}.\ (\mathcal{C}, \mathcal{E}) \xrightarrow{\mathcal{L}} (\mathcal{C}', (\mathsf{Join}\ \mathcal{E}''\ \mathcal{E}'))$ $\diamond$

5.3 Conformance Relation

In the context of the present work, we assume that a *specification* of system behavior is given in the form of a UMLSC H (in practice we use its HAs representation) and we make reference mainly to the LTS associated to H by the stuttering semantics, i.e. ${}^{\Sigma}\mathsf{LTS}(H)$, over $L = L_I \times L_U$. An *implementation* for H will be modeled by an input-enabled LTS 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 input-enabled LTSs 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 [41].

In the approach to conformance testing introduced by Tretmans, [42], 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 labeled 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 [41]. 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* $\sqsubseteq^{\mathcal{F}}_{\mathsf{co}}$ follows:

Definition 5.4
For LTSs $\mathcal{S} = (S, s_{in}, L, \longrightarrow)$, *with* $L = L_I \times L_U$, $\mathcal{S}' = (S', s'_{in}, L', \longrightarrow')$, *with* $L' = L'_I \times L'_U$, *and* $\mathcal{F} \subseteq (L_I \times L_U)^*$: $\mathcal{S}' \sqsubseteq^{\mathcal{F}}_{\mathsf{co}} \mathcal{S}$ *iff* $\forall \gamma \in \mathcal{F}, i \in L_I.\ \mathit{OUT}\,\mathcal{S}'\ \gamma\ i \subseteq \mathit{OUT}\,\mathcal{S}\ \gamma\ i$ $\circ$

$$
\begin{array}{l}
\Sigma\mathrm{Queue}(\mathcal{F} : \Theta_E) \triangleq \\
(\mathsf{Sel}\ \mathcal{F}\ f\ \mathcal{F}') \Rightarrow (f; \lambda X : {}^{\Sigma}\Theta_E.\ \ ((X \neq \Sigma \Rightarrow \Sigma\mathrm{Queue}(\mathsf{Join}\ \mathcal{F}'\ X)) + \\
\qquad\qquad\qquad\qquad\qquad\qquad (X = \Sigma \Rightarrow \Sigma\mathrm{Queue}(\mathcal{F}'))))
\end{array}
$$

Figure 13: Definition of $\Sigma\mathrm{Queue}(\mathcal{F})$

In the following we will let $\sqsubseteq_{\mathsf{co}}$ (i.e. "conforms to") denote $\sqsubseteq_{\mathsf{co}}^{(\mathsf{lan}\ \mathcal{S})}$ and we will mainly focus on $\sqsubseteq_{\mathsf{co}}$. Intuitively, $\mathcal{S}' \sqsubseteq_{\mathsf{co}} \mathcal{S}$ 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 [41, 42] but its technical definition has been adapted here for UMLSCs. The following lemmas relate the conformance relation with LTS languages.

Lemma 5.2
For $\mathcal{S}'$ finite LTS over $L'_I \times L'_U$, $\mathcal{S}$ finite LTS over $L_I \times L_U$, the following holds:
$(\mathsf{lan}\ \mathcal{S}') \subseteq (\mathsf{lan}\ \mathcal{S})$ *implies* $\mathcal{S}' \sqsubseteq_{\mathsf{co}} \mathcal{S}$ ◇

Lemma 5.3
For $\mathcal{S}'$ finite LTS over $L'_I \times L'_U$, $\mathcal{S}$ finite LTS over $L_I \times L_U$, with $L'_I \subseteq L_I$, the following holds:
$\mathcal{S}' \sqsubseteq_{\mathsf{co}} \mathcal{S}$ *implies* $(\mathsf{lan}\ \mathcal{S}') \subseteq (\mathsf{lan}\ \mathcal{S})$ ◇

The notion of *verdict* of is central in conformance testing. A *verdict* is the result of testing a system $\mathcal{S}$ against a test case $\mathcal{T}$, the latter being an experimenter as defined in Sect. 3. The test is passed if all computations are successful:

Definition 5.5 (Verdict)
The verdict $\mathcal{V}$ *of* $\mathcal{T}$ *on* $\mathcal{S}$ *is defined as follows:*

$$\mathcal{V}\ \mathcal{T}\ \mathcal{S} = \begin{cases} \textbf{pass} & \textit{if } \bot \notin \mathrm{Result}(\mathcal{T}, \mathcal{S}) \\ \textbf{fail} & \textit{otherwise} \end{cases}$$

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

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

○

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

Definition 5.6 (Completeness)
Given specification $\mathcal{S}$ *and test suite* $\aleph$

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

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

5.4 The Test Case Generation Algorithm

Once a formal specification of a system has been developed, it is possible to mechanically generate test cases for that specification. The test case generation algorithm TD proposed in this paper is a non-deterministic algorithm which 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, returns a test case $\mathcal{U}$ in the test case language introduced in Sect. 3.3. The definition of the test case generation algorithm TD is given in Fig.14. The intuitive behavior of the algorithm is rather simple; at each call, the algorithm generates a single test case.

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.

$\mathsf{TD}_{L,L'}\ \mathcal{F}\ =$
Non-deterministically choose between options (1) and (2) below
1) generate "$\tau; \mathcal{W}$"
2) generate "$e; \lambda x : L'_U.\ x = u_1\ \Rightarrow\ \mathcal{U}_1$
$+$
$\vdots$
$+$
$x = u_k\ \Rightarrow\ \mathcal{U}_k$
$+$
$x \notin \{u_1, \ldots, u_k\}\ \Rightarrow\ \delta$"
where:
e is non-deterministically chosen in $L_I \cap L'_I$
such that $\mathsf{out}^*\ \mathcal{F}\ e = \{u_1, \ldots, u_k\} \neq \emptyset$, and
$\mathcal{U}_j \in \mathsf{TD}_{L,L'}\ (\mathcal{F}\ \mathsf{after}^*\ (e, u_j))$ for $j = 1, \ldots, k$

Figure 14: The Test Case Generation Algorithm

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 $\mathsf{out}^*\ \mathcal{F}\ e = \{u_1, \ldots, u_k\}$ is non-empty (notice that such an e exists when dealing with input enabled LTS over i/o-pairs associated to UMLSCs); Intuitively, $u_1, \ldots, u_k$ 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 $u_1, \ldots, u_k$, it stops without reporting success, otherwise, assuming that the output of the implementation is u_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 $(\mathsf{TD}_{L,L'}\ \mathcal{F})$. Notice that, by construction, test cases generated by TD have a tree-like structure; there is no looping possibility in their execution.
The following proposition easily follows from the definition of the algorithm, and the above considerations, by observing that sets $\{u_1, \ldots, u_k\}$ are finite when $\mathcal{F}$ is the language of ${}^{\Sigma}\mathsf{LTS}(H)$ for some HA H.

Proposition 5.2
For every HA H with ${}^{\Sigma}LTS(H)$ over i/o-pair set L, and i/o-pair set L', every test case $\mathcal{U} \in TD_{L,L'}\ (\mathsf{lan}({}^{\Sigma}LTS(H)))$ is finite. $\diamond$

Typically $\mathsf{lan}({}^{\Sigma}\mathsf{LTS}(H))$ 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 $\mathsf{lan}({}^{\Sigma}\mathsf{LTS}(H))$. Notice also that the set of all test cases generated using $\mathsf{TD}_{L,L'}$ on $\mathsf{lan}({}^{\Sigma}\mathsf{LTS}(H))$ 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 $\mathsf{TD}_{L,L'}\ (\mathsf{lan}({}^{\Sigma}\mathsf{LTS}(H)))$ are finite.

The following theorem establishes completeness of the test case generation algorithm, when applied to (the language of) a specification ${}^{\Sigma}\mathsf{LTS}(H)$:

Theorem 5.1
For every HA H with ${}^{\Sigma}LTS(H)$ over $L = L_I \times L_U$, and set $L' = L'_I \times L'_U$, the test suite

$$TD_{L,L'}\ (lan({}^{\Sigma}LTS(H)))$$

is complete *w.r.t.* ${}^{\Sigma}LTS(H)$ *and* $\sqsubseteq_{\mathtt{co}}$. ◇

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 the test case derivation algorithm to our running example. Let us consider again the specification $\mathcal{S}$ of Fig. 2 and the (obviously incomplete) 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.15. We can apply the algorithm in order to obtain, among others, the test case $\mathcal{U}_1$ shown in Fig. 16. 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. 17, which is also derived using the algorithm. Clearly $\mathcal{V}\,\mathcal{U}_5\mathcal{S}' =$ **fail**.

6 Relating Testing and Conformance Relations

In this section we report the major results concerning the relationship between the stuttering and the non-stuttering semantics and the relationship between the Testing Preorders (and Equivalence) and the Conformance Relation. We shall make explicit reference to HAs representing UMLSCs. In particular, in the following we shall assume that for each (HA representing a specific) UMLSC H the set of events E on which H is defined, i.e. its *alphabet*, is given explicitly. Set E will include all the events occurring in H. Under the above conditions, we will speak of UMLSC H *on* E. Moreover, in the case of specifications where the behavior of the system is only partially specified, there might be elements of E which do not occur in H.

It is worth reminding the reader here that both $\mathsf{LTS}(H)$ and ${}^{\Sigma}\mathsf{LTS}(H)$ have a finite number of states and a finite number of step-transitions. Furthermore, they are defined on *the same* set of states, namely the set Conf_H of configurations of H. In the remainder of this paper we will use the notation ${}^{\Sigma}\mathcal{C}$ for configuration $\mathcal{C}$ when we want to emphasize it being a state of ${}^{\Sigma}\mathsf{LTS}(H)$, thus avoiding confusion about which LTS we are dealing with.

6.1 Relating the stuttering and the non-stuttering semantics

In this section we take a closer look at the formal relationship between the stuttering semantics and the non-stuttering one.

Theorem 6.1
For all HAs $H = (F, E, \rho)$, $e \in E$ and $\mathcal{C} \in \mathrm{Conf}_H$ the following holds:

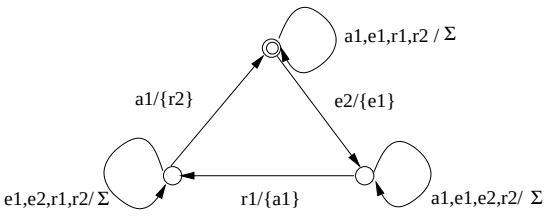

Figure 15: An implementation of the HA of Fig. 2

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

Figure 16: A test case generated from the running example

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

Figure 17: Another test case generated from the running example

i) $\forall \mathcal{C}' \in \mathrm{Conf}_H, \mathcal{E} \in \Theta_E.\ (\mathcal{C} \xrightarrow{e/\mathcal{E}} \mathcal{C}'$ iff $\mathcal{C} \xrightarrow{e/\mathcal{E}}_{\Sigma} \mathcal{C}')$

ii) $(\nexists \mathcal{C}' \in \mathrm{Conf}_H, \mathcal{E} \in \Theta_E.\ \mathcal{C} \xrightarrow{e/\mathcal{E}} \mathcal{C}')$ iff $\mathcal{C} \xrightarrow{e/\Sigma}_{\Sigma} \mathcal{C}'$ ◇

Thus the two semantics generate the same step-transitions, except for stuttering, i.e. when the machine does not accept the current event e in the current state. This refusal is modeled (a) *implicitly* in the non-stuttering semantics by not generating a step-transition at all and (b) *explicitly* in the stuttering semantics by producing Σ as output action in the step-transition. The original semantics, whose step-relation is recalled in Def. 3.3, simply generates step-transitions with the empty set as a label when stuttering. It is important to point out that in the stuttering semantics, the absence of reaction on a given input e generates stuttering—and is represented by Σ—*only* if $e \in E$. If $e \notin E$ then no transition at all is generated, in a similar way as in the non-stuttering semantics. For this reason, in this paper, the definition of a HA always includes the explicit specification of the input set E, specially when we compare different HAs on the basis of testing/conformance relations, as in the following sections. The following is a useful corollary to Theorem 6.1:

Corollary 6.1
For all HAs $H = (F, E, \rho)$, $\mathcal{C}, \mathcal{C}' \in \mathrm{Conf}_H, \gamma \in (E \times \Theta_E)^*$: $\mathcal{C} \xrightarrow{\gamma} \mathcal{C}'$ iff ${}^{\Sigma}\mathcal{C} \xrightarrow{\gamma}_{\Sigma} {}^{\Sigma}\mathcal{C}'$ ◇

We close this section with a lemma relating the languages of $\mathsf{LTS}(H)$ and ${}^{\Sigma}\mathsf{LTS}(H)$, where we use the following operator $(\_ \setminus \_)$, where $\gamma \setminus \Sigma$ is equal to γ where all occurrences of Σ are removed.

Definition 6.1 $(\gamma \setminus \Sigma)$
For $\gamma \in (E \times {}^{\Sigma}\Theta_E)$ *we define* $\gamma \setminus \Sigma$ *as follows*

$$
\gamma \setminus \Sigma = \begin{cases} \epsilon & \text{if } \gamma = \epsilon \\ \gamma' \setminus \Sigma & \text{if } \gamma = (e, \Sigma)\gamma', \text{ for some } e \in E, \gamma' \in (E \times {}^{\Sigma}\Theta_E)^* \\ (e, \mathcal{E})(\gamma' \setminus \Sigma) & \text{if } \gamma = (e, \mathcal{E})\gamma', \text{ for some } e \in E, \mathcal{E} \in \Theta_E, \gamma' \in (E \times {}^{\Sigma}\Theta_E)^* \end{cases}
$$

∘

Lemma 6.1
For HA $H = (F, E, \rho)$, *all* $\mathcal{C}, \mathcal{C}' \in \mathrm{Conf}_H$, $\gamma \in (E \times {}^{\Sigma}\Theta_E)$, *the following holds:*

i) ${}^{\Sigma}\mathcal{C} \xrightarrow{\gamma}_{\Sigma} {}^{\Sigma}\mathcal{C}'$ *implies* $\mathcal{C} \xrightarrow{\gamma\setminus\Sigma} \mathcal{C}'$

ii) $\gamma \in \mathit{lan}\ {}^{\Sigma}\mathit{LTS}(H)$ *implies* $\gamma \setminus \Sigma \in (\mathit{lan}\ {}^{\Sigma}\mathit{LTS}(H)) \cap (\mathit{lan}\ \mathit{LTS}(H))$

iii) $(\mathit{lan}\ \mathit{LTS}(H)) \subseteq (\mathit{lan}\ {}^{\Sigma}\mathit{LTS}(H))$ ◇

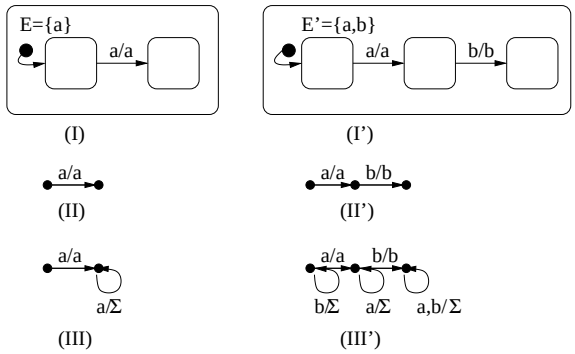

Figure 18: Example 6.1

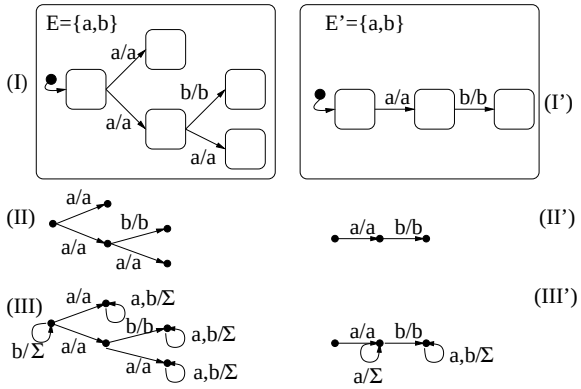

Figure 19: Example 6.2

6.2 Testing Preorders and the Conformance Relation

The following two lemmas show that, under proper conditions, the Conformance Relation is *stronger* than the *MAY* and *MUST* preorders.

Lemma 6.2
For all HAs H on E and H' on $E' \subseteq E$ the following holds: ${}^{\Sigma}LTS(H') \sqsubseteq_{\mathsf{co}} {}^{\Sigma}LTS(H)$ *implies* $LTS(H') \sqsubseteq_{\sim_{MAY}} LTS(H)$ ◇

Lemma 6.3
For all HAs H and H' on E the following holds:
${}^{\Sigma}LTS(H') \sqsubseteq_{\mathsf{co}} {}^{\Sigma}LTS(H)$ *implies* $LTS(H) \sqsubseteq_{\sim_{MUST}} LTS(H')$ ◇

Notice that Lemma 6.2 holds also for incomplete implementations ($E' \subset E$), but it requires the specification not be partial w.r.t. the implementation, which would imply $E \subset E'$. The condition $E' \subseteq E$ is indeed essential, as shown by the following example.

Example 6.1 *Let $E = \{a\} \subseteq \{a, b\} = E'$, with H (resp. H') as in Fig. 18 (I) (resp. (I')), $LTS(H)$ (resp. $LTS(H')$) as in Fig. 18 (II) (resp. (II')) and ${}^{\Sigma}LTS(H)$ (resp. ${}^{\Sigma}LTS(H')$) as in Fig. 18 (III) (resp. (III')). Clearly ${}^{\Sigma}LTS(H') \sqsubseteq_{\mathsf{co}} {}^{\Sigma}LTS(H)$, but $LTS(H') \sqsubseteq_{\sim_{MAY}} LTS(H)$ does not hold since $(a, a)(b, b) \in \mathsf{lan}\, LTS(H') \setminus \mathsf{lan}\, LTS(H)$ (see Theorem 4.1 (a)).*

Notice furthermore that in Lemma 6.2 the implication is strictly one way as shown by the following example.

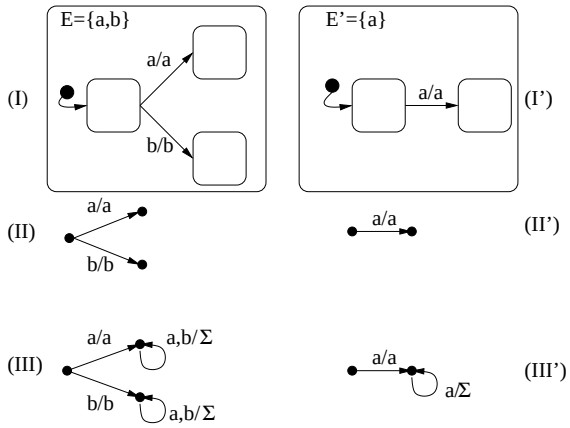

Figure 20: Example 6.4

Example 6.2 *Let $E = E' = \{a, b\}$ with H (resp. H') as in Fig. 19 (I) (resp. (I')), $LTS(H)$ (resp. $LTS(H')$) as in Fig. 19 (II) (resp. (II')) and ${}^{\Sigma}LTS(H)$ (resp. ${}^{\Sigma}LTS(H')$) as in Fig. 19 (III) (resp. (III')). We have $LTS(H') \sqsubseteq_{\sim MAY} LTS(H)$ but ${}^{\Sigma}LTS(H') \not\sqsubseteq_{\mathsf{co}} {}^{\Sigma}LTS(H)$ since $OUT\,{}^{\Sigma}LTS(H')\ (a,a)(a,\Sigma)\ b = \{b\} \not\subseteq \{\Sigma\} = OUT\,{}^{\Sigma}LTS(H)\ (a,a)(a,\Sigma)\ b$.*

Notice that in Lemma 6.3 we require that both H and H' have the same input set E. The following examples show that for H on E and H' on E' neither $E \subseteq E'$ alone nor $E' \subseteq E$ alone is enough:

Example 6.3 *Let H and H' as in Example 6.1. It is easy to see that $LTS(H) \sqsubseteq_{\sim MUST} LTS(H')$ does not hold since $(a,a)(b,b) \in \mathsf{Ian}\ LTS(H') \setminus \mathsf{Ian}\ LTS(H)$ (see Corollary B.1).*

Example 6.4 *Let $E = \{a, b\} \supseteq \{a\} = E'$ with H (resp. H') as in Fig. 20 (I) (resp. (I')), $LTS(H)$ (resp. $LTS(H')$) as in Fig. 20 (II) (resp. (II')) and ${}^{\Sigma}LTS(H)$ (resp. ${}^{\Sigma}LTS(H')$) as in Fig. 20 (III) (resp. (III')). Clearly ${}^{\Sigma}LTS(H') \sqsubseteq_{\mathsf{co}} {}^{\Sigma}LTS(H)$, but $LTS(H) \sqsubseteq_{\sim MUST} LTS(H')$ does not hold since $\mathsf{AS}\ LTS(H')\ \epsilon = \{\{(a,a)\}\}$ and $\mathsf{AS}\ LTS(H)\ \epsilon = \{\{(a,a),(b,b)\}\}$ which implies $mfs(\ \mathbf{c}\ (\mathsf{AS}\ LTS(H')\ \epsilon)) \subset\!\not\subset mfs(\ \mathbf{c}\ (\mathsf{AS}\ LTS(H)\ \epsilon))$.*

Notice furthermore that in Lemma 6.3 the implication is strictly one way as shown by the following

Example 6.5 *Let H and H' as in Example 6.2. We have $LTS(H) \sqsubseteq_{\sim MUST} LTS(H')$ but we we have seen that ${}^{\Sigma}LTS(H') \not\sqsubseteq_{\mathsf{co}} {}^{\Sigma}LTS(H)$.*

The following examples show that there is no containment relation between the testing preorder $\sqsubseteq_{\sim}$ and (the reverse of) the $\sqsubseteq_{\mathsf{co}}$ relation:

Example 6.6 *Let $E = E' = \{a, b\}$ with H (resp. H') as in Fig. 21 (I) (resp. (I')), $LTS(H)$ (resp. $LTS(H')$) as in Fig. 21 (II) (resp. (II')) and ${}^{\Sigma}LTS(H)$ (resp. ${}^{\Sigma}LTS(H')$) as in Fig. 21 (III) (resp. (III')). We have $LTS(H) \sqsubseteq_{\sim} LTS(H')$ since $LTS(H) \sqsubseteq_{\sim MAY} LTS(H')$ (actually $\mathsf{Ian}\ LTS(H) = \mathsf{Ian}\ LTS(H')$) and $LTS(H) \sqsubseteq_{\sim MUST} LTS(H')$ but ${}^{\Sigma}LTS(H') \not\sqsubseteq_{\mathsf{co}} {}^{\Sigma}LTS(H)$ since $OUT\,{}^{\Sigma}LTS(H')\ (a,a)(a,\Sigma)\ b = \{b\} \not\subseteq \{\Sigma\} = OUT\,{}^{\Sigma}LTS(H)\ (a,a)(a,\Sigma)\ b$.*

Example 6.7 *Let $E = E' = \{a, b\}$ with H (resp. H') as in Fig. 22 (I) (resp. (I')), $LTS(H)$ (resp. $LTS(H')$) as in Fig. 22 (II) (resp. (II')) and ${}^{\Sigma}LTS(H)$ (resp. ${}^{\Sigma}LTS(H')$) as in Fig. 22 (III) (resp. (III')). We have ${}^{\Sigma}LTS(H') \sqsubseteq_{\mathsf{co}} {}^{\Sigma}LTS(H)$ but $LTS(H) \sqsubseteq_{\sim} LTS(H')$ does not hold since $LTS(H) \sqsubseteq_{\sim MAY} LTS(H')$ does not hold: $(a,a)(b,b) \in \mathsf{Ian}\ LTS(H) \setminus \mathsf{Ian}\ LTS(H')$.*

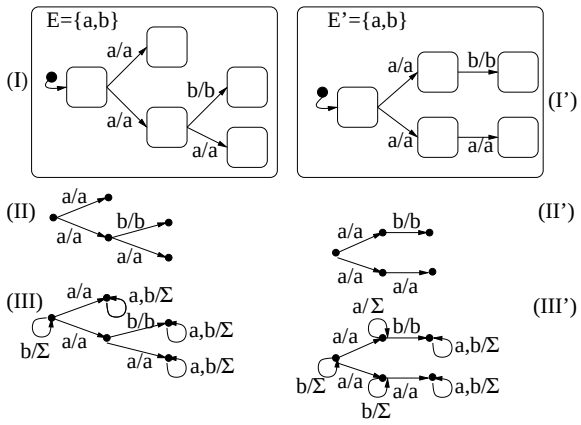

Figure 21: Example 6.6

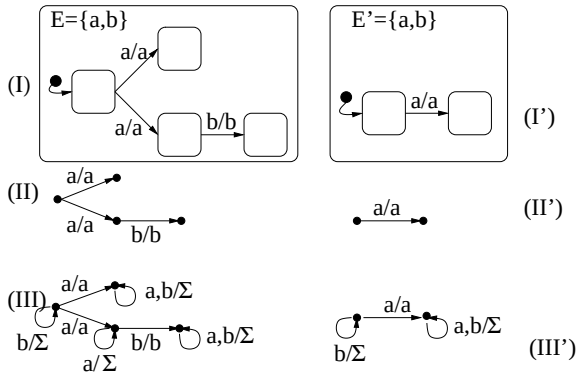

Figure 22: Example 6.7

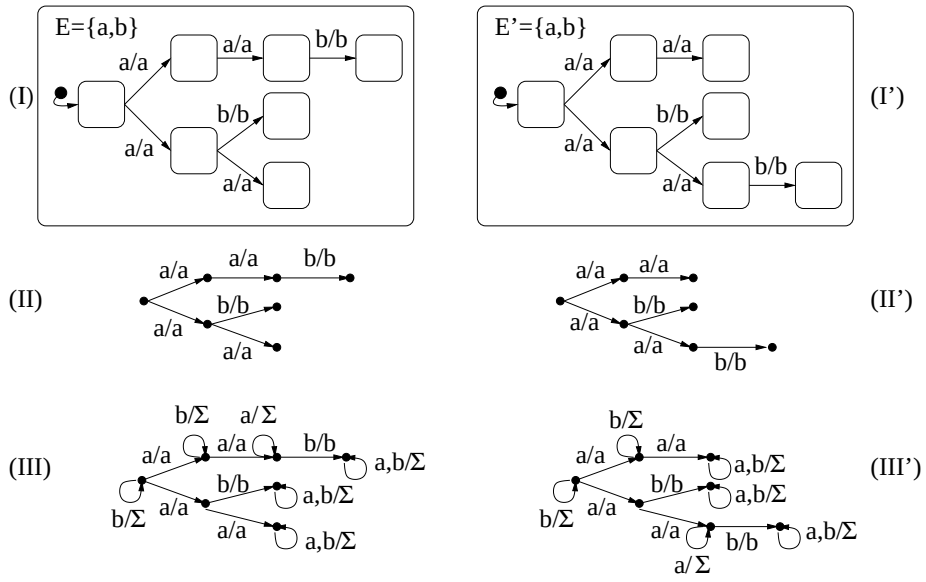

Figure 23: Example 6.8

Finally, the following two examples show that testing equivalence over the non-stuttering semantics is not strong enough for detecting LTS's capability of refusing to react and, consequently, for discriminating among them on such basis (Example 6.8). This in turn implies that testing equivalence does not enjoy substitutivity properties with respect to $\sqsubseteq_{\mathsf{co}}$ (Example 6.9).

Example 6.8 *Let $E = E' = \{a, b\}$ with H (resp. H') as in Fig. 23 (I) (resp. (I')), $LTS(H)$ (resp. $LTS(H')$) as in Fig. 23 (II) (resp. (II')) and ${}^{\Sigma}LTS(H)$ (resp. ${}^{\Sigma}LTS(H')$) as in Fig. 23 (III) (resp. (III')). We have $LTS(H) \simeq LTS(H')$ but $(a,a)(b,\Sigma)(a,a)(b,b)$ is an element of $\mathit{Ian}\,{}^{\Sigma}LTS(H) \setminus \mathit{Ian}\,{}^{\Sigma}LTS(H')$ and $(a,a)(b,\Sigma)(a,a)(b,\Sigma)$ is an element of $\mathit{Ian}\,{}^{\Sigma}LTS(H') \setminus \mathit{Ian}\,{}^{\Sigma}LTS(H)$*

Example 6.9 *Take H and H' as in Example 6.8 and let $H'' = H$. From Example 6.8 we know that $LTS(H'') \simeq LTS(H')$ and trivially ${}^{\Sigma}LTS(H'') \sqsubseteq_{\mathsf{co}} {}^{\Sigma}LTS(H)$. On the other hand ${}^{\Sigma}LTS(H') \not\sqsubseteq_{\mathsf{co}} {}^{\Sigma}LTS(H)$ since $\mathit{OUT}\,{}^{\Sigma}LTS(H')\ (a,a)(b,\Sigma)(a,a)\ b = \{\Sigma\} \not\subseteq \{b\} = \mathit{OUT}\,{}^{\Sigma}LTS(H)\ (a,a)(b,\Sigma)(a,a)\ b$. Similarly, we have that ${}^{\Sigma}LTS(H) \sqsubseteq_{\mathsf{co}} {}^{\Sigma}LTS(H'')$—trivially—but ${}^{\Sigma}LTS(H) \not\sqsubseteq_{\mathsf{co}} {}^{\Sigma}LTS(H')$ since $\mathit{OUT}\,{}^{\Sigma}LTS(H)\ (a,a)(b,\Sigma)(a,a)\ b = \{b\} \not\subseteq \{\Sigma\} = \mathit{OUT}\,{}^{\Sigma}LTS(H')\ (a,a)(b,\Sigma)(a,a)\ b$.*

The above examples show that (testing equivalence based on) the non-stuttering semantics is not *adequate* for conformance testing in the sense that one cannot replace (testing) equivalent LTSs still preserving conformance. More specifically, equivalent implementations are not conformant with the same specification. Similarly, the same implementation turns out not to be conformant to equivalent specifications. Such inadequacy comes from the fact that (the experimenters testing those LTSs generated according to) the non-stuttering semantics are unable to detect absence of reaction and to take proper actions when this happens. Due to the non-stuttering semantics, experimental systems can only deadlock in such situations.

In order to be adequate, the semantics must explicitly deal with stuttering, so that experimenters can detect absence of reaction and behave accordingly. This intuitive consideration is supported by Lemmas 6.4 and 6.5 below:

Lemma 6.4
For all HAs H on E and H' on E' the following holds:

i) ${}^{\Sigma}LTS(H) \sqsubseteq_{\simeq_{MAY}} {}^{\Sigma}LTS(H')$ implies ${}^{\Sigma}LTS(H) \sqsubseteq_{\mathsf{co}} {}^{\Sigma}LTS(H')$

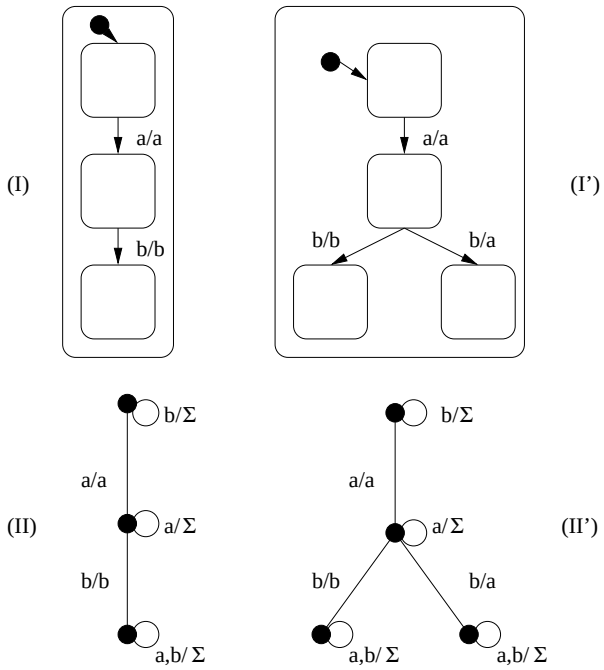

Figure 24: Example 6.11

ii) ${}^{\Sigma}LTS(H) \sqsubseteq_{\mathsf{co}} {}^{\Sigma}LTS(H')$ *and* $E \subseteq E'$ *implies* ${}^{\Sigma}LTS(H) \sqsubseteq_{\sim_{MAY}} {}^{\Sigma}LTS(H')$ ◇

Lemma 6.5
For all HAs H *on* E *and* H' *on* E' *the following holds:*
${}^{\Sigma}LTS(H) \sqsubseteq_{\sim_{MUST}} {}^{\Sigma}LTS(H')$ *implies* ${}^{\Sigma}LTS(H') \sqsubseteq_{\mathsf{co}} {}^{\Sigma}LTS(H)$ ◇

Notice that in Lemma 6.5 the implication is strictly one way as shown by the following

Example 6.10 *Let* H *and* H' *as in Example 6.1. We know from that example that* ${}^{\Sigma}LTS(H') \sqsubseteq_{\mathsf{co}} {}^{\Sigma}LTS(H)$. *On the other hand it is easy to see that* ${}^{\Sigma}LTS(H) \sqsubseteq_{\sim_{MUST}} {}^{\Sigma}LTS(H')$ *does not hold, since* $(\mathit{Ian}\,{}^{\Sigma}LTS(H')) \not\subseteq (\mathit{Ian}\,{}^{\Sigma}LTS(H))$ *and this would violate Corollary B.1.*

Finally notice that in general ${}^{\Sigma}\mathrm{LTS}(H) \sqsubseteq_{\mathsf{co}} {}^{\Sigma}\mathrm{LTS}(H')$ does *not* imply ${}^{\Sigma}\mathrm{LTS}(H) \sqsubseteq_{\sim_{MUST}} {}^{\Sigma}\mathrm{LTS}(H')$, as shown by the following

Example 6.11 *Let* $E = E' = \{a, b\}$, *with* H *(resp.* H'*) as in Fig. 24 (I) (resp. (I')) and* ${}^{\Sigma}LTS(H)$ *(resp.* ${}^{\Sigma}LTS(H')$*) as in Fig. 24 (II) (resp. (II')). Clearly* ${}^{\Sigma}LTS(H) \sqsubseteq_{\mathsf{co}} {}^{\Sigma}LTS(H')$, *but it is easy to see that* ${}^{\Sigma}LTS(H) \sqsubseteq_{\sim_{MUST}} {}^{\Sigma}LTS(H')$ *does not hold, since* $(\mathit{Ian}\,{}^{\Sigma}LTS(H')) \not\subseteq (\mathit{Ian}\,{}^{\Sigma}LTS(H))$ *and this would violate Corollary B.1.*

This last remark shows that in the stuttering semantics, the testing preorder is strictly stronger than the conformace relation.
The following theorem establishes the adequacy of the testing relations based on the stuttering semantics for the conformance relation. On an intuitive level, it is worth pointing out that point (iii) of the theorem essentially states that if an implementation is "less non-deterministic" than one conforming to a specification, then it also conforms to the specification. Similarly, point (vi) says that if a specification is "more non-deterministic" than one to which an implementation conforms, than the implementation will also conform to this specification.

Theorem 6.2
For all HAs H *on* E, H' *on* E' *and* H'' *on* E'', *with* $E' \subseteq E$, *the following holds:*

i) ${}^{\Sigma}LTS(H'') \sqsubseteq_{\sim MAY} {}^{\Sigma}LTS(H') \wedge {}^{\Sigma}LTS(H') \sqsubseteq_{\mathsf{co}} {}^{\Sigma}LTS(H)$ *implies* ${}^{\Sigma}LTS(H'') \sqsubseteq_{\mathsf{co}} {}^{\Sigma}LTS(H)$

ii) ${}^{\Sigma}LTS(H') \sqsubseteq_{\sim MUST} {}^{\Sigma}LTS(H'') \wedge {}^{\Sigma}LTS(H') \sqsubseteq_{\mathsf{co}} {}^{\Sigma}LTS(H)$ *implies* ${}^{\Sigma}LTS(H'') \sqsubseteq_{\mathsf{co}} {}^{\Sigma}LTS(H)$

iii) ${}^{\Sigma}LTS(H') \sqsubseteq_{\sim} {}^{\Sigma}LTS(H'') \wedge {}^{\Sigma}LTS(H') \sqsubseteq_{\mathsf{co}} {}^{\Sigma}LTS(H)$ *implies* ${}^{\Sigma}LTS(H'') \sqsubseteq_{\mathsf{co}} {}^{\Sigma}LTS(H)$

iv) ${}^{\Sigma}LTS(H') \sqsubseteq_{\mathsf{co}} {}^{\Sigma}LTS(H) \wedge {}^{\Sigma}LTS(H) \sqsubseteq_{\sim MAY} {}^{\Sigma}LTS(H'')$ *implies* ${}^{\Sigma}LTS(H') \sqsubseteq_{\mathsf{co}} {}^{\Sigma}LTS(H'')$

v) ${}^{\Sigma}LTS(H') \sqsubseteq_{\mathsf{co}} {}^{\Sigma}LTS(H) \wedge {}^{\Sigma}LTS(H'') \sqsubseteq_{\sim MUST} {}^{\Sigma}LTS(H)$ *implies* ${}^{\Sigma}LTS(H') \sqsubseteq_{\mathsf{co}} {}^{\Sigma}LTS(H'')$

vi) ${}^{\Sigma}LTS(H') \sqsubseteq_{\mathsf{co}} {}^{\Sigma}LTS(H) \wedge {}^{\Sigma}LTS(H'') \sqsubseteq_{\sim} {}^{\Sigma}LTS(H)$ *implies* ${}^{\Sigma}LTS(H') \sqsubseteq_{\mathsf{co}} {}^{\Sigma}LTS(H'')$ ◇

A useful corollary of the above theorem states the substitutivity properties of $\simeq$ with respect to $\sqsubseteq_{\mathsf{co}}$.

Corollary 6.2
For all HAs H on E, H' on E' and H'' on E'', with $E' \subseteq E$ the following holds:

i) ${}^{\Sigma}LTS(H') \simeq {}^{\Sigma}LTS(H'') \wedge {}^{\Sigma}LTS(H') \sqsubseteq_{\mathsf{co}} {}^{\Sigma}LTS(H)$ *implies* ${}^{\Sigma}LTS(H'') \sqsubseteq_{\mathsf{co}} {}^{\Sigma}LTS(H)$

ii) ${}^{\Sigma}LTS(H') \sqsubseteq_{\mathsf{co}} {}^{\Sigma}LTS(H) \wedge {}^{\Sigma}LTS(H'') \simeq {}^{\Sigma}LTS(H)$ *implies* ${}^{\Sigma}LTS(H') \sqsubseteq_{\mathsf{co}} {}^{\Sigma}LTS(H'')$ ◇

We close this section with the following three propositions relating the non-stuttering semantics and the stuttering one via the testing relations in the way one would expect:

Proposition 6.1
For all HAs H on E and H' on E' the following holds:
${}^{\Sigma}LTS(H) \sqsubseteq_{\sim MAY} {}^{\Sigma}LTS(H')$ *implies* $LTS(H) \sqsubseteq_{\sim MAY} LTS(H')$ ◇

Proposition 6.2
For all HAs H, H' on E the following holds:
${}^{\Sigma}LTS(H) \sqsubseteq_{\sim MUST} {}^{\Sigma}LTS(H')$ *implies* $LTS(H) \sqsubseteq_{\sim MUST} LTS(H')$ ◇

Proposition 6.3
For all HAs H, H' on E the following holds:
${}^{\Sigma}LTS(H) \sqsubseteq_{\sim} {}^{\Sigma}LTS(H')$ *implies* $LTS(H) \sqsubseteq_{\sim} LTS(H')$ ◇

Notice again that the above implications are strictly one way, as can be seen from Example 6.8, using Theorem 4.1 (a)and Corollary B.1.

7 Conclusions

The main contribution of the present paper is a theoretical framework for testing theory and verification as well as test case generation in a conformance testing setting. We presented a testing theory for UML Statecharts (UMLSCs) with an algorithm for automatic verification of testing equivalence—based on a formal "non-stuttering" semantics—and a conformance relation for UMLSCs as well as an algorithm for test case generation—based on a formal "stuttering" semantics. The automatic verification algorithm has been proved correct and the test case generation algorithm was proved complete. Both proofs are presented in this paper. The formal relationships between the stuttering and non-stuttering semantics were investigated and all related proofs provided. In particular, we proved that the non-stuttering semantics for the testing preorders is not a good choice when also conformance is an issue. In fact we showed that the conformance relation for UMLSDs is strictly stronger than the reverse of the *MUST* preorder based on the non-stuttering semantics, and then also stronger than the associated *MAY* preorder. Moreover, the testing preorder and the conformance relation are incomparable; neither one is stronger than the

other nor vice-versa. Furthermore, no substitutivity property holds: replacing an implementation conforming to a specification with a testing equivalent implementation may break conformance; symmetrically, an implementation conforming to a specification is not guaranteed to conform also to another, testing equivalent, specification. On the basis of the above negative results, we adopted a stuttering semantics also for the general testing theory. This amounts to giving experimenters the power of recognizing absence of system reaction, i.e. stuttering and behaving accordingly. We showed that, in this case, the *MAY* (resp. *MUST*) preorder is stronger than $\sqsubseteq_{\mathtt{co}}$ (resp. inverse of $\sqsubseteq_{\mathtt{co}}$). As a consequence, one can replace testing equivalent specifications and implementations still preserving their conformance relation. More specifically, if an implementation is "less non-deterministic" than one conforming to a specification, then it also conforms to the specification. Similarly, if a specification is "more non-deterministic" than one to which an implementation conforms, then the implementation will also conform to this specification. This is an important result in the framework of a system development approach in which e.g. implementations are replaced with equivalent or "more-deterministic" ones in a stepwise manner, still maintaining the conformance relation with their specifications.

Our work represents also a contribution to the field of research on relating state-based and behavioral specification. In particular, in [6] it is argued that behavioral relations, and in particular testing preorders, may form the basis for studying notions like sub-typing/sub-classes. This can be applied also to UML.

With respect to the testing equivalence verification algorithm, the determinization phase of the testing equivalence verification algorithm may take exponential time, but this should not surprise the reader because it has been proved that the verification of testing equivalence is a PSPACE-complete problem [4]. Other equivalences are easier to verify but they may be too strong, like e.g. bisimulation equivalence itself which distinguishes machines also on the basis of their internal structure and not only on the basis of their interaction with the external environment. Anyway, the fact that the semantics generate finite LTSs over i/o-pairs allows us to to perform bisimulation equivalence verification directly on such LTSs, should this turn up useful.

In order to use the test generation algorithm in practice proper *test selection* strategies are needed which will be subject of future work. Some work on test selection in a formal test derivation framework is already present in the literature (see, e.g. [7, 2, 12]), 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 [10]. Closely related 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 [37] and TGV/AGEDIS [39], and one of the next steps will be an investigation on the possibility of providing a connection between our work and 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 results presented in the present paper to UML specifications consisting of *collections* of UMLSCs interacting via queues [15], 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 developing a suitable extension of testing theory to the distributed case.

A further 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 [20]. Of course (infinite) data sets pose further problems in the test selection procedures.

The results addressed in the present paper have been originally proposed in [25, 26, 16], although in isolation, while in the present paper they have been dealt with in a uniform framework and notation. Moreover all proofs, which were omitted in the above mentioned papers, are provided in the present paper.

A Hierarchical Automata

The first step of our approach is a purely syntactical one and consists in translating UMLSCs into what is usually called HAs. HAs can be seen as an *abstract syntax* for UMLSCs in the sense that they abstract from the purely syntactical/graphical details and describe only the essential aspects of the statechart. They are composed of simple sequential automata related by a *refinement function*. A state is mapped via the refinement function into the set of (parallel) automata which refine it. The translation from UMLSCs to HAs has been dealt with in [24]. In the sequel we will be concerned only with HAs. In this section we recall the notion of HAs as defined in [32, 24].

A.1 Basic definitions and Semantics

The first notion is that of a (sequential) automaton

Definition A.1 (Sequential Automata)
A sequential automaton A is a 4-tuple $(\sigma_A, s^0_A, \lambda_A, \delta_A)$ where σ_A is a finite set of states *with $s^0_A \in \sigma_A$ the* initial state*, λ_A is a finite set of* transition labels*, with $\lambda_A \cap \sigma_A = \emptyset$ and $\delta_A \subseteq \sigma_A \times \lambda_A \times \sigma_A$ is the* transition relation. ∘

In the context of HAs, the labels in λ_A have a particular structure. For transition t we require its label to be a 5-tuple (sr, ev, g, ac, td), where sr is the *source restriction*, ev is the *trigger* event, g is the *guard*, ac is the *actions* list and td is the *target determinator*. In the sequel we use the following functions SRC, TGT, SR, EV, G, AC, TD, defined in the obvious way; for transition $t = (s, (sr, ev, g, ac, td), s')$, $SRC\ t = s, TGT\ t = s', SR\ t = sr, EV\ t = ev, G\ t = g, AC\ t = ac, TD\ t = td$. Their meaning is described in [24]. Hierarchical Automata are defined as follows:

Definition A.2 (Hierarchical Automata)
A HA H is a tuple (F, E, ρ), where F is a finite set of sequential automata with mutually disjoint sets of states, i.e. $\forall A_1, A_2 \in F.\ \sigma_{A_1} \cap \sigma_{A_2} = \emptyset$ and E is a finite set of events*; the* refinement function *$\rho : \bigcup_{A \in F} \sigma_A \mapsto 2^F$ imposes a tree structure to F, i.e. (i) there exists a unique root automaton $A_{root} \in F$ such that $A_{root} \notin \bigcup rng\ \rho$, (ii) every non-root automaton has exactly one ancestor state: $\bigcup rng\ \rho = F \setminus \{A_{root}\}$ and $\forall A \in F \setminus \{A_{root}\}.\ \exists_1 s \in \bigcup_{A' \in F \setminus \{A\}} \sigma_{A'}.\ A \in (\rho\ s)$ and (iii) there are no cycles: $\forall S \subseteq \bigcup_{A \in F} \sigma_A.\ \exists s \in S.\ S \cap \bigcup_{A \in \rho s} \sigma_A = \emptyset$.* ∘

We say that a state s for which $\rho\ s = \emptyset$ holds is a *basic* state.

The notion of *conflict* between transitions needs to be extended in order to deal with state hierarchy. When transitions t and t' are in conflict we write $t \# t'$. The complete formal definition of conflict for HAs can be found in [24, 15] where also the notion of *priority* for (conflicting) transitions is defined. Intuitively transitions coming from deeper states have higher priority. For the purposes of the present paper it is sufficient to say that priorities form a partial order. We let πt denote the priority of transition t and $\pi t \sqsubseteq \pi t'$ mean that t has lower priority than (the same priority as) t'. In the sequel we will be concerned only with HAs.

In the sequel we implicitly make reference to a generic HA $H = (F, E, \rho)$. Moreover, we also assume implicitly that each transition of each sequential automaton in F is uniquely identified by its label. This can always be obtained by adding unique identifiers to labels whenever necessary. The following definition characterizes a couple of useful functions:

Definition A.3
For $A \in F$ the automata *and* transitions under *A are defined respectively as*
$\mathsf{Au}\ A = \{A\} \cup (\bigcup_{A' \in \left(\bigcup_{s \in \sigma_A} (\rho_A s)\right)} (\mathsf{Au}\ A'))$, $\mathsf{Tr}\ A = \bigcup_{A' \in \mathsf{Au}\ A} \delta_{A'}$. ∘

In the remainder of this section we will deal with the UML semantics of HAs.
A *configuration* denotes a global state of a HA, composed of local states of component sequential automata:

Definition A.4 (Configurations)
A configuration *of H is a set $\mathcal{C} \subseteq \bigcup_{A \in F} \sigma_A$ such that (i) $\exists_1 s \in \sigma_{A_{root}}.\ s \in \mathcal{C}$ and (ii) $\forall s, A.\ s \in \mathcal{C} \wedge A \in \rho\ s \Rightarrow \exists_1 s' \in A.\ s' \in \mathcal{C}$.* ∘

The set of all configurations of H is denoted by Conf_H, while $\mathcal{C}_{\mathrm{in}}$ denotes its *initial* configuration, namely the configuration composed only by initial states.

The operational semantics of a HA is defined as a LTS, where the states are the configuration/ input-queue pairs of the associated UMLSC and the transitions are characterized by the *step*-relation. Each transition of the LTS is labeled by the set of (unique identifiers of the) transitions of the associated UMLSC which have been fired in the step.

While in classical statecharts the external environment is modeled by a set, in the definition of UML statecharts, the nature of the input-queue of a statechart is not specified; in particular, the management policy of such a queue is not defined. In our overall approach to UMLSCs semantics definition, we choose *not* to fix any particular semantics, such as set, or multi-set or FIFO-queue etc., but to model the input queue in a policy-independent way, freely using a notion of abstract data types. In the following we assume that for set D, Θ_D denotes the class of all structures of a certain kind (like FIFO queues, or multi-sets, or sets) over D and we assume to have basic operations for manipulating such structures. In particular, for $\mathcal{D}, \mathcal{D}'$, etc. in Θ_D, and $d \in D$, $(\mathsf{Add}\ \mathcal{D}\ d)$ denotes the structure obtained by adding d to structure $\mathcal{D}$. Similarly, $(\mathsf{Join}\ \mathcal{D}\ \mathcal{D}')$ denotes the structure obtained by merging $\mathcal{D}$ with $\mathcal{D}'$. The predicate $\mathsf{is\_join}^n_{j=1} \mathcal{D}_j\ \mathcal{I}$ states that $\mathcal{I}$ is a possible *join* of $\mathcal{D}_1 \ldots \mathcal{D}_n$ and it is a way for expressing non-deterministic merge of $\mathcal{D}_1 \ldots \mathcal{D}_n$. The predicate $(\mathsf{Sel}\ \mathcal{D}\ d\ \mathcal{D}')$ states that $\mathcal{D}'$ is the structure resulting from selecting d from $\mathcal{D}$; of course, the selection policy depends on the choice for the particular semantics. We assume that if $\mathcal{D}$ is the empty structure, denoted by $\langle\rangle$, then $(\mathsf{Sel}\ \mathcal{D}\ d\ \mathcal{D}')$ yields FALSE for all d and $\mathcal{D}'$. Finally, given sequence $r \in D^*$, $(new\ r)$ is the structure containing the elements of r (again, the existence and nature of any relation among the elements of $(new\ r)$ depends on the semantics of the particular structure).

Definition A.5 (Operational semantics)
The operational semantics of an HA $H = (F, E, \rho)$ is the LTS over 2^{Tr_H} $(\mathrm{Conf}_H \times \Theta_E, (\mathcal{C}_{in}, \mathcal{E}_0), \rightarrow)$ where (i) $\mathrm{Conf}_H \times \Theta_E$ is the set of statuses, (ii) $(\mathcal{C}_{in}, \mathcal{E}_0) \in \mathrm{Conf}_H \times \Theta_E$ is the initial status, with $\mathcal{C}_{in}$ the configuration composed only of initial states of automata in F and $\mathcal{E}_0$ the given initial input queue, (iii) $\rightarrow \subseteq (\mathrm{Conf}_H \times \Theta_E) \times 2^{\mathsf{Tr}_H} \times (\mathrm{Conf}_H \times \Theta_E)$ is the step*-transition relation defined below.* ∘

As usual, we write $(\mathcal{C}, \mathcal{E}) \xrightarrow{\mathcal{L}} (\mathcal{C}', \mathcal{E}')$ for $(\mathcal{C}, \mathcal{E}), \mathcal{L}, (\mathcal{C}', \mathcal{E}')) \in \rightarrow$. Any such transition denotes the result of firing a maximal set $\mathcal{L}$ of non-conflicting transitions of the sequential automata of H which respect priorities. Relation $\rightarrow$ is the smallest relation which satisfies the rule below:

Definition A.6 (Transition Deduction System)

$$\frac{(\mathsf{Sel}\ \mathcal{E}\ e\ \mathcal{E}'') \qquad H \uparrow \emptyset :: (\mathcal{C}, \{e\}) \xrightarrow{\mathcal{L}} (\mathcal{C}', \mathcal{E}')}{(\mathcal{C}, \mathcal{E}) \xrightarrow{\mathcal{L}} (\mathcal{C}', (\mathsf{Join}\ \mathcal{E}''\mathcal{E}'))}$$

∘

In the above rule we make use of an auxiliary relation, namely $A \uparrow P :: (\mathcal{C}, \mathcal{E}) \xrightarrow{\mathcal{L}} (\mathcal{C}', \mathcal{E}')$. Such a relation, which is defined by the deduction system proposed in [24] and recalled in Fig. 25, models labeled transitions of the semantics of HA A under specific constraints P related to transition priority. $\mathcal{L}$ is the set containing the transitions of the sequential automata of A which are selected

Progress Rule

$$\frac{t \in LE_A\ \mathcal{C}\ \mathcal{E} \quad \nexists t' \in P \cup E_A\ \mathcal{C}\ \mathcal{E}.\ \pi\ t \sqsubset \pi\ t'}{A \uparrow P :: (\mathcal{C}, \mathcal{E}) \xrightarrow{\{t\}} (DEST\ t, new(ACt))}$$

Stuttering Rule

$$\frac{\{s\} = \mathcal{C} \cap \sigma_A \quad \rho_A\ s = \emptyset \quad \forall t \in LE_A\ \mathcal{C}\ \mathcal{E}.\ \exists t' \in P.\ \pi t \sqsubset \pi t'}{A \uparrow P :: (\mathcal{C}, \mathcal{E}) \xrightarrow{\emptyset} (\{s\}, \langle\rangle)}$$

Composition Rule

$$\frac{\begin{array}{l}\{s\} = \mathcal{C} \cap \sigma_A \\ \rho_A\ s = \{A_1, \ldots, A_n\} \neq \emptyset \\ \left(\bigwedge_{j=1}^{n} A_j \uparrow (P \cup LE_A\ \mathcal{C}\ \mathcal{E}) :: (\mathcal{C}, \mathcal{E}) \xrightarrow{\mathcal{L}_j} (\mathcal{C}_j, \mathcal{E}_j)\right) \wedge \mathsf{is\_join}_{j=1}^{n}\ \mathcal{E}_j\ \mathcal{I} \\ \left(\bigcup_{j=1}^{n} \mathcal{L}_j = \emptyset\right) \Rightarrow (\forall t \in LE_A\ \mathcal{C}\ \mathcal{E}.\ \exists t' \in P.\ \pi t \sqsubset \pi t')\end{array}}{A \uparrow P :: (\mathcal{C}, \mathcal{E}) \xrightarrow{\bigcup_{j=1}^{n} \mathcal{L}_j} (\{s\} \cup \bigcup_{j=1}^{n} \mathcal{C}_j, \mathcal{I})}$$

where

i) $LE_A\,\mathcal{C}\,\mathcal{E} =_{df} \{t \in \delta_A \mid \{(SRC\,t)\} \cup (SR\,t) \subseteq \mathcal{C} \ \wedge (EV\,t) \in \mathcal{E} \wedge (\mathcal{C}, \mathcal{E}) \models (G\,t)\}$; $(\mathcal{C}, \mathcal{E}) \models (G\,t)$ formalizes that guard $(G\,t)$ is true of $(\mathcal{C}, \mathcal{E})$

ii) $E_A\,\mathcal{C}\,\mathcal{E} =_{df} \bigcup_{A' \in (\mathsf{Au}\ A)} LE_{A'}\,\mathcal{C}\,\mathcal{E}$

iii) $(DEST\ t) =_{df} \{s \mid \exists s' \in (TD\ t).\ (TGT\ t) \preceq s \preceq s'\}$

iv) $\preceq$ is the state-nesting partial order and $\sqsubseteq$ is the priority partial order based on the priority mapping π: $\pi t \sqsubseteq \pi t'$ means that the priority of t is smaller than or equal to that of t'.

Figure 25: Core Semantics of UML Hierarchical Automata

to fire when the current configuration (resp. input) is $\mathcal{C}$ (resp. $\mathcal{E}$) and the firing of which brings to configuration (resp. output events) $\mathcal{C}'$ (resp. $\mathcal{E}'$).

Several proofs we give in this section are carried out by induction either on the length d of the derivation for proving $A \uparrow P :: (\mathcal{C}, \mathcal{E}) \xrightarrow{\mathcal{L}} (\mathcal{C}', \mathcal{E}')$ [33] or on the structure of the subset of F affected by $\mathcal{C}$. With respect to structural induction, let $F_{\mathcal{C}}$ be the set $\{A \in F \mid \mathcal{C} \cap \sigma_A \neq \emptyset\}$. It is easy to define a relation on $F_{\mathcal{C}}$ such that X is related to Y iff $s \in \mathcal{C} \cap \sigma_Y$ and $X \in (\rho_Y s)$. Notice that since $\mathcal{C}$ is a configuration, for each A in $F_{\mathcal{C}}$ there is a unique state $s \in \sigma_A \cap \mathcal{C}$. The transitive and reflexive closure of such a relation is a well-founded partial order, since antisymmetry is a consequence of property (iii) in the definition of HAs; the bottom elements are those X such that $\rho\sigma_X = \emptyset$ [30].

The following lemma gives some insights on the core semantics.

Lemma A.1
For all HA $H = (F, E, \rho), A \in F, P \subseteq (\mathsf{Tr}\,H), \mathcal{E} \in \Theta_E, \mathcal{C} \in \mathrm{Conf}_H$, *s.t.* $\sigma_A \cap \mathcal{C} \neq \emptyset$ *the following holds:*

i) $\exists \mathcal{L} \subseteq (\mathsf{Tr}\,H), \mathcal{C}' \in \mathrm{Conf}_H, \mathcal{E}' \in \Theta_E.\ A \uparrow P :: (\mathcal{C}, \mathcal{E}) \xrightarrow{\mathcal{L}} (\mathcal{C}', \mathcal{E}')$

ii) $A \uparrow P :: (\mathcal{C}, \mathcal{E}) \xrightarrow{\emptyset} (\mathcal{C}', \mathcal{E}')$ *and* $\mathcal{C} \in \mathrm{Conf}_A$ *implies* $\mathcal{C}' = \mathcal{C}$ *and* $\mathcal{E}' = \langle\rangle$

iii) $A \uparrow P :: (\mathcal{C}, \mathcal{E}) \xrightarrow{\emptyset} (\mathcal{C}, \langle\rangle)$ *implies* $\nexists \mathcal{L} \neq \emptyset, \mathcal{C}' \in \mathrm{Conf}_H, \mathcal{E}' \in \Theta_E.\ A \uparrow P :: (\mathcal{C}, \mathcal{E}) \xrightarrow{\mathcal{L}} (\mathcal{C}', \mathcal{E}')$ $\diamond$

Proof

Part i -
By induction on the structure of $F_{\mathcal{C}}$.

Base case ($\bigcup_{X \in \rho_A\ \sigma_A} X = \emptyset$):
If $\forall t \in LE_A\ \mathcal{C}\ \mathcal{E}.\ \exists t' \in P.\ \pi t \sqsubset \pi t'$ then the Stuttering Rule can be applied immediately to prove

the assert. If instead $\exists t \in LE_A\ \mathcal{C}\ \mathcal{E}.\ \ \nexists t' \in P.\ \pi\ t \sqsubset \pi\ t'$ then the Progress Rule can be applied immediately to prove the assert.

Induction step ($\bigcup_{X \in \rho_A\ \sigma_A} X \neq \emptyset$):
If $\exists t \in LE_A\ \mathcal{C}\ \mathcal{E}.\ \ \nexists t' \in P \cup E_A\ \mathcal{C}\ \mathcal{E}.\ \pi t \sqsubset \pi t'$ then the Progress Rule can be applied immediately to prove the assert. Suppose then that $\forall t \in LE_A\ \mathcal{C}\ \mathcal{E}.\ \exists t' \in P \cup E_A\ \mathcal{C}\ \mathcal{E}.\ \pi t \sqsubset \pi t'$. By the Induction Hypothesis we know that $A_j \uparrow P \cup (LE_A\ \mathcal{C}\ \mathcal{E}) :: (\mathcal{C}, \mathcal{E}) \xrightarrow{\mathcal{L}_j} (\mathcal{C}_j, \mathcal{E}_j)$ for $j = 1, \ldots, n$ where $\rho_A s = \{A_1, \ldots, A_n\}$ with $\{s\} = \mathcal{C} \cap \sigma_A$, so that we can apply the Composition Rule for getting the assert. To that purpose we must be sure that also the fourth premise of the Composition Rule is fulfilled, which we show in the sequel. From the hypothesis $\forall t \in LE_A\ \mathcal{C}\ \mathcal{E}.\ \exists t' \in P \cup E_A\ \mathcal{C}\ \mathcal{E}.\ \pi t \sqsubset \pi t'$, using set theory, Lemma A.2 below, and noting that δ_A is finite, we can first of all conclude

$$\forall t \in LE_A\ \mathcal{C}\ \mathcal{E}.\ \exists t' \in P \cup \left(\bigcup_{j=1\ldots n} E_{A_j}\ \mathcal{C}\ \mathcal{E}\right).\ \pi t \sqsubset \pi t' \quad \text{(I)}$$

Moreover, if $\bigcup_{j=1\ldots n} \mathcal{L}_j = \emptyset$, by Lemma A.3 below we get also

$$\forall t \in LE_{A_j}\ \mathcal{C}\ \mathcal{E}.\ \exists t' \in P \cup LE_A\ \mathcal{C}\ \mathcal{E}.\ \pi t \sqsubset \pi t' \quad \text{(II)}$$

Since all sets involved are finite, by combining (I) and (II) above together we get, from set theory, the fourth premise of the Composition Rule.

Part ii -
By derivation induction.

Base case ($d = 1$):
In this case only the Stuttering Rule could have been applied. Moreover $\mathcal{C} \in \mathrm{Conf}_A$ and $\{s\} = \sigma_A \cap \mathcal{C}$ implies $\mathcal{C} = \{s\}$.

Induction step ($d > 1$):
In this case the Composition Rule must have been applied in the last step of the derivation.

$\mathcal{L} = \emptyset$

$\Rightarrow$ {Def. of Composition Rule}

$\bigwedge_{j=1,\ldots,n} A_j \uparrow P \cup LE_A\ \mathcal{C}\ \mathcal{E} :: (\mathcal{C}, \mathcal{E}) \xrightarrow{\emptyset} (\mathcal{C}'_j, \mathcal{E}'_j)$

$\Rightarrow$ {$\mathcal{C} \in \mathrm{Conf}_A, s \in \mathcal{C} \cap \sigma_A$, Def. A.4 implies $\mathcal{C} = \{s\} \cup \left(\bigcup_{j=1,\ldots,n} \mathcal{C}_j\right)$ with $\mathcal{C}_j \in \mathrm{Conf}_{A_j}$; Lemma A.4 below}

$\bigwedge_{j=1,\ldots,n} A_j \uparrow P \cup LE_A\ \mathcal{C}\ \mathcal{E} :: (\mathcal{C}_j, \mathcal{E}) \xrightarrow{\emptyset} (\mathcal{C}''_j, \mathcal{E}''_j)$

$\Rightarrow$ {Induction Hypothesis}

$\bigwedge_{j=1,\ldots,n} \mathcal{C}_j = \mathcal{C}''_j \wedge \mathcal{E}''_j = \langle\rangle$

$\Rightarrow$ {Def. of Composition Rule}

$\mathcal{C}' = \mathcal{C} \wedge \mathcal{E}' = \langle\rangle$

Part iii -
By derivation induction.

Base case ($d = 1$):
In this case only the Stuttering Rule could have been applied, which trivially proves the assert.

Induction step ($d > 1$):
In this case only the Composition Rule could have been applied in the last step of the derivation.

In this case, $\mathcal{L} = \emptyset$ implies $A_j \uparrow (P \cup LE_A\ \mathcal{C}\ \mathcal{E}) :: (\mathcal{C}, \mathcal{E}) \xrightarrow{\emptyset} (\mathcal{C}_j, \mathcal{E}_j)$ for $j = 1, \ldots, n$, which, for the Induction Hypothesis brings to $\not\exists \mathcal{L}_j \neq \emptyset.\ A_j \uparrow (P \cup LE_A\ \mathcal{C}\ \mathcal{E}) :: (\mathcal{C}, \mathcal{E}) \xrightarrow{\mathcal{L}_j}$ for $j = 1, \ldots, n$. Thus the Composition Rule cannot be applied in any other way for producing a transition $A \uparrow P :: (\mathcal{C}, \mathcal{E}) \xrightarrow{\mathcal{L}} (\mathcal{C}', \mathcal{E}')$ with $\mathcal{L} \neq \emptyset$. □

In the above proof we used the following lemmas:

Lemma A.2
For all HA $H = (F, E, \rho), A \in F, \mathcal{E} \in \Theta_E, \mathcal{C} \in \mathrm{Conf}_H$ *where* $\sigma_A \cap \mathcal{C} = \{s\}$ *and* $\rho_A s = \{A_1, \ldots, A_n\}$ *the following holds:* $E_A\ \mathcal{C}\ \mathcal{E} = LE_A\ \mathcal{C}\ \mathcal{E} \cup \left(\bigcup_{j=1,\ldots,n} E_{A_j}\ \mathcal{C}\ \mathcal{E}\right)$ ◇

Proof

By induction on the structure of $F_{\mathcal{C}}$.

Base case ($\bigcup_{X \in \rho_A\ \sigma_A} X = \emptyset$):
Trivially $E_A\ \mathcal{C}\ \mathcal{E} = LE_A\ \mathcal{C}\ \mathcal{E}$ if $\rho_A\ s = \emptyset$

Induction step ($\bigcup_{X \in \rho_A\ \sigma_A} X \neq \emptyset$):

$E_A\ \mathcal{C}\ \mathcal{E}$

$=$ {Def. of E_A}

$\bigcup_{A' \in (\mathsf{Au}\ A)} LE_{A'}\ \mathcal{C}\ \mathcal{E}$

$=$ {Def. of Au }

$\bigcup_{A' \in \{A\} \cup \left(\bigcup_{A'' \in (\rho_A \sigma_A)} \mathsf{Au}\ A''\right)} LE_{A'}\ \mathcal{C}\ \mathcal{E}$

$=$ {Def. of σ_A and $\rho_A s$}

$\bigcup_{A' \in \{A\} \cup (\bigcup_{j=1,\ldots,n} \mathsf{Au}\ A_j) \cup \left(\bigcup_{A'' \in (\rho_A \{s' \in \sigma_A \mid s' \neq s\})} \mathsf{Au}\ A''\right)} LE_{A'}\ \mathcal{C}\ \mathcal{E}$

$=$ {$s' \in \sigma_A$ with $s' \neq s$ and $A'' \in \rho_a s'$ implies $LE_{A''}\ \mathcal{C}\ \mathcal{E} = \emptyset$}

$\bigcup_{A' \in \{A\} \cup (\bigcup_{j=1,\ldots,n} \mathsf{Au}\ A_j)} LE_{A'}\ \mathcal{C}\ \mathcal{E}$

$=$ {Set Theory}

$LE_A\ \mathcal{C}\ \mathcal{E} \cup \left(\bigcup_{j=1,\ldots,n} \bigcup_{A' \in \mathsf{Au}\ A_j} LE_{A'}\ \mathcal{C}\ \mathcal{E}\right)$

$=$ {Def. of E_A}

$LE_A\ \mathcal{C}\ \mathcal{E} \cup \left(\bigcup_{j=1,\ldots,n} E_{A_j}\ \mathcal{C}\ \mathcal{E}\right)$ □

Lemma A.3
For all HA $H = (F, E, \rho), A \in F, P \subseteq \mathit{Tr}\ H, \mathcal{E} \in \Theta_E, \mathcal{C} \in \mathrm{Conf}_H$ *the following holds:*
$A \uparrow P :: (\mathcal{C}, \mathcal{E}) \xrightarrow{\emptyset}$ *implies* $\forall t \in E_A\ \mathcal{C}\ \mathcal{E}.\ \exists t' \in P.\ \pi t \sqsubset \pi t'$ ◇

Proof

By derivation induction.

Base case ($d = 1$):
In this case only the Stuttering Rule could have been applied and the assert follows trivially.

Induction step ($d > 1$):
In this case the Composition Rule must have been applied in the last step of the derivation. Since $\bigcup_{j=1\ldots n} \mathcal{L}_j = \emptyset$, we know

$$\forall t \in LE_A\ \mathcal{C}\ \mathcal{E}.\ \exists t' \in P.\ \pi t \sqsubset \pi t' \quad \text{(I)}$$

by the fourth premise of the Composition Rule, and

$$\forall t \in E_{A_j}\ \mathcal{C}\ \mathcal{E}.\ \exists t' \in P \cup LE_A\ \mathcal{C}\ \mathcal{E}.\ \pi t \sqsubset \pi t' \quad \text{(II)}$$

for $j = 1, \ldots, n$ because of the Induction Hypothesis. By using Lemma A.2 and I and II above we get the assert. □

Lemma A.4
For all HA $H = (F, E, \rho), A \in F, P \subseteq (Tr\ H), \mathcal{E} \in \Theta_E, \mathcal{C} \in \mathrm{Conf}_H$, *s.t.* $\sigma_A \cap \mathcal{C} \neq \emptyset$, *the following holds: if* $A \uparrow P :: (\mathcal{C}, \mathcal{E}) \xrightarrow{\emptyset}$, $\mathcal{C}' \in \mathrm{Conf}_A$ *and* $\mathcal{C}' \subseteq \mathcal{C}$ *then* $A \uparrow P :: (\mathcal{C}', \mathcal{E}) \xrightarrow{\emptyset}$ ◇

Proof

We proceed derivation induction.
Base case ($d = 1$):
If the derivation has length 1, then only the Stuttering Rule could have been applied. Moreover $\mathcal{C}' \in \mathrm{Conf}_A$, $\mathcal{C}' \subseteq \mathcal{C}$ and $\{s\} = \sigma_A \cap \mathcal{C}$ implies, in this case, $\mathcal{C}' = \{s\}$. Thus the assert follows.
Induction step ($d > 1$):
In this case the Composition Rule must have been applied in the last step of the derivation, which, together with $\mathcal{C}' \in \mathrm{Conf}_A$ and $\mathcal{C}' \subseteq \mathcal{C}$, implies $\mathcal{C}' = \{s\} \cup \left(\bigcup_{j=1,\ldots,n} \mathcal{C}'_j\right)$ where $\mathcal{C}'_j \in \mathrm{Conf}_{A_j}$ for $j = 1, \ldots, n$. We know, $\bigwedge_{j=1,\ldots,n} A_j \uparrow P \cup LE_A\ \mathcal{C}\ \mathcal{E} :: (\mathcal{C}, \mathcal{E}) \xrightarrow{\emptyset}$ and obviously $\mathcal{C}'_j \subseteq \mathcal{C}$. So, by Induction Hypothesis, we get $A_j \uparrow P \cup LE_A\ \mathcal{C}\ \mathcal{E} :: (\mathcal{C}'_j, \mathcal{E}) \xrightarrow{\emptyset}$. Moreover, we note that $\forall t \in LE_A\ \mathcal{C}\ \mathcal{E}.\ \exists t' \in P.\ \pi t \sqsubset \pi t' \wedge \mathcal{C}' \subseteq \mathcal{C}$ implies $\forall t \in LE_A\ \mathcal{C}'\ \mathcal{E}.\ \exists t' \in P.\ \pi t \sqsubset \pi t'$. Thus, by applying the Composition Rule we get $A \uparrow P :: (\mathcal{C}', \mathcal{E}) \xrightarrow{\emptyset}$ that is, the assert. □

B Detailed Proofs

B.1 Proofs related to Sect. 4

Proof of Proposition 4.1

$$\exists \mathcal{L}.\ \mathcal{L} \neq \emptyset\ \wedge\ (\mathcal{C}, \mathcal{E}) \xrightarrow{\mathcal{L}} (\mathcal{C}', (join\ \mathcal{E}''\ \mathcal{E}'))$$

$\Leftrightarrow$ {Def. of $\xrightarrow{\mathcal{L}}$ (see Def. 3.3)}

$$\exists e \in E, \mathcal{L} \neq \emptyset.\ H \uparrow \emptyset :: (\mathcal{C}, \{e\}) \xrightarrow{\mathcal{L}} (\mathcal{C}', \mathcal{E}') \wedge (\mathsf{Sel}\ \mathcal{E}\ e\ \mathcal{E}'')$$

$\Leftrightarrow$ {Def. 4.2, Def. of $Queue(\mathcal{F})$}

$$\exists e \in E.\ \mathcal{C} \xrightarrow{(e,\mathcal{E}')} \mathcal{C}' \wedge Queue(\mathcal{E}) \xrightarrow{e} \lambda X.Queue(join\ \mathcal{E}''\ X)$$

$\Leftrightarrow$ {Def. 3.5, $Queue(\mathcal{F})$ does not perform silent moves}

$$(Queue(\mathcal{E})\ ||\ \mathcal{C}) \rightsquigarrow (Queue(join\ \mathcal{E}''\ \mathcal{E}')\ ||\ \mathcal{C}')$$ □

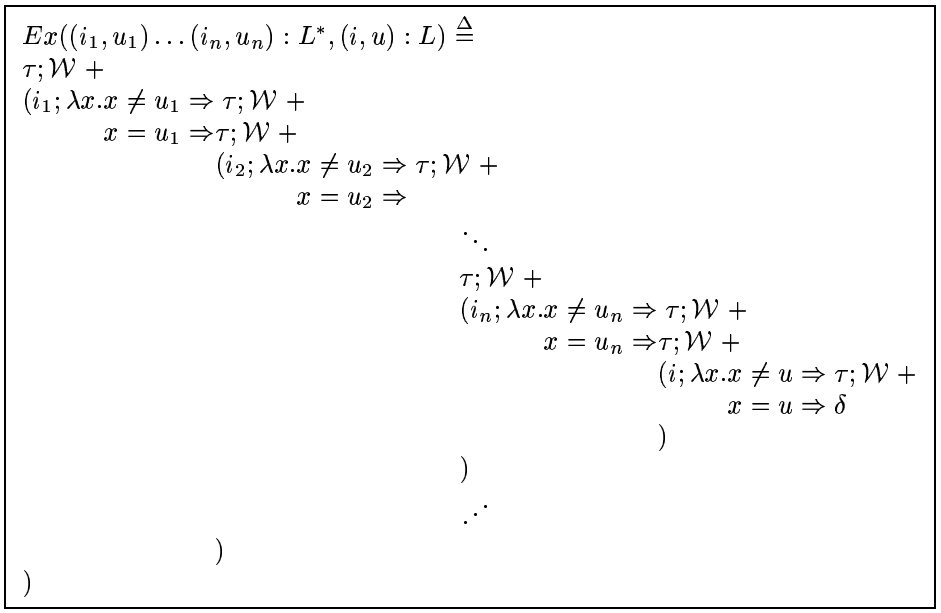

Figure 26: Definition of $Ex((i_1,u_1)\ldots(i_n,u_n),(i,u))$

Proof of Theorem 4.1

In order to prove Theorem 4.1, we first need some auxiliary notions and results. First of all, it is convenient to define the experimenter $\mathcal{W}$ which always experiences *success*, i.e. $\mathcal{W} \triangleq \mathbf{W};\delta$. We now define a particular class of experimenters. The first kind of such experimenters is that of those which can fail only if the LTS they are experimenting with, after having performed the input/output sequence $\gamma = (i_1,u_1)(i_2,u_2)\ldots(i_n,u_n)$, will react with output u on input i. The definition of such an experimenter $Ex(\gamma,(i,u))$ is given in Fig.26. The second kind of experimenters of interest is that of those which can fail only if the LTS under test, after having performed the input/output sequence γ as before cannot accept any input from finite set $I = \{i'_1,\ldots i'_k\}$. The definition of such an experimenter $Ex(\gamma,I)$ is given in Fig.27. We let EX denote the set of all the experimenters of the kind $Ex(\gamma,(i,u))$ or $Ex(\gamma,I)$. Moreover, in the sequel, for $A \subseteq L$ we let $(IN\ A)$ denote the set $\{i \mid (i,u) \in A\}$.

Finally, a third kind of experimenters is used, $Ex(\gamma)$, which succeeds only if the system under test can perform γ, for $\gamma = (i_1,u_1)\ldots(i_n,u_n)$. The definition for $Ex(\gamma)$ shown in Fig.28.

The proof of Theorem 4.1 follows:

Part a)

We first show that if $\mathcal{S} \sqsubseteq_{\sim MAY} \mathcal{S}'$ then $\mathcal{S} <<_{MAY} \mathcal{S}'$. Suppose $\mathcal{S} <<_{MAY} \mathcal{S}'$ does not hold; then we can set up the following derivation,

$\mathcal{S} <<_{MAY} \mathcal{S}' \equiv$ FALSE

$\Rightarrow$ {Def. of $<<_{MAY}$ }

$\mathsf{lan}\ \mathcal{S} \not\subseteq \mathsf{lan}\ \mathcal{S}'$

$\Rightarrow$ {Set theory}

$\exists \gamma \in \mathsf{lan}\ \mathcal{S}.\ \gamma \notin \mathsf{lan}\ \mathcal{S}'$

$\Rightarrow$ {Def. of $Ex(\gamma)$, Def. 3.7}

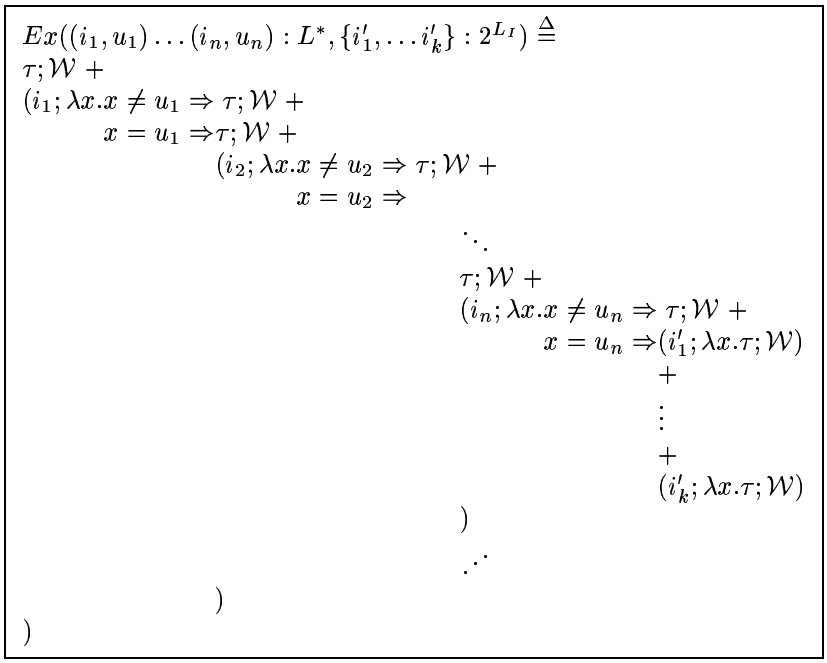

Figure 27: Definition of $Ex((i_1,u_1)\dots(i_n,u_n),\{i'_1,\dots i'_k\})$

$$\top \in \text{Result}(Ex(\gamma),\mathcal{S}) \wedge \top \notin \text{Result}(Ex(\gamma),\mathcal{S}')$$

thus concluding that $\mathcal{S} \sqsubseteq_{\sim MAY} \mathcal{S}'$ does not hold.
Now we show that if $\mathcal{S} <<_{MAY} \mathcal{S}'$ then $\mathcal{S} \sqsubseteq_{\sim MAY} \mathcal{S}'$. Suppose $\top \in \text{Result}(\mathcal{T},\mathcal{S})$ for some experimenter $\mathcal{T}$, so that there must be a computation starting from $\upsilon_{\text{in}} \parallel s_{\text{in}}$ leading to success and there must be a finite prefix of the computation, say

$$\upsilon_{\text{in}} \parallel s_{\text{in}} \rightsquigarrow \dots \upsilon_k \parallel s$$

which leads to success. Such a prefix gives rise to a derivation $s_{\text{in}} \xrightarrow{\gamma} s$, on the side of $\mathcal{S}$, for $\gamma = (i_1,u_1)\dots(i_n,u_n)$, and to a sequence of transitions $\upsilon_j \xrightarrow{\mu_j} \mathcal{O}_j$, for $j = 0\dots k-1$ such that either $\mu_j = i_i$ and $\upsilon_{j+1} = (\mathcal{O}_j\ u_i)$, for some i with $1 \leq i \leq n$, or $\mu_j = \tau$ and $\upsilon_{j+1} = \mathcal{O}_j$. Notice that the derivation on the side of the experimenter involves a sequence γ' which is equal to γ up to τ moves. We know that $\mathsf{lan}\ \mathcal{S} \subseteq \mathsf{lan}\ \mathcal{S}'$ since $\mathcal{S} <<_{MAY} \mathcal{S}'$; so also $\mathcal{S}'$ can perform $s'_{\text{in}} \xrightarrow{\gamma} s'$ for some s', and thus can be composed with the derivation we had for $\mathcal{T}$. And since this experimenter reported success somewhere in such a derivation, also this time it will do so. If the sequence we built is not maximal, we can extend it with further derivations starting from $\upsilon_k \parallel s'$. In any case we found a successful computation for $\mathcal{T} \parallel \mathcal{S}'$. This proves $\mathcal{S} \sqsubseteq_{\sim MAY} \mathcal{S}'$.

Part b)
The implication $\mathcal{S} \sqsubseteq_{\sim MUST} \mathcal{S}' \Rightarrow \mathcal{S} <<_{MUST} \mathcal{S}'$ follows from Lemma B.2 below. The converse can be proved as follows. Suppose $\mathcal{S} <<_{MUST} \mathcal{S}'$ and $\bot \notin \text{Result}(\mathcal{T},\mathcal{S})$. Let us consider an arbitrary computation starting from $\mathcal{T} \parallel \mathcal{S}'$:

$$\upsilon_{\text{in}} \parallel s'_{\text{in}} \rightsquigarrow \dots \upsilon_k \parallel s' \dots \tag{1}$$

There are two possibilities: either the above sequence is finite, or it is infinite. Let us first consider the case in which it is finite and stops at $\upsilon_k \parallel s'$. We must show that for some $i, 0 \leq i \leq k$, $\upsilon_i \in$ *Success*. This computation gives rise to two derivations: one on the side of $\mathcal{S}'$, $s'_{\text{in}} \xrightarrow{\gamma} s'$, for some $\gamma \in L^*$, and one on the side of the experimenter, starting with υ_{in}, ending with υ_k, and involving

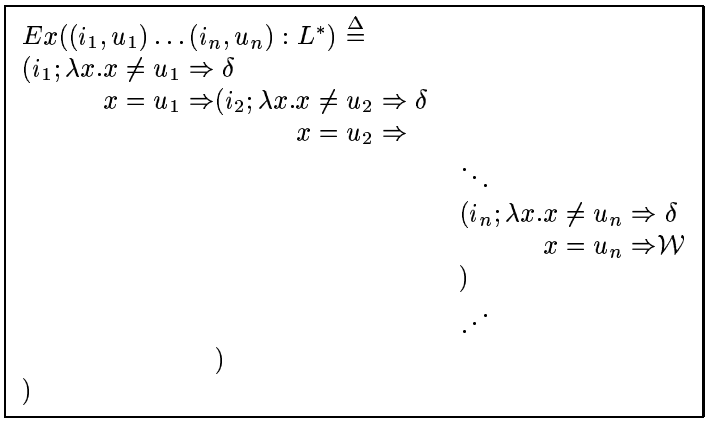

Figure 28: Definition of $Ex((i_1,u_1)\ldots(i_n,u_n))$

γ' which is equal to γ up to occurrences of τ. Now, $(\mathsf{S}\ s'\ \epsilon) \in (\mathsf{AS}\ \mathcal{S}'\ \gamma)$ and then there exists $T \in mfs(\mathbf{c}\ (\mathsf{AS}\ \mathcal{S}'\ \gamma))$ with $T \subseteq (\mathsf{S}\ s'\ \epsilon)$. Moreover, since $mfs(\mathbf{c}\ (\mathsf{AS}\ \mathcal{S}'\ \gamma)) \subset\subset mfs(\mathbf{c}\ (\mathsf{AS}\ \mathcal{S}\ \gamma))$ we can find $Z' \in mfs(\mathbf{c}\ (\mathsf{AS}\ \mathcal{S}\ \gamma))$ such that $Z' \subseteq T$. Thus we get $Z' \subseteq (\mathsf{S}\ s'\epsilon)$. Now there are three cases for Z':

i) $Z' \in \mathsf{AS}\ \mathcal{S}\ \gamma$. In this case there exists a s such that $s_{\mathrm{in}} \xrightarrow{\gamma} s$ and $Z' = (\mathsf{S}\ s\ \epsilon) \subseteq (\mathsf{S}\ s'\ \epsilon)$. So $v_k \parallel s$ cannot be extended and therefore the derivation $s_{\mathrm{in}} \xrightarrow{\gamma} s$ can be combined with the above mentioned derivation for $\mathcal{T}$ involving γ', in order to give a computation $v_{\mathrm{in}} \parallel s_{\mathrm{in}} \rightsquigarrow \ldots v_k \parallel s$ for $\mathcal{T} \parallel \mathcal{S}$. Since $\perp \notin \mathrm{Result}(\mathcal{T},\mathcal{S})$, there must be $i, 0 \leq i \leq k$, such that $v_i \in \mathit{Success}$.

ii) $Z' \in \mathbf{c}\ (\mathsf{AS}\ \mathcal{S}\ \gamma) \setminus \mathsf{AS}\ \mathcal{S}\ \gamma$. In this case there exists s such that $s_{\mathrm{in}} \xrightarrow{\gamma} s$ and $(\mathsf{S}\ s\ \epsilon) \in \mathsf{AS}\ \mathcal{S}\ \gamma$ with $(\mathsf{S}\ s\ \epsilon) \subseteq Z'$. Thus we have that $(\mathsf{S}\ s\ \epsilon) \subseteq (\mathsf{S}\ s'\ \epsilon)$. Then a similar reasoning as in case i) can be applied.

iii) $Z' \in (mfs\ (\mathbf{c}\ (\mathsf{AS}\ \mathcal{S}\ \gamma))) \setminus \mathbf{c}\ (\mathsf{AS}\ \mathcal{S}\ \gamma)$. In this case, there exists a set K in $\mathbf{c}\ (\mathsf{AS}\ \mathcal{S}\ \gamma)$ such that $Z' \in mfs\{K\}$. Moreover there exists a s such that $s_{\mathrm{in}} \xrightarrow{\gamma} s$ and $\mathsf{S}\ s\ \epsilon \in \mathsf{AS}\ \mathcal{S}\ \gamma$, such that $\mathsf{S}\ s\ \epsilon \subseteq K$. We know that $Z' \subseteq \mathsf{S}\ s'\ \epsilon$. We know also that $v_k \parallel s'$ cannot be extended, and that $IN\ (\mathsf{S}\ s\ \epsilon) \subseteq INZ'$ because of the definition of mfs. All this together brings to the fact that $v_k \parallel s$ cannot be extended. Then a similar reasoning as in case i) can be applied.

Let us now consider the case in which the computation is infinite. Also in this case the computation gives rise to two derivations, which may be infinite: one on the side of $\mathcal{S}'$, $s'_{\mathrm{in}} \xrightarrow{\gamma}$, for some γ, and one on the side of the experimenter, starting with v_{in}, involving γ' which is equal to γ up to occurrences of τ. Both γ and γ' may be infinite sequences. Suppose now that for every natural number n there exists $m > n$ such that the m-th element of γ' is different from τ. This means that for each finite prefix $\bar{\gamma}$ of γ there exists s' such that $(\mathsf{S}\ s'\ \epsilon) \in \mathsf{AS}\ \mathcal{S}'\ \bar{\gamma}$ and so there exists $T \in mfs(\mathbf{c}\ (\mathsf{AS}\ \mathcal{S}'\ \bar{\gamma}))$ with $T \subseteq (\mathsf{S}\ \mathcal{S}'\ \epsilon)$. But then, since $mfs(\mathbf{c}\ (\mathsf{AS}\ \mathcal{S}'\ \bar{\gamma})) \subset\subset mfs(\mathbf{c}\ (\mathsf{AS}\ \mathcal{S}\ \bar{\gamma}))$, we can find $Y \in mfs(\mathbf{c}\ (\mathsf{AS}\ \mathcal{S}\ \bar{\gamma}))$ with $Y \subseteq T$.This means that $\mathsf{AS}\ \mathcal{S}\ \bar{\gamma}$ is non-empty and then there exists s such that $s_{\mathrm{in}} \xrightarrow{\bar{\gamma}} s$. But then, since the above holds for every finite prefix of γ, we get also $s_{\mathrm{in}} \xrightarrow{\gamma}$ and we can build an infinite computation by composing the derivation with the above derivation on the side of the experimenter involving γ'. Since $\perp \notin \mathrm{Result}(\mathcal{T},\mathcal{S})$, there will be a $v_i \in \mathit{Success}$ for some i. So the computation (1) above is successful. A similar reasoning applies also to the case in which there exists n such that the m-th element of γ' is τ for all $m \geq n$. Both in the case the successful state v_i occurs in the silent suffix of γ' and in the case in which it occurs before such a suffix we can build a successful computation proceeding as above. Thus we conclude that in all cases $\perp \notin \mathrm{Result}(\mathcal{T},\mathcal{S}')$, and so $\mathcal{S} \sqsubseteq_{\approx_{MUST}} \mathcal{S}'$.

Part c)

Obviously follows from parts a) and b). □

The above proof used Lemma B.2 below, which in turn uses the the following lemma, which shows the relationship between $\sqsubseteq^{EX}_{\sim MUST}$ and the languages of the relevant LTSs. Lemma B.2 shows that EX is sufficiently expressive for the $MUST$ preorders.

Lemma B.1 *For all finite LTSs $\mathcal{S}$, $\mathcal{S}'$ over $L = L_I \times L_U$ the following holds: $\mathcal{S} \sqsubseteq^{EX}_{\sim MUST} \mathcal{S}'$ implies* $\mathsf{lan}\ \mathcal{S}' \subseteq \mathsf{lan}\ \mathcal{S}$ ◇

Proof
If $\gamma = \epsilon$, then trivially $\gamma \in \mathsf{lan}\ \mathcal{S}$. Suppose $\gamma = \gamma' x \in \mathsf{lan}\ \mathcal{S}' \setminus \mathsf{lan}\ \mathcal{S}$; then we can produce the following derivation

$\gamma' x \notin \mathsf{lan}\ \mathcal{S}$

$\Rightarrow$ {Def. of experimenter $Ex(\gamma', x)$}

$\perp \notin \text{Result}(Ex(\gamma', x), \mathcal{S})$

$\Rightarrow$ $\{\mathcal{S} \sqsubseteq^{EX}_{\sim MUST} \mathcal{S}'\}$

$\perp \notin \text{Result}(Ex(\gamma', x), \mathcal{S}')$

$\Rightarrow$ {Def. of experimenter $Ex(\gamma', x)$}

$\gamma' x \notin \mathsf{lan}\ \mathcal{S}'$

which is a contradiction, since we assumed $\gamma \in \mathsf{lan}\ \mathcal{S}'$. □

Lemma B.2
For all finite LTSs $\mathcal{S}$, $\mathcal{S}'$ over $L = L_I \times L_U$ if $\mathcal{S} \sqsubseteq^{EX}_{\sim MUST} \mathcal{S}'$ then $\mathcal{S} <<_{MUST} \mathcal{S}'$ ◇

Proof
Let us assume $\mathcal{S} = (S, s_{\text{in}}, L, \rightarrow)$ and $\mathcal{S} = (S', s'_{\text{in}}, L, \rightarrow)$ for $L = L_I \times L_U$. From the definition of $<<_{MUST}$ we know we must show that $mfs(\mathbf{c}\ (\mathsf{AS}\ \mathcal{S}'\ \gamma)) \subset\subset mfs(\mathbf{c}\ (\mathsf{AS}\ \mathcal{S}\ \gamma))$. Note that we only need to consider sequences γ which are in $\mathsf{lan}\ \mathcal{S}'$, because if $\gamma \notin \mathsf{lan}\ \mathcal{S}'$ then $\mathsf{AS}\ \mathcal{S}'\ \gamma = \emptyset$, and thus $mfs(\mathbf{c}\ (\mathsf{AS}\ \mathcal{S}'\ \gamma)) = \emptyset$, and then the relation $\subset\subset$ trivially holds. So, consider $\gamma \in \mathsf{lan}\,\mathcal{S}'$. From Lemma B.1 we know that $\gamma \in \mathsf{lan}\,\mathcal{S}$ and thus $\mathsf{AS}\ \mathcal{S}\ \gamma \neq \emptyset$ and $mfs(\mathbf{c}\ (\mathsf{AS}\ \mathcal{S}\ \gamma)) \neq \emptyset$. Now we can continue the proof by deriving a contradiction if we assume that $mfs(\mathbf{c}\ (\mathsf{AS}\ \mathcal{S}'\ \gamma)) \subset\subset mfs(\mathbf{c}\ (\mathsf{AS}\ \mathcal{S}\ \gamma))$does *not* hold. Under this assumption, by the definition of $\subset\subset$ and considering that both $mfs(\mathbf{c}\ (\mathsf{AS}\ \mathcal{S}\ \gamma))$ and $mfs(\mathbf{c}\ (\mathsf{AS}\ \mathcal{S}'\ \gamma))$ are non-empty, we can assume that there exists a set $R \in mfs(\mathbf{c}\ (\mathsf{AS}\ \mathcal{S}'\ \gamma))$ such that $Z \not\subseteq R$ for all $Z \in mfs(\mathbf{c}\ (\mathsf{AS}\ \mathcal{S}\ \gamma))$.
Now we first show that in each set $Z \in mfs(\mathbf{c}\ (\mathsf{AS}\ \mathcal{S}\ \gamma))$ we can choose an element (i, u) in such a way that it is not only different from all elements in the set R, but also such that the input part i is different from all input parts of elements in R. We show this by contradiction: for any $Z \in mfs(\mathbf{c}\ (\mathsf{AS}\ \mathcal{S}\ \gamma))$we assume that it is impossible to choose such an element and we reach a contradiction. For ease of notation, let $mfs(\mathbf{c}\ (\mathsf{AS}\ \mathcal{S}\ s))$ be the set $\{Z_1, \ldots, Z_k\}$.
So, now suppose that Z_i differs from R only by elements that differ only in their output part, that is $IN\ Z_i \subseteq IN\ R \wedge \exists (i, u) \in Z_i.\ (i, u) \notin R$.
Because of the definitions of mfs and $\mathbf{c}$, for each Z_i one of the following cases applies:

i) $Z_i \in \mathsf{AS}\ \mathcal{S}\ \gamma$

ii) $Z_i \in \mathbf{c}\ (\mathsf{AS}\ \mathcal{S}\ \gamma) \setminus \mathsf{AS}\ \mathcal{S}\ \gamma$

iii) $Z_i \in mfs\ (\mathbf{c}\ (\mathsf{AS}\ \mathcal{S}\ \gamma)) \setminus \mathbf{c}\ (\mathsf{AS}\ \mathcal{S}\ \gamma)$

Case i) Suppose $Z_i \in \mathsf{AS}\ \mathcal{S}\ \gamma$.
We first show that $\bigcup_{K' \in \mathsf{AS}\ \mathcal{S}'\ \gamma} K' \subseteq \bigcup_{K \in \mathsf{AS}\ \mathcal{S}\ \gamma} K$. For $\mathsf{AS}\ \mathcal{S}'\ \gamma = \emptyset$ or if $\mathsf{AS}\ \mathcal{S}'\ \gamma = \{\emptyset\}$ this is trivial. For $\mathsf{AS}\ \mathcal{S}'\ \gamma$ containing a non-empty set we can derive:

$x \in \bigcup_{K' \in \mathsf{AS}\ \mathcal{S}'\ \gamma} K'$

$\Rightarrow$ {Set theory}

$\exists K' \in \mathsf{AS}\ \mathcal{S}'\ \gamma.\ x \in K'$

$\Rightarrow$ {Def. of lan and of AS}

$\gamma x \in \mathsf{lan}\ \mathcal{S}'$

$\Rightarrow$ {$\mathcal{S} \sqsubseteq^{EX}_{\sim MUST} \mathcal{S}'$, Lemma B.1}

$\gamma x \in \mathsf{lan}\ \mathcal{S}$

$\Rightarrow$ {Def. of AS}

$\exists K \in \mathsf{AS}\ \mathcal{S}\ \gamma.\ x \in K$

$\Rightarrow$ {Set theory}

$x \in \bigcup_{K \in \mathsf{AS}\ \mathcal{S}\ \gamma} K$

Now we can show that we can find in $\mathbf{c}\ (\mathsf{AS}\ \mathcal{S}\ \gamma)$ a set T that extends Z_i with exactly those elements in R which have the same input part as those in Z_i. We can do this because T is an intermediate set between Z_i—which is in $\mathsf{AS}\ \mathcal{S}\ \gamma$—and $\bigcup_{K \in \mathsf{AS}\ \mathcal{S}\ \gamma} K$—which is an element of $\mathbf{c}\ (\mathsf{AS}\ \mathcal{S}\ \gamma)$—and which contains all elements that are in R since we showed $\bigcup_{K' \in \mathsf{AS}\ \mathcal{S}'\ \gamma} K' \subseteq \bigcup_{K \in \mathsf{AS}\ \mathcal{S}\ \gamma} K$. Formally, $T = Z_i \cup \{(i,u) \mid i \in IN Z_i\ \wedge\ (i,u) \in R\}$.
It is now easy to see that the set $Z = \{(i,u) \mid i \in IN Z_i\ \wedge\ (i,u) \in R\}$ is an element of $mfs(\mathbf{c}\ (\mathsf{AS}\ \mathcal{S}\ \gamma))$. In fact $Z \subseteq T \in \mathbf{c}\ (\mathsf{AS}\ \mathcal{S}\ \gamma)$; moreover Z is functional, since $Z \subseteq R$, and it is maximal. This last fact can be proven by contradiction: suppose there exists an element $(i,u) \in T \setminus Z$; then by definition of Z, since $IN\ T = IN\ Z$, there exists u' such that $(i,u') \in Z$ and since Z is functional we get $u = u'$. So, in the end we found $Z \in mfs(\mathbf{c}\ (\mathsf{AS}\ \mathcal{S}\ \gamma))$ with $Z \subseteq R$. This contradicts our original assumption that there exists $R \in mfs(\mathbf{c}\ (\mathsf{AS}\ \mathcal{S}'\ \gamma))$ such that $Z \not\subseteq R$ for all $Z \in mfs(\mathbf{c}\ (\mathsf{AS}\ \mathcal{S}\ \gamma))$.
Case ii) Suppose $Z_i \in \mathbf{c}\ (\mathsf{AS}\ \mathcal{S}\ \gamma) \setminus \mathsf{AS}\ \mathcal{S}\ \gamma$ and such that $IN\ Z_i \subseteq IN\ R$. In this case the reasoning is the same as in Case i).
Case iii) Suppose $Z_i \in mfs\ (\mathbf{c}\ (\mathsf{AS}\ \mathcal{S}\ \gamma)) \setminus \mathbf{c}\ (\mathsf{AS}\ \mathcal{S}\ \gamma)$. In that case, by definition of mfs, we know that there exists a set $K \in \mathbf{c}\ (\mathsf{AS}\ \mathcal{S}\ \gamma)$ such that $IN\ Z_i = IN\ K$ and then $IN\ K \subseteq IN\ R$. For K we can setup a reasoning like in case ii) leading to the fact that there will exist a set in $mfs\ (\mathbf{c}\ (\mathsf{AS}\ \mathcal{S}\ \gamma))$ which is a subset of R which is in contradiction with the assumptions, and thus such an Z_i cannot exist.
This ends the proof for each of the cases for Z_i and shows that in all sets $Z \in mfs\ (\mathbf{c}\ (\mathsf{AS}\ \mathcal{S}\ \gamma))$ we can choose an element, with an input part different from all those of elements of R. And because we can find such an element in every such Z we can also find it in each set $A \in \mathsf{AS}\ \mathcal{S}\ \gamma$. In fact, let A' be a set in $\mathbf{c}\ (\mathsf{AS}\ \mathcal{S}\ \gamma)$. If it is functional then $A' = Z$ for some Z as above. If it is not functional then it includes some functional set Z as above. In particular this holds for all $A \in \mathsf{AS}\ \mathcal{S}\ \gamma$.
Now let's choose in each $Z_i \in mfs\ (\mathbf{c}\ (\mathsf{AS}\ \mathcal{S}\ \gamma))$ one element x_i such that the input part of x_i is not in INR. Let's call X the set of input parts of all this elements x_i, that is $X = \{i \mid (i,u) \in \{x_1, \ldots, x_k\}\} = \{i_1, \ldots, i_k\}$. Clearly, $\bot \notin \mathrm{Result}(Ex(\gamma, X), \mathcal{S})$ because for each $A \in \mathsf{AS}\ \mathcal{S}\ \gamma$ there exists an element (i,u) with $i \in X$ as shown before. On the other hand, as we will show below, $\bot \in \mathrm{Result}(Ex(\gamma, X), \mathcal{S}')$ which contradicts the hypothesis $\mathcal{S} \sqsubseteq^{EX}_{\sim MUST} \mathcal{S}'$.

$$\boxed{Ex(\{i_1,\ldots,i_k\}) \triangleq (i_1;\lambda x.\tau;\mathcal{W}) + \ldots + (i_k;\lambda x.\tau;\mathcal{W})}$$

Figure 29: Definition of $Ex(X)$

What we have to show is that there exists a state s' of $\mathcal{S}'$ which can be reached by s'_{in} by performing γ such that the following computation, where $Ex(X)$ is shown in Fig.29, is unsuccessful:

$$Ex(\gamma, X) \mid\mid s'_{\text{in}} \rightsquigarrow \ldots \rightsquigarrow Ex(X) \mid\mid s'$$

Recall that $\gamma \in \mathsf{lan}\,\mathcal{S}'$. So, the fact that the above computation is unsuccessful means that there exists s' such that $IN(\mathsf{S}\ s'\ \epsilon) \cap X = \emptyset$.
There are three possibilities:
i) $R \in \mathsf{AS}\ \mathcal{S}'\ \gamma$ which means that there exists an s' such that $(\mathsf{S}\ s'\ \epsilon) = R \in \mathsf{AS}\ \mathcal{S}'\ \gamma$. And we know that $(IN\ R) \cap X = \emptyset$.
ii) $R \in \mathbf{c}\ (\mathsf{AS}\ \mathcal{S}'\ \gamma) \setminus \mathsf{AS}\ \mathcal{S}'\ \gamma$. By definition of closure, this implies that there exists an s' such that $(\mathsf{S}\ s'\ \epsilon) \in \mathsf{AS}\ \mathcal{S}'\ \gamma$ and $(\mathsf{S}\ s'\ \epsilon) \subseteq R$. So, $IN(\mathsf{S}\ s'\ \epsilon) \subseteq IN\ R$, but again we know that $(IN\ R) \cap X = \emptyset$.
iii) $R \in mfs(\mathbf{c}\ (\mathsf{AS}\ \mathcal{S}'\ \gamma)) \setminus (\mathbf{c}\ (\mathsf{AS}\ \mathcal{S}'\ \gamma) \cup \mathsf{AS}\ \mathcal{S}'\ \gamma)$. This means that there exists a set $K \in \mathbf{c}\ (\mathsf{AS}\ \mathcal{S}'\ \gamma)$ such that $R \in mfs\{K\}$. By definition of closure, this implies that there exists an s' such that $(\mathsf{S}\ s'\ \epsilon) \in \mathsf{AS}\ \mathcal{S}'\ \gamma$ and $(\mathsf{S}\ s'\epsilon) \subseteq K$. So, $IN(\mathsf{S}\ s'\ \epsilon) \subseteq IN\ K$. Since R is a maximal functional subset of K we know that $IN\ K = IN\ R$, so again $(\mathsf{S}\ s'\ \epsilon) \subseteq R$ with $IN\ R \cap X = \emptyset$.
This proves the lemma. □

The following is an obvious corollary of Lemma B.1

Corollary B.1 *For all finite LTSs $\mathcal{S}$, $\mathcal{S}'$ over $L = L_I \times L_U$ the following holds: $\mathcal{S} \sqsubseteq_{MUST} \mathcal{S}'$ implies $\mathsf{lan}\,\mathcal{S}' \subseteq \mathsf{lan}\,\mathcal{S}$* ◇

Proof of Theorem 4.2

$\mathsf{T_{FAA}}\ \mathcal{S}\ \leq_{FAA}\ \mathsf{T_{FAA}}\mathcal{S}'$

$\Leftrightarrow$ {Def. of $\leq_{FAA}$ }

$\mathsf{lan}\ (\mathsf{T_{FAA}}\ \mathcal{S}) = \mathsf{lan}\ (\mathsf{T_{FAA}}\ \mathcal{S}') \wedge \forall \gamma \in \mathsf{lan}(\mathsf{T_{FAA}}\ \mathcal{S}).\ \mathsf{AS_{FAA}}\ (\mathsf{T_{FAA}}\ \mathcal{S}')\ \gamma \subseteq \mathsf{AS_{FAA}}\ (\mathsf{T_{FAA}}\ \mathcal{S})\ \gamma$

$\Leftrightarrow$ {Def. of $\mathsf{T_{FAA}}$}

$\mathsf{lan}\ \mathcal{S} = \mathsf{lan}\ \mathcal{S}' \wedge \forall \gamma \in \mathsf{lan}\ \mathcal{S}.\ mfs(\mathbf{c}\ (\mathsf{AS}\ \mathcal{S}'\ \gamma)) \subseteq mfs(\mathbf{c}\ (\mathsf{AS}\ \mathcal{S}\ \gamma))$

$\Leftrightarrow$ {Lemma B.3 below ; Notice that $\mathsf{lan}\ \mathcal{S} = \mathsf{lan}\ \mathcal{S}'$ implies $\bigcup_{x \in (\mathsf{AS}\ \mathcal{S}'\ \gamma)} x = \bigcup_{x \in (\mathsf{AS}\ \mathcal{S}\ \gamma)} x$ }

$\mathsf{lan}\ \mathcal{S} = \mathsf{lan}\ \mathcal{S}' \wedge \forall \gamma \in \mathsf{lan}\ \mathcal{S}.\ mfs(\mathbf{c}\ (\mathsf{AS}\ \mathcal{S}'\ \gamma)) \subset\subset mfs(\mathbf{c}\ (\mathsf{AS}\ \mathcal{S}\ \gamma))$

$\Leftrightarrow$ {$\gamma \notin \mathsf{lan}\ \mathcal{S} = \mathsf{lan}\ \mathcal{S}'$ iff $\mathsf{AS}\ \mathcal{S}\ \gamma = \mathsf{AS}\ \mathcal{S}'\ \gamma = \emptyset$ from the def. of AS}

$\mathsf{lan}\ \mathcal{S} = \mathsf{lan}\ \mathcal{S}' \wedge \forall \gamma \in L^*.\ mfs(\mathbf{c}\ (\mathsf{AS}\ \mathcal{S}'\ \gamma)) \subset\subset mfs(\mathbf{c}\ (\mathsf{AS}\ \mathcal{S}\ \gamma))$

$\Leftrightarrow$ {Def. of $<<_{MUST}$; Theo .4.1 (b)and Corollary B.1}

$\mathsf{lan}\ \mathcal{S} \subseteq \mathsf{lan}\ \mathcal{S}' \wedge \forall \gamma \in L^*.\ mfs(\mathbf{c}\ (\mathsf{AS}\ \mathcal{S}'\ \gamma)) \subset\subset mfs(\mathbf{c}\ (\mathsf{AS}\ \mathcal{S}\ \gamma))$

$\Leftrightarrow$ {Def. of $<<_{MAY}$, $<<_{MUST}$, $<<$ }

$\mathcal{S}\ <<\ \mathcal{S}'$ □

In the proof above the following lemma have been used.

Lemma B.3
For X, Y finite sets of finite subsets of L such that $\bigcup_{x \in X} x = \bigcup_{y \in Y} y$ the following holds: $mfs(\mathbf{c}\ X) \subset\subset mfs(\mathbf{c}\ Y)$ iff $mfs(\mathbf{c}\ X) \subseteq mfs(\mathbf{c}\ Y)$ $\diamond$

Proof

The proof consists of two parts, one for each implication.
i) $\Leftarrow$-part: trivial.
ii) $\Rightarrow$-part uses Lemma B.4 below

$x \in mfs(\mathbf{c}\ X)$

$\Rightarrow$ {Def. of $\subset\subset$}

$\exists y \in mfs(\mathbf{c}\ Y).\ y \subseteq x$

$\Rightarrow$ {Def. of mfs}

$\exists z \in \mathbf{c}\ Y.\ y \in mfs\{z\} \wedge y \subseteq x$

$\Rightarrow$ {Lemma B.4 below}

$(x \cup z) \in \mathbf{c}\ Y \wedge x \in mfs\{x \cup z\}$

$\Rightarrow$ {Def. of mfs}

$x \in mfs(\mathbf{c}\ Y)$ $\square$

Lemma B.4 *For X, Y finite sets of finite sets over L such that $\bigcup_{x \in X} x = \bigcup_{y \in Y} y$ the following holds: $z \in \mathbf{c}\ Y \wedge y \in mfs\{z\} \wedge y \subseteq x \Rightarrow x \cup z \in \mathbf{c}\ Y \wedge x \in mfs\{x \cup z\}$* $\diamond$

Proof

First we prove that $x \cup z \in \mathbf{c}\ Y$. We know that $z \in \mathbf{c}\ Y$ and obviously $z \subseteq x \cup z$. Moreover, from the definition of closure we know that $(\bigcup_{y \in Y} y) \in \mathbf{c}\ Y$. So:

$x \subseteq \bigcup_{v \in X} v$

$\Rightarrow$ {By hypothesis $\bigcup_{v \in X} v = \bigcup_{w \in Y} w$}

$x \subseteq \bigcup_{w \in Y} w$

$\Rightarrow$ {$z \in \mathbf{c}\ Y$}

$x \subseteq \bigcup_{w \in Y} w \wedge z \in \mathbf{c}\ Y$

$\Rightarrow$ {$z \in \mathbf{c}\ Y \Rightarrow z \subseteq \bigcup_{w \in Y} w$ by def. of closure}

$x \cup z \subseteq \bigcup_{w \in Y} w$

$\Rightarrow$ {$z \subseteq x \cup z$}

$z \subseteq (x \cup z) \wedge (x \cup z) \subseteq \bigcup_{w \in Y} w$

$\Rightarrow$ {Def. of $\mathbf{c}$ }

$x \cup z \in \mathbf{c}\ Y$

Now we prove that $x \in mfs\{x \cup z\}$. The proof is by derivation of a contradiction. For finite set w over L, we let $func\ w$ denote the predicate $\forall (i_1, u_1), (i_2, u_2) \in w.\ i_1 = i_2 \Rightarrow u_1 = u_2$. Suppose

$x \notin mfs\{x \cup z\}$

$x \notin mfs\{x \cup z\}$

$\Rightarrow$ {Def. of mfs; x is functional}

$\exists k \subseteq x \cup z.\ x \subset k \wedge func\ k$

$\Rightarrow$ {Set theory}

$\exists a \in z \setminus x.\ a \in k$

$\Rightarrow$ {$y \in mfs\{z\} \wedge x \cap z = y$ see note below; Set theory}

$y \subset k \cap z$

$\Rightarrow$ {$k \cap z$ is functional since k is functional; $k \cap z \subseteq z$}

$y \notin mfs\{z\}$

The fact that we derive that y is not in $mfs\{z\}$ is in contradiction with the assumptions. So we proved $x \in mfs\{x \cup z\}$. Note that in the one but last step in the proof above we used $y = x \cap z$. The reason is that $y \subseteq x$ and $y \in mfs\{z\}$ by hypothesis and this last fact implies $y \subseteq z$. Thus $y \subseteq x \cap z$. Moreover, x is functional so also $x \cap z$ must be functional and of course $x \cap z \subseteq z$. But $y \in mfs\{z\}$ so it cannot be $y \subset (x \cap z)$. □

B.2 Proofs related to Sect. 5

Proof of Lemma 5.1

Part i -
Follows directly from Lemma A.1 (i)
Part ii -

$\mathcal{C} \xrightarrow{e/\Sigma}_{\Sigma} \mathcal{C}'$

$\Rightarrow$ {Second rule of Def. 5.3}

$H \uparrow \emptyset :: (\mathcal{C}, \{e\}) \xrightarrow{\emptyset} (\mathcal{C}', \mathcal{E})$

$\Rightarrow$ {Lemma A.1 (ii)}

$\mathcal{C} = \mathcal{C}'$

Part iii -
By contradiction. Suppose there exist $\mathcal{C}' \in \mathrm{Conf}_H$ and $\mathcal{E} \in \Theta_E$ such that $\mathcal{C} \xrightarrow{e/\mathcal{E}}_{\Sigma} \mathcal{C}'$ while also $\mathcal{C} \xrightarrow{e/\Sigma}_{\Sigma} \mathcal{C}$. By the first rule of Def. 5.3, $\mathcal{C} \xrightarrow{e/\mathcal{E}}_{\Sigma} \mathcal{C}'$ would imply $H \uparrow \emptyset :: (\mathcal{C}, \{e\}) \xrightarrow{\mathcal{L}} (\mathcal{C}', \mathcal{E})$ for some $\mathcal{L} \neq \emptyset$. But from $\mathcal{C} \xrightarrow{e/\Sigma}_{\Sigma} \mathcal{C}$, by the second rule of Def. 5.3, we would also have $H \uparrow \emptyset :: (\mathcal{C}, \{e\}) \xrightarrow{\emptyset} (\mathcal{C}'', \mathcal{E}')$, which, using Lemma A.1 (ii) and (iii), would lead to $\nexists \mathcal{L} \neq \emptyset.\ H \uparrow \emptyset :: (\mathcal{C}, \{e\}) \xrightarrow{\mathcal{L}}$, which is a contradiction. □

Proof of Proposition 5.1

We first consider the direct implication.

$(\mathcal{C}, \mathcal{E}) \xrightarrow{\mathcal{L}} (\mathcal{C}', (\mathsf{Join}\ \mathcal{E}''\ \mathcal{E}'))$

$\Rightarrow$ {Def. of $\xrightarrow{\mathcal{L}}$ (see Def. 3.3)}

$\exists e \in E.\ H \uparrow \emptyset :: (\mathcal{C}, \{e\}) \xrightarrow{\mathcal{L}} (\mathcal{C}', \mathcal{E}') \wedge (\mathsf{Sel}\ \mathcal{E}\ e\ \mathcal{E}'')$

There are two cases: $\mathcal{L} \neq \emptyset$ and $\mathcal{L} = \emptyset$.

Case 1 - $\mathcal{L} \neq \emptyset$

$\Rightarrow$ {First rule of Def. 5.3, Def. of ΣQueue($\mathcal{F}$)}

$\exists e \in E.\ \mathcal{C} \xrightarrow{e/\mathcal{E}'}_{\Sigma} \mathcal{C}' \wedge$

$\Sigma\text{Queue}(\mathcal{E}) \xrightarrow{e} \lambda X.X \neq \Sigma \Rightarrow \Sigma\text{Queue}(\mathsf{Join}\ \mathcal{E}''\ X) + X = \Sigma \Rightarrow \Sigma\text{Queue}(\mathcal{E}'')$

$\Rightarrow$ {Def. 3.5,Def. of ΣQueue($\mathcal{F}$), $\mathcal{E}' \in \Theta_E$}

$(\Sigma\text{Queue}(\mathcal{E}) \mid\mid \mathcal{C}) \rightsquigarrow (\Sigma\text{Queue}(\mathsf{Join}\ \mathcal{E}''\ \mathcal{E}') \mid\mid \mathcal{C}')$

Case 2 - $\mathcal{L} = \emptyset$

$\Rightarrow$ {Lemma A.1 ii)}

$\exists e \in E.\ H \uparrow \emptyset :: (\mathcal{C}, \{e\}) \xrightarrow{\emptyset} (\mathcal{C}', \mathcal{E}') \wedge (\mathsf{Sel}\ \mathcal{E}\ e\ \mathcal{E}'') \wedge \mathcal{E}' = \langle\rangle$

$\Rightarrow$ {Second rule of Def. 5.3, Def. of ΣQueue($\mathcal{F}$)}

$\exists e \in E.\ \mathcal{C} \xrightarrow{e/\Sigma}_{\Sigma} \mathcal{C}' \wedge \mathcal{E}' = \langle\rangle \wedge$

$\Sigma\text{Queue}(\mathcal{E}) \xrightarrow{e} \lambda X.X \neq \Sigma \Rightarrow \Sigma\text{Queue}(\mathsf{Join}\ \mathcal{E}''\ X) + X = \Sigma \Rightarrow \Sigma\text{Queue}(\mathcal{E}'')$

$\Rightarrow$ {Def. 3.5,Def. of ΣQueue($\mathcal{F}$)}

$(\Sigma\text{Queue}(\mathcal{E}) \mid\mid \mathcal{C}) \rightsquigarrow (\Sigma\text{Queue}(\mathcal{E}'') \mid\mid \mathcal{C}') \wedge \mathcal{E}' = \langle\rangle$

$\Rightarrow$ {$\mathcal{E}' = \langle\rangle \Rightarrow (\mathsf{Join}\ \mathcal{E}''\ \mathcal{E}') = \mathcal{E}''$}

$(\Sigma\text{Queue}(\mathcal{E}) \mid\mid \mathcal{C}) \rightsquigarrow (\Sigma\text{Queue}(\mathsf{Join}\ \mathcal{E}''\ \mathcal{E}') \mid\mid \mathcal{C}')$

Now we consider the reverse implication.

$(\Sigma\text{Queue}(\mathcal{E}) \mid\mid \mathcal{C}) \rightsquigarrow (\Sigma\text{Queue}(\mathsf{Join}\ \mathcal{E}''\ \mathcal{E}') \mid\mid \mathcal{C}')$

$\Rightarrow$ {Def. 3.5,Def. of ΣQueue($\mathcal{F}$)}

$\exists e \in E, u \in {}^{\Sigma}\Theta_E.\ \mathcal{C} \xrightarrow{e/u}_{\Sigma} \mathcal{C}' \wedge (\mathsf{Sel}\ \mathcal{E}\ e\ \mathcal{E}'') \wedge$

$\Sigma\text{Queue}(\mathcal{E}) \xrightarrow{e} \lambda X.X \neq \Sigma \Rightarrow \Sigma\text{Queue}(\mathsf{Join}\ \mathcal{E}''\ X) + X = \Sigma \Rightarrow \Sigma\text{Queue}(\mathcal{E}'')$

There are two cases: $X \neq \Sigma$ and $X = \Sigma$.

Case 1 - $X \neq \Sigma$

$\Rightarrow$ {First rule of Def. 5.3}

$\exists e \in E, \mathcal{L} \neq \emptyset, \mathcal{E}' \in \Theta_E.\ H \uparrow \emptyset :: (\mathcal{C}, \{e\}) \xrightarrow{\mathcal{L}} (\mathcal{C}', \mathcal{E}') \wedge (\mathsf{Sel}\ \mathcal{E}\ e\ \mathcal{E}'')$

$\Rightarrow$ {Def. of $\xrightarrow{\mathcal{L}}$ (see Def. 3.3)}

$\exists \mathcal{L}.\ (\mathcal{C}, \mathcal{E}) \xrightarrow{\mathcal{L}} (\mathcal{C}', (\mathsf{Join}\ \mathcal{E}''\ \mathcal{E}'))$

Case 2 - $X = \Sigma$

$\Rightarrow$ {Second rule of Def. 5.3}

$\exists e \in E, \mathcal{E}' \in \Theta_E.\ H \uparrow \emptyset :: (\mathcal{C}, \{e\}) \xrightarrow{\emptyset} (\mathcal{C}', \mathcal{E}') \wedge (\mathsf{Sel}\ \mathcal{E}\ e\ \mathcal{E}'')$

$\Rightarrow$ {Def. of $\xrightarrow{\mathcal{L}}$ (see Def. 3.3)}

$\exists \mathcal{L}.\ (\mathcal{C}, \mathcal{E}) \xrightarrow{\mathcal{L}} (\mathcal{C}', (\mathsf{Join}\ \mathcal{E}''\ \mathcal{E}'))$ □

Proof of Lemma 5.2

$\gamma \in (\mathsf{lan}\ \mathcal{S}) \ \wedge\ i \in L_I \ \wedge\ u \in \mathsf{OUT}\ \mathcal{S}'\ \gamma\ i$

$\Rightarrow$ {Lemma B.5}

$\gamma(i,u) \in (\mathsf{lan}\ \mathcal{S}')$

$\Rightarrow$ {$(\mathsf{lan}\ \mathcal{S}') \subseteq (\mathsf{lan}\ \mathcal{S})$ by hypothesis}

$\gamma(i,u) \in (\mathsf{lan}\ \mathcal{S})$

$\Rightarrow$ {Lemma B.5}

$u \in \mathsf{OUT}\ \mathcal{S}\ \gamma\ i$ □

Proof of Lemma 5.3

By contradiction; suppose that there exists $\gamma \in (\mathsf{lan}\ \ \mathcal{S}') \setminus (\mathsf{lan}\ \ \mathcal{S})$. Let $\bar{\gamma}$ the longest prefix of γ such that $\bar{\gamma} \in (\mathsf{lan}\ \ \mathcal{S})$; such a prefix exists since at least $\epsilon \in (\mathsf{lan}\ \ \mathcal{S})$ by definition. Let us assume $\gamma = \bar{\gamma}(i,u)\gamma'$ for some $i \in L'_I, u \in L'_U, \gamma' \in (L'_I \times L'_U)^*$. We can now derive the following:

$\bar{\gamma}(i,u) \in (\mathsf{lan}\ \ \mathcal{S}')$

$\Rightarrow$ {Lemma B.5}

$u \in \mathsf{OUT}\ \mathcal{S}'\ \bar{\gamma}\ i$

$\Rightarrow$ {$\bar{\gamma} \in (\mathsf{lan}\ \ \mathcal{S})$ by assumption; $i \in L'_I$ by assumption; $L'_I \subseteq L_I$ and $\mathcal{S}'\ \sqsubseteq_{\mathsf{co}}\ \mathcal{S}$ by hypothesis}

$u \in \mathsf{OUT}\ \mathcal{S}\ \bar{\gamma}\ i$

$\Rightarrow$ {Lemma B.5}

$\bar{\gamma}(i,u) \in (\mathsf{lan}\ \ \mathcal{S})$

which is a contradiction since $\bar{\gamma}(i,u)$ is a prefix of γ. □

The above proofs used the following lemma:

Lemma B.5
For $\mathcal{S}$ finite LTS over $L_I \times L_U, i \in L_I, u \in L_U$ and $\gamma \in (L_I \times L_U)^*$ the following holds:
$u \in (\mathsf{OUT}\ \mathcal{S}\ \gamma\ i)$ iff $\gamma(i,u) \in (\mathsf{lan}\ \mathcal{S})$ ◇

Proof

$u \in (\mathsf{OUT}\ s\ \gamma\ i)$

$\Leftrightarrow$ {Def. of OUT}

$\exists s'.\ (s \xrightarrow{\gamma} s')\ \wedge s' \xrightarrow{(i,u)}$

$\Leftrightarrow$ {Def. of $\xrightarrow{\gamma(i,u)}$}

$\exists s'.\ s \xrightarrow{\gamma(i,u)} s'$

$\Leftrightarrow$ {Def. of lan}

$\gamma(i,u) \in (\mathsf{lan}\ s)$ □

Proof of Theorem 5.1

Let $\mathcal{S} = {}^{\Sigma}\mathrm{LTS}(H)$ and $L = L_I \times L_U$. We proceed separately in the proof of soundness and exhaustiveness.

Part 1: - soundness

Suppose there exists test case $\mathcal{U} \in \mathsf{TD}_{L,L'}\ (\mathsf{lan}\ \mathcal{S})$ and implementation $\mathcal{S}'$ over $L' = L'_I \times L'_U$ such that $\mathcal{S}' \sqsubseteq_{\mathsf{co}} \mathcal{S}$ and $\mathcal{V}\ \mathcal{U}\ \mathcal{S}' = \mathbf{fail}$. Using Lemma B.6 with $\mathsf{lan}\ \mathcal{S}$ for $\mathcal{F}$, we know that this would imply that there exists $\gamma \in\ (\mathsf{lan}\ \mathcal{S})$, $i \in L_I$ and $u \in L'_U$ such that $u \in \mathsf{OUT}\ \mathcal{S}'\ \gamma\ i$ and $u \notin \mathsf{OUT}^*\ (\mathsf{lan}\ \mathcal{S})\ \gamma\ i$. But then, by Lemma B.12, we get also $u \notin \mathsf{OUT}\ \mathcal{S}\ \gamma\ i$ which contradicts $\mathcal{S}' \sqsubseteq_{\mathsf{co}} \mathcal{S}$.

Part 2: - exhaustiveness

Suppose $\mathcal{S}' \not\sqsubseteq_{\mathsf{co}} \mathcal{S}$, for implementation $\mathcal{S}'$ over $L' = L'_I \times L'_U$. This means that there exist $\gamma \in (\mathsf{lan}\ \mathcal{S})$, $i \in L_I$, $u \in L'_U$ such that $u \in \mathsf{OUT}\ \mathcal{S}'\ \gamma\ i \setminus \mathsf{OUT}\ \mathcal{S}\ \gamma\ i$. Moreover, $\mathsf{OUT}^*\ (\mathsf{lan}\ \mathcal{S})\ \gamma\ i \neq \emptyset$ because $\mathcal{S}$ is input enabled and $\mathsf{OUT}^*\ (\mathsf{lan}\ \mathcal{S})\ \gamma\ i = \mathsf{OUT}\ \ \mathcal{S}\ \gamma\ i$ by Lemma B.12. Thus we can apply Lemma B.7 with $\mathsf{lan}\ \mathcal{S}$ for $\mathcal{F}$ to get the assert. □

Lemma B.6

Let $L = L_I \times L_U$ and $L' = L'_I \times L'_U$. For all $\mathcal{F} \subseteq L^*$, implementation $\mathcal{S}' = (S', s'_{in}, L', \longrightarrow)$, $\mathcal{U} \in \mathsf{TD}_{L,L'}\ \mathcal{F}$, unsuccessful computation $\eta \in \mathrm{Comp}(\mathcal{U}, \mathcal{S}')$ there exist $\gamma \in \mathcal{F}$, $i \in L_I$, $u \in L'_U$ such that η runs over $\gamma(i,u)$ and $u \in (\mathsf{OUT}\ s'_{in}\ \gamma\ i) \setminus (\mathsf{OUT}^*\ \mathcal{F}\ \gamma\ i)$ ◇

Proof

By induction on the structure of $\mathcal{U}$

Base case ($\mathcal{U} = \tau; \mathbf{W}; \delta$):

There is no unsuccessful computation in $\mathrm{Comp}(\tau; \mathbf{W}; \delta, \mathcal{S}')$ so the assert is trivially proven.

Induction step: ($\mathcal{U} = \bar{i}; \lambda x \ldots$):

We can assume by the Induction Hypothesis that there exists $\bar{i}$ such that $\bar{i} \in L_I \cap L'_I$, $\mathsf{OUT}^*\ \mathcal{F}\ \epsilon\ \bar{i} = \{u_1 \ldots u_k\} \neq \emptyset$ and $\mathcal{U} = \bar{i}; \bar{\mathcal{I}}$ with

$$\begin{array}{lcll} \bar{\mathcal{I}} & = & \lambda x : L'_U. & x = u_1\ \Rightarrow\ \mathcal{U}_1 \\ & & & + \\ & & & \vdots \\ & & & + \\ & & & x = u_k\ \Rightarrow\ \mathcal{U}_k \\ & & & + \\ & & & x \notin \{u_1, \ldots, u_k\}\ \Rightarrow\ \delta \end{array}$$

where $\mathcal{U}_j \in \mathsf{TD}_{L,L'}\ (\mathcal{F}\ \mathsf{after}^*\ (\bar{i}, u_j))$ for $j = 1, \ldots, k$. So every (unsuccessful) computation $\eta \in \mathrm{Comp}(\mathcal{U}, \mathcal{S}')$ must have the form

$$\eta = \mathcal{U} \mid\mid s'_{\mathrm{in}} \rightsquigarrow (\bar{\mathcal{I}}\ \bar{u}) \mid\mid s' \rightsquigarrow \ldots$$

for some $\bar{u} \in L'_U$ and s' such that $s'_{\text{in}} \xrightarrow{(\bar{i},\bar{u})} s'$. Notice that such $\bar{u}$ and s' exist since $\mathcal{S}'$ is input enabled over L' and $\bar{i} \in L'_I$. We distinguish two cases:

Case 1: $\bar{u} \notin \mathsf{OUT}^*\ \mathcal{F}\ \epsilon\ \bar{i}$
For every such $\bar{u} \notin \mathsf{OUT}^*\ \mathcal{F}\ \epsilon\ \bar{i}$ and s' such that $s'_{\text{in}} \xrightarrow{(\bar{i},\bar{u})} s'$ there is only the unsuccessful computation

$$\eta_1 = \mathcal{U} \mid\mid s'_{\text{in}} \rightsquigarrow (\delta \mid\mid s')$$

since in this case $\bar{\mathcal{I}}\bar{u} = \delta$, by definition of $\mathcal{U}$. Thus the assert is proven with $\gamma = \epsilon$, $i = \bar{i}$, $u = \bar{u}$: $\bar{u} \in (\mathsf{OUT}\ \mathcal{S}'\ \epsilon\ \bar{i})$, since $s'_{\text{in}} \xrightarrow{(\bar{i},\bar{u})} s'$, and $\bar{u} \notin \mathsf{OUT}^*\ \mathcal{F}\ \epsilon\ \bar{i}$

Case 2: $\bar{u} \in \mathsf{OUT}^*\ \mathcal{F}\ \epsilon\ \bar{i}$
For every η as above we know that its continuation

$$\eta_2 = (\bar{\mathcal{I}}\bar{u}) \mid\mid s' \rightsquigarrow \ldots$$

is an unsuccessful computation in $\text{Comp}(\bar{\mathcal{U}}, s')$ where $\bar{\mathcal{U}} = \bar{\mathcal{I}}\ \bar{u}$ are elements of $\mathsf{TD}_{L,L'}(\mathcal{F}\mathsf{after}^*(\bar{i},\bar{u}))$ and s' input enabled over L'. Thus, for every $\bar{u}$ and s' as above we can apply the Induction Hypothesis with $\mathcal{F}\ \mathsf{after}^*\ (\bar{i},\bar{u}), s', \bar{\mathcal{U}}$ and find $\bar{\gamma} \in \mathcal{F}\ \mathsf{after}^*\ (\bar{i},\bar{u})$, $i \in L_I$ and $u \in L'_U$ such that η_2 runs over $\bar{\gamma}(i,u)$ and $u \in \mathsf{OUT}\ s'\ \bar{\gamma}\ i$ but $u \notin \mathsf{OUT}^*\ (\mathcal{F}\ \mathsf{after}^*\ (\bar{i},\bar{u}))\ \bar{\gamma}\ i$.
We now observe that

- $(\bar{i},\bar{u})\bar{\gamma} \in \mathcal{F}$, by def. of after^*, since $\bar{\gamma} \in \mathcal{F}\ \mathsf{after}^*\ (\bar{i},\bar{u})$,
- η runs over $(\bar{i},\bar{u})\bar{\gamma}(i,u)$, since $\eta = \mathcal{U} \mid\mid \mathcal{S}' \rightsquigarrow \eta_2, \mathcal{U} \xrightarrow{\bar{i}} \bar{\mathcal{I}}, s'_{\text{in}} \xrightarrow{(\bar{i},\bar{u})} s', \bar{\mathcal{U}} = \bar{\mathcal{I}}\bar{u}$ and η_2 runs over $\bar{\gamma}(i,u)$,
- $u \in \mathsf{OUT}\ \mathcal{S}'\ (\bar{i},\bar{u})\bar{\gamma}\ i$, by Lemma B.9, since $s'_{\text{in}} \xrightarrow{(\bar{i},\bar{u})} s'$ and $u \in \mathsf{OUT}\ s'\ \bar{\gamma}\ i$
- $u \notin \mathsf{OUT}^*\ \mathcal{F}\ (\bar{i},\bar{u})\bar{\gamma}\ i$, by Lemma B.14, since u is not an element of $\mathsf{OUT}^*\ (\mathcal{F}\ \mathsf{after}^*\ (\bar{i},\bar{u}))\ \bar{\gamma}\ i$

which proves the assert with $\gamma = (\bar{i},\bar{u})\bar{\gamma}$ and i and u as above. □

Lemma B.7
*Let $L = L_I \times L_U$ and $L' = L'_I \times L'_U$. For all $\mathcal{F} \subseteq L^*$, $i \in L_I$, $\gamma \in \mathcal{F}$ such that $\mathsf{OUT}^*\ \mathcal{F}\ \gamma\ i \neq \emptyset$, implementation $\mathcal{S}' = (S', s'_{in}, L', \longrightarrow)$, and $u \in L'_U$ such that $u \in (\mathsf{OUT}\ \mathcal{S}'\ \gamma\ i) \setminus (\mathsf{OUT}^*\ \mathcal{F}\ \gamma\ i)$, there exists $\mathcal{U} \in \mathsf{TD}_{L,L'}\ \mathcal{F}$ such that $\mathcal{V}\ \mathcal{U}\ \mathcal{S}' =$ **fail*** ◇

Proof

We proceed by induction on γ.
Base case ($\gamma = \epsilon$):
By hypothesis we know that $i \in L_I$ and $\mathsf{OUT}^*\ \mathcal{F}\ \epsilon\ i \neq \emptyset$; moreover $i \in L'_I$ by Lemma B.5 since $(\mathsf{OUT}\ \mathcal{S}'\ \gamma\ i) \neq \emptyset$ and $\mathcal{S}'$ is an LTS labeled over L'. Thus the following test case $\mathcal{U}$ belongs to $\mathsf{TD}_{L,L'}\ \mathcal{F}$: $\mathcal{U} = i;\bar{\mathcal{I}}$ with

$$\begin{array}{lcll} \bar{\mathcal{I}} & = & \lambda x : L'_U. & x = u_1 \ \Rightarrow\ \mathcal{U}_1 \\ & & & + \\ & & & \vdots \\ & & & + \\ & & & x = u_k \ \Rightarrow\ \mathcal{U}_k \\ & & & + \\ & & & x \notin \{u_1, \ldots, u_k\} \ \Rightarrow\ \delta \end{array}$$

where $\{u_1, \ldots, u_k\} = \mathsf{OUT}^*\ \mathcal{F}\ \epsilon\ i$ and $\mathcal{U}_j$ is an element of $\mathsf{TD}_{L,L'}\ (\mathcal{F}\ \mathsf{after}^*\ (i, u_j))$ for $j = 1, \ldots, k$. Moreover $\bar{\mathcal{I}} u = \delta$, since by hypothesis $u \notin \mathsf{OUT}^*\ \mathcal{F}\ \epsilon\ i$, and $s'_{\text{in}} \xrightarrow{(i,u)} s'$ for some s', since $u \in \mathsf{OUT}\ s'_{\text{in}}\ \epsilon\ i$. Thus we can build the following unsuccessful computation

$$\mathcal{U} \mid\mid s'_{\text{in}} \rightsquigarrow (\bar{\mathcal{I}} u) \mid\mid s'$$

which makes $\mathcal{V}\ \mathcal{U}\ \mathcal{S}' = \textbf{fail}$ hold.

Induction step ($\gamma = (\bar{i}, \bar{u})\bar{\gamma}$):
We know that

- $\bar{i} \in L_I$, since $(\bar{i}, \bar{u})\bar{\gamma} \in \mathcal{F} \subseteq L^*$ by hypothesis,
- $\bar{i} \in L'_I$, by Lemma B.11, since $\mathsf{OUT}\ \mathcal{S}'\ (\bar{i}, \bar{u})\bar{\gamma}\ i \neq \emptyset$ by hypothesis,
- $\mathsf{OUT}^*\ \mathcal{F}\ \epsilon\ \bar{i} \neq \emptyset$, by Lemma B.13, since $\mathsf{OUT}^*\ \mathcal{F}\ (\bar{i}, \bar{u})\bar{\gamma}\ i \neq \emptyset$, by hypothesis.

Thus the following test case $\mathcal{U}$ belongs to $\mathsf{TD}_{L,L'}\ \mathcal{F}$: $\mathcal{U} = \bar{i}; \bar{\mathcal{I}}$ with

$$\begin{array}{lcll} \bar{\mathcal{I}} & = & \lambda x : L'_U. & x = u_1 \ \Rightarrow\ \mathcal{U}_1 \\ & & & + \\ & & & \vdots \\ & & & + \\ & & & x = u_k \ \Rightarrow\ \mathcal{U}_k \\ & & & + \\ & & & x \notin \{u_1, \ldots, u_k\} \ \Rightarrow\ \delta \end{array}$$

where $\{u_1, \ldots, u_k\} = \mathsf{OUT}^*\ \mathcal{F}\ \epsilon\ \bar{i}$ and $\mathcal{U}_j$ is an element of $\mathsf{TD}_{L,L'}\ (\mathcal{F}\ \mathsf{after}^*\ (\bar{i}, u_j))$ for $j = 1, \ldots, k$. We observe now that

- $\bar{u} \in \mathsf{OUT}^*\ \mathcal{F}\ \epsilon\ \bar{i}$, by Lemma B.13, since $\mathsf{OUT}^*\ \mathcal{F}\ (\bar{i}, \bar{u})\bar{\gamma}\ i \neq \emptyset$ by hypothesis,
- $\bar{\gamma} \in \mathcal{F}\ \mathsf{after}^*\ (\bar{i}, \bar{u})$, by definition of after^*, since $(\bar{i}, \bar{u})\bar{\gamma} \in \mathcal{F}$ by hypothesis,
- $\mathsf{OUT}^*\ (\mathcal{F}\ \mathsf{after}^*\ (\bar{i}, \bar{u}))\ \bar{\gamma}\ i \neq \emptyset$,by Lemma B.14, since $\mathsf{OUT}^*\ \mathcal{F}\ (\bar{i}, \bar{u})\bar{\gamma}\ i \neq \emptyset$ by hypothesis.

Thus, using the Induction Hypothesis, we know that for every s' such that $s'_{\text{in}} \xrightarrow{(\bar{i}, \bar{u})} s'$ and u element of $\mathsf{OUT}\ s'\ \bar{\gamma}\ i \setminus \mathsf{OUT}^*\ (\mathcal{F}\ \mathsf{after}^*(\bar{i}, \bar{u}))\ \bar{\gamma}\ i$, there exists $\bar{\mathcal{U}}$ element of
$\mathsf{TD}_{L,L'}(\mathcal{F}\ \mathsf{after}^*\ (\bar{i}, \bar{u}))$ such that $\mathcal{V}\ \bar{\mathcal{U}}\ \bar{s}' = \textbf{fail}$, which means that there exists an unsuccessful computation

$$\bar{\mathcal{U}} \mid\mid s' \rightsquigarrow \ldots$$

But this in turn implies that for every such $(\bar{i}, \bar{u})$ and every $u \in \mathsf{OUT}\ \mathcal{S}'\ (\bar{i}, \bar{u})\bar{\gamma}\ i \setminus \mathsf{OUT}^*\ \mathcal{F}\ (\bar{i}, \bar{u})\bar{\gamma}\ i$ there is an unsuccessful computation

$$\mathcal{U} \mid\mid s'_{\text{in}} \rightsquigarrow \bar{\mathcal{U}} \mid\mid s' \rightsquigarrow \ldots$$

in $\text{Comp}(\mathcal{U}, \mathcal{S}')$. This last fact follows from the fact that $\mathsf{OUT}\ \mathcal{S}'\ (\bar{i}, \bar{u})\bar{\gamma}\ i = \bigcup_{s' : s'_{\text{in}} \xrightarrow{(\bar{i}, \bar{u})} s'} \mathsf{OUT}\ s'\ \bar{\gamma}\ i$, by Lemma B.9, and $\mathsf{OUT}^*\ (\mathcal{F}\ \mathsf{after}^*\ (\bar{i}, \bar{u}))\ \bar{\gamma}\ i = \mathsf{OUT}^*\ \mathcal{F}\ (\bar{i}, \bar{u})\bar{\gamma}\ i$, by Lemma B.14. So, $\mathcal{V}\ \mathcal{U}\ \mathcal{S}' = \textbf{fail}$. □

General properties of operators used in the above proofs

The following lemmas follow from the relevant definitions. The detailed proofs are provided in [14].

Lemma B.8 *For all LTS* $(S, s_{in}, L, \longrightarrow)$, $s \in S$, $\gamma, \gamma' \in L^*$ *the following holds:* s *after* $\gamma\gamma' = \bigcup_{s':s \xrightarrow{\gamma} s'} s'$ *after* γ' $\diamond$

Lemma B.9 *For all LTS* $(S, s_{in}, L_I \times L_U, \longrightarrow)$, $s \in S$, $\gamma, \gamma' \in (L_I \times L_U)^*$, $i \in L_I$ *the following holds:* $\mathsf{OUT}\ s\ \gamma\gamma'\ i = \bigcup_{s':s \xrightarrow{\gamma} s'} \mathsf{OUT}\ s'\ \gamma'\ i$ $\diamond$

Lemma B.10 *For all LTS* $(S, s_{in}, L_I \times L_U, \longrightarrow)$, $Z_1, Z_2 \subseteq S$, $i \in L_I$ *the following holds:* $\mathsf{out}\ (Z_1 \cup Z_2)\ i = (\mathsf{out}\ Z_1\ i) \cup (\mathsf{out}\ Z_2\ i)$ $\diamond$

Lemma B.11 *For all LTS* $\mathcal{S} = (S, s_{in}, L_I \times L_U, \longrightarrow)$, $s \in S$, $i \in L_I \subseteq X$ *for some* X , $u \in L_U$, $\gamma \in (L_I \times L_U)^*$ *and* $i \in X$ *the following holds:* $\mathsf{OUT}\ s\ (i,u)\gamma\ i' \neq \emptyset \Rightarrow i \in L_I$ $\diamond$

Lemma B.12 *For all LTS* $\mathcal{S} = (S, s_{in}, L_I \times L_U, \longrightarrow)$, $\gamma \in (\mathsf{Ian}\ \mathcal{S})$, $i \in L_I$ *the following holds:* $\mathsf{OUT}^*\ (\mathsf{Ian}\ \mathcal{S})\ \gamma\ i = \mathsf{OUT}\ s_{in}\ \gamma\ i$ $\diamond$

Lemma B.13 *For all* $\mathcal{F} \subseteq (L_I \times L_U)^*$ $i, i' \in L_I$, $u \in L_U$, $\gamma \in (L_I \times L_U)^*$ *the following holds:* $\mathsf{OUT}^*\ \mathcal{F}\ (i,u)\gamma\ i' \neq \emptyset \Rightarrow u \in \mathsf{OUT}^*\ \mathcal{F}\ \epsilon\ i$ $\diamond$

Lemma B.14
For all $\mathcal{F} \subseteq (L_I \times L_U)^*$, $\gamma, \gamma' \in (L_I \times L_U)^*$, $i \in L_I$ *the following holds:* $\mathsf{OUT}^*\ (\mathcal{F}\ \mathsf{after}^*\ \gamma)\ \gamma'\ i = \mathsf{OUT}^*\ \mathcal{F}\ \gamma\gamma'\ i$ $\diamond$

Lemma B.15 *For all* $\mathcal{F} \subseteq (L_I \times L_U)^*$, $\gamma, \gamma' \in (L_I \times L_U)^*$, *the following holds:* $(\mathcal{F}\ \mathsf{after}^*\ \gamma)\ \mathsf{after}^*\ \gamma' = \mathcal{F}\ \mathsf{after}^*\ \gamma\gamma'$ $\diamond$

B.3 Proofs related to Sect. 6

Proof of Theorem 6.1

Part i -

$\mathcal{C} \xrightarrow{e/\mathcal{E}} \mathcal{C}'$

$\Leftrightarrow$ {Def. 4.2}

$\exists \mathcal{L} \neq \emptyset.\ H \uparrow \emptyset :: (\mathcal{C}, \{e\}) \xrightarrow{\mathcal{L}} (\mathcal{C}', \mathcal{E})$

$\Leftrightarrow$ {Rule (1) of Def. 5.3}

$\mathcal{C} \xrightarrow{e/\mathcal{E}}_{\Sigma} \mathcal{C}'$

Part ii -

$\nexists \mathcal{C}' \in \mathrm{Conf}_H, \mathcal{E} \in \Theta_E.\ \mathcal{C} \xrightarrow{e/\mathcal{E}} \mathcal{C}'$

$\Leftrightarrow$ {Def. 4.2; Logics}

$\nexists \mathcal{L} \subseteq \mathsf{Tr}\ H, \mathcal{C}' \in \mathrm{Conf}_H, \mathcal{E} \in \Theta_E.\ H \uparrow \emptyset :: (\mathcal{C}, \{e\}) \xrightarrow{\mathcal{L}} (\mathcal{C}', \mathcal{E})$

$\vee$

$\exists \mathcal{C}' \in \mathrm{Conf}_H, \mathcal{E} \in \Theta_E.\ H \uparrow \emptyset :: (\mathcal{C}, \{e\}) \xrightarrow{\emptyset} (\mathcal{C}', \mathcal{E})$

$\Leftrightarrow$ {Lemma A.1 (i) and (ii)}

$H \uparrow \emptyset :: (\mathcal{C}, \{e\}) \xrightarrow{\emptyset} (\mathcal{C}, \langle\rangle)$

$\Rightarrow$ {Rule (2) of Def. 5.3}

$\Leftarrow$ {Rule (2) of Def. 5.3; Lemma A.1 (ii)}

$\mathcal{C} \xrightarrow{e/\Sigma}_{\Sigma} \mathcal{C}$ □

Proof of Lemma 6.1

Part i -

Suppose ${}^{\Sigma}\mathcal{C} \xrightarrow{\gamma}_{\Sigma} {}^{\Sigma}\mathcal{C}'$. Without loss of generality assume $\gamma = \alpha_0\beta_0\alpha_1\beta_1 \ldots \alpha_n\beta_n$ with $\alpha_i \in (E \times \Theta_E)^*$, $\beta_i \in (E \times \{\Sigma\})^*$ for $i = 0, \ldots, n$. In particular, this means that there exist ${}^{\Sigma}\mathcal{C}_0, {}^{\Sigma}\mathcal{C}_1, \ldots, {}^{\Sigma}\mathcal{C}_{n+1}$ such that

$$ {}^{\Sigma}\mathcal{C}_0 = {}^{\Sigma}\mathcal{C},\ {}^{\Sigma}\mathcal{C}_{n+1} = {}^{\Sigma}\mathcal{C}' $$

$$ {}^{\Sigma}\mathcal{C}_0 \xrightarrow{\alpha_0}_{\Sigma} {}^{\Sigma}\mathcal{C}_1 \xrightarrow{\beta_0}_{\Sigma} {}^{\Sigma}\mathcal{C}_1 \xrightarrow{\alpha_1}_{\Sigma} \ldots \xrightarrow{\beta_{n-1}}_{\Sigma} {}^{\Sigma}\mathcal{C}_n \xrightarrow{\alpha_n}_{\Sigma} {}^{\Sigma}\mathcal{C}_{n+1} \xrightarrow{\beta_n}_{\Sigma} {}^{\Sigma}\mathcal{C}_{n+1} $$

Notice that ${}^{\Sigma}\mathcal{C}_i \xrightarrow{\beta_{i-1}}_{\Sigma} {}^{\Sigma}\mathcal{C}_i$ i.e., the configuration does not change by performing β_{i-1}, due to Lemma 5.1(ii). Thus, from the above sequence of transitions we easily get the following one:

$$ {}^{\Sigma}\mathcal{C}_0 \xrightarrow{\alpha_0}_{\Sigma} {}^{\Sigma}\mathcal{C}_1 \xrightarrow{\alpha_1}_{\Sigma} \ldots \xrightarrow{\alpha_{n-1}}_{\Sigma} {}^{\Sigma}\mathcal{C}_n \xrightarrow{\alpha_n}_{\Sigma} {}^{\Sigma}\mathcal{C}_{n+1} $$

But then, noting that $\gamma \setminus \Sigma = \alpha_0\alpha_1 \ldots \alpha_n \in (E \times \Theta_E)^*$ and using Corollary 6.1, we get easily $\mathcal{C} \xrightarrow{\gamma\setminus\Sigma} \mathcal{C}'$

Part ii -

Let $\mathsf{LTS}(H) = (\mathrm{Conf}_H, \mathcal{C}_{\mathrm{in}}, \rightarrow)$ and ${}^{\Sigma}\mathsf{LTS}(H) = (\mathrm{Conf}_H, {}^{\Sigma}\mathcal{C}_{\mathrm{in}}, \rightarrow_{\Sigma})$

$\gamma \in (\mathsf{lan}\ {}^{\Sigma}\mathsf{LTS}(H))$

$\Rightarrow$ {Def. of lan}

$\exists \mathcal{C}'.\ {}^{\Sigma}\mathcal{C}_{\mathrm{in}} \xrightarrow{\gamma}_{\Sigma} {}^{\Sigma}\mathcal{C}'$

$\Rightarrow$ {Part (i) of this lemma}

$\exists \mathcal{C}'.\ (\mathcal{C}_{\mathrm{in}} \xrightarrow{\gamma\setminus\Sigma} \mathcal{C}')$

$\Rightarrow$ {Corollary 6.1; $\gamma \setminus \Sigma \in (E \times \Theta_E)^*$}

$\exists \mathcal{C}'.\ (\mathcal{C}_{\mathrm{in}} \xrightarrow{\gamma\setminus\Sigma} \mathcal{C}') \wedge ({}^{\Sigma}\mathcal{C}_{\mathrm{in}} \xrightarrow{\gamma\setminus\Sigma}_{\Sigma} {}^{\Sigma}\mathcal{C}')$

$\Rightarrow$ {Def. of lan; Logics}

$\gamma \setminus \Sigma \in (\mathsf{lan}\ \mathsf{LTS}(H)) \cap (\mathsf{lan}\ {}^{\Sigma}\mathsf{LTS}(H))$

Part iii -

Directly follows from Corollary 6.1. □

Proof of Lemma 6.2

We prove $\mathsf{lan}\ \mathsf{LTS}(H') \subseteq \mathsf{lan}\ \mathsf{LTS}(H)$, that is $\mathsf{LTS}(H') <<_{MAY} \mathsf{LTS}(H)$, which, due to Theorem 4.1 is equivalent to $\mathsf{LTS}(H') \sqsubseteq_{MAY} \mathsf{LTS}(H)$.

$\gamma \in \mathsf{lan}\ \mathsf{LTS}(H')$

$\Rightarrow$ {Lemma 6.1 (iii)}

$\gamma \in \mathsf{lan}\ {}^{\Sigma}\mathsf{LTS}(H')$

$\Rightarrow$ {Lemma 5.3; ${}^{\Sigma}\mathsf{LTS}(H') \sqsubseteq_{\mathsf{co}} {}^{\Sigma}\mathsf{LTS}(H)$ and $E' \subseteq E$ by hypothesis}

$\gamma \in \mathsf{lan}\ {}^{\Sigma}\mathsf{LTS}(H)$

$\Rightarrow$ {Lemma 6.1 (ii); $\gamma \in \mathsf{lan}\ \mathsf{LTS}(H')$ implies $\gamma \setminus \Sigma = \gamma$}

$\gamma \in \mathsf{lan}\ \mathsf{LTS}(H)$ □

Proof of Lemma 6.3

We let $\mathsf{LTS}(H) = (\mathrm{Conf}_H, \mathcal{C}_{\mathrm{in}}, \rightarrow)$, $\mathsf{LTS}(H') = (\mathrm{Conf}_{H'}, \mathcal{C}'_{\mathrm{in}}, \rightarrow)$, ${}^{\Sigma}\mathsf{LTS}(H) = (\mathrm{Conf}_H, {}^{\Sigma}\mathcal{C}_{\mathrm{in}}, \rightarrow_{\Sigma})$, and ${}^{\Sigma}\mathsf{LTS}(H') = (\mathrm{Conf}_{H'}, {}^{\Sigma}\mathcal{C}'_{\mathrm{in}}, \rightarrow_{\Sigma})$.

We proceed by contradiction. Suppose ${}^{\Sigma}\mathsf{LTS}(H') \sqsubseteq_{\mathsf{co}} {}^{\Sigma}\mathsf{LTS}(H)$ that is there is an experimenter $\mathcal{U}$ such that $\perp \notin \mathrm{Result}(\mathcal{U}, \mathsf{LTS}(H))$ and $\perp \in \mathrm{Result}(\mathcal{U}, \mathsf{LTS}(H'))$. Let $\eta = \mathcal{U} \mid\mid \mathcal{C}'_{\mathrm{in}} \rightsquigarrow \ldots$ be an unsuccessful computation in $\mathrm{Comp}(\mathcal{U}, \mathsf{LTS}(H'))$. We distinguish two cases according to η.

Case 1: η is finite
W.l.g. let us assume $\eta = \mathcal{U} \mid\mid \mathcal{C}'_{\mathrm{in}} \rightsquigarrow \ldots \rightsquigarrow \mathcal{U}_n \mid\mid \mathcal{C}'$ and there exist no $\mathcal{U}_{n+1} \mid\mid \mathcal{C}''$ such that $\mathcal{U}_n \mid\mid \mathcal{C}' \rightsquigarrow \mathcal{U}_{n+1} \mid\mid \mathcal{C}''$. In this case we have a derivation $\mathcal{C}'_{\mathrm{in}} \xrightarrow{\gamma} \mathcal{C}'$ on the side of $\mathsf{LTS}(H')$, with $\gamma = (e_1, \mathcal{E}_1) \ldots (e_k, \mathcal{E}_k) \in (E \times \Theta_E)^*$, and a sequence of transitions $\mathcal{U}_j \xrightarrow{\mu_j} \mathcal{O}_j$, for $j = 1 \ldots n-1$, such that either $\mu_j = e_i$ and $\mathcal{U}_{j+1} = (\mathcal{O}_j\ \mathcal{E}_i)$, for some i with $1 \leq i \leq k$, or $\mu_j = \tau$ and $\mathcal{U}_{j+1} = \mathcal{O}_j$. Notice that the derivation on the side of the experimenter involves a sequence γ' which is equal to γ up to τ moves.

First of all, notice that it cannot be $\mathcal{U}_n \xrightarrow{\tau}$ since otherwise η could not be a computation, not being maximal. There are two other possibilities left[8] :

Case 1.1: $\forall e \in E.\ \mathcal{U}_n \xrightarrow{e} \Rightarrow \nexists \mathcal{E} \in \Theta_E.\ \mathcal{C}' \xrightarrow{e/\mathcal{E}}$
By Lemma 6.2[9] we know that $\gamma \in \mathsf{lan}\ \mathsf{LTS}(H)$ since $\gamma \in \mathsf{lan}\ \mathsf{LTS}(H')$, by definition of $\mathsf{lan}\ \mathsf{LTS}(H')$ and ${}^{\Sigma}\mathsf{LTS}(H') \sqsubseteq_{\mathsf{co}} {}^{\Sigma}\mathsf{LTS}(H)$ by hypothesis. Moreover, by Lemma 6.1 (iii), we also know that $\gamma \in \mathsf{lan}\ {}^{\Sigma}\mathsf{LTS}(H)$. Thus, again by ${}^{\Sigma}\mathsf{LTS}(H') \sqsubseteq_{\mathsf{co}} {}^{\Sigma}\mathsf{LTS}(H)$, we get

$$\mathsf{OUT}\ {}^{\Sigma}\mathsf{LTS}(H')\ \gamma\ e \subseteq \mathsf{OUT}\ {}^{\Sigma}\mathsf{LTS}(H)\ \gamma\ e \tag{2}$$

since $\gamma \in \mathsf{lan}\ {}^{\Sigma}\mathsf{LTS}(H)$ and $e \in E$[10].

Moreover, by hypothesis we know that there is no $\mathcal{E}$ such that $\mathcal{C}' \xrightarrow{e/\mathcal{E}}$, so, by Theorem 6.1, we also know that ${}^{\Sigma}\mathcal{C}' \xrightarrow{e/\Sigma}_{\Sigma}$. Moreover, ${}^{\Sigma}\mathcal{C}'_{\mathrm{in}} \xrightarrow{\gamma}_{\Sigma} {}^{\Sigma}\mathcal{C}'$, by Corollary 6.1, since $\mathcal{C}'_{\mathrm{in}} \xrightarrow{\gamma} \mathcal{C}'$ and $\gamma \in (E \times \Theta_E)^*$. By definition of OUT, we get $\Sigma \in \mathsf{OUT}\ {}^{\Sigma}\mathsf{LTS}(H')\ \gamma\ e$ so, using relation (2) above we can conclude $\Sigma \in \mathsf{OUT}\ {}^{\Sigma}\mathsf{LTS}(H)\ \gamma\ e$. But then, again by definition of OUT, we derive that there exists ${}^{\Sigma}\mathcal{C}$ such that ${}^{\Sigma}\mathcal{C}_{\mathrm{in}} \xrightarrow{\gamma}_{\Sigma} {}^{\Sigma}\mathcal{C}$ and ${}^{\Sigma}\mathcal{C} \xrightarrow{e/\Sigma}_{\Sigma}$, and again by Theorem 6.1 and its corollary we get $\mathcal{C} \xrightarrow{e/\mathcal{E}}$ for no $\mathcal{E} \in \Theta_E$ and $\mathcal{C}_{\mathrm{in}} \xrightarrow{\gamma} \mathcal{C}$. This means that we can build the following computation $\mathcal{U} \mid\mid \mathcal{C}_{\mathrm{in}} \rightsquigarrow \ldots \mathcal{U}_n \mid\mid \mathcal{C}$, which is an unsuccessful computation since η above was so. This contradicts $\perp \notin \mathrm{Result}(\mathcal{U}, \mathsf{LTS}(H))$

Case 1.2: $\mathcal{U}_n \xrightarrow{e}$ for no $e \in E$
By Lemma 6.2 we know that $\gamma \in \mathsf{lan}\ \mathsf{LTS}(H)$ since $\gamma \in \mathsf{lan}\ \mathsf{LTS}(H')$ and ${}^{\Sigma}\mathsf{LTS}(H') \sqsubseteq_{\mathsf{co}} {}^{\Sigma}\mathsf{LTS}(H)$. This means, by definition of $\mathsf{lan}\ \mathsf{LTS}(H)$, that $\mathcal{C}_{\mathrm{in}} \xrightarrow{\gamma} \mathcal{C}$ for some $\mathcal{C}$. But then we can build the following computation $\mathcal{U} \mid\mid \mathcal{C}_{\mathrm{in}} \rightsquigarrow \ldots \mathcal{U}_n \mid\mid \mathcal{C}$, which is an unsuccessful computation since η above was so. This contradicts $\perp \notin \mathrm{Result}(\mathcal{U}, \mathsf{LTS}(H))$

Case 2: η is infinite
Also in this case the computation gives rise to one derivation on the side of $\mathsf{LTS}(H')$, $\mathcal{C}'_{\mathrm{in}} \xrightarrow{\gamma}$ and to a sequence of transitions $\mathcal{U}_j \xrightarrow{\mu_j} \mathcal{O}_j$ on the side of $\mathcal{U}$, which involves infinite string $\gamma' \in (E \times \Theta_E)^{\infty}$, which is equal to γ up to τ moves. We distinguish two cases:

[8] Notice that $E \subseteq E'$ is necessary otherwise considering only cases 1.1 and 1.2. would not be enough. In particular, it could be the case that $\mathcal{U}_n \xrightarrow{e'}$ for some $e' \in E \setminus E'$ (notice that in this case $\mathcal{U}$ should be over $E \cup E'$) and $\mathcal{U}_n \not\xrightarrow{e}$ for all $e \in E'$. This would mean that η would be maximal but we could not infer ${}^{\Sigma}\mathcal{C}_{\mathrm{in}} \xrightarrow{e'/\Sigma}_{\Sigma}$ and in fact it could very well be ${}^{\Sigma}\mathcal{C}_{\mathrm{in}} \xrightarrow{e'/\mathcal{E}'}_{\Sigma}$ for some $\mathcal{E}' \in \Theta_E$ and this extra step could bring to success so we would not reach contradiction.

[9] Here we need $E' \subseteq E$.

[10] Notice that we can say $e \in E$ because $E = E'$. Otherwise the hypothesis would be $e \in E'$ and here we would need again $E' \subseteq E$.

Case 2.1: $\forall n \geq 0.\ \exists m \geq n.$ the m-th element of γ' is not τ
From the operational semantics rules of experimental systems (Def. 3.5) we get that also γ is infinite. For each finite prefix $\bar{\gamma}$ of γ, by Lemma 6.2 we get $\bar{\gamma} \in \mathsf{lan}\ \mathsf{LTS}(H)$ since ${}^{\Sigma}\mathsf{LTS}(H') \sqsubseteq_{\mathsf{co}} {}^{\Sigma}\mathsf{LTS}(H)$ by hypothesis. By definition of $\mathsf{lan}\ \mathsf{LTS}(H)$ this means $\mathcal{C}_{\mathrm{in}} \xrightarrow{\bar{\gamma}}$. Thus we can build infinite computation $\mathcal{U} \mid\mid \mathcal{C}_{\mathrm{in}} \rightsquigarrow \ldots$, using, in each step j exactly the same $\mathcal{U}_j$ appearing in η and the same prefix of γ on which η is running up to step j. Notice also that there can be more than one successive steps using the same prefix $\bar{\gamma}$ due to the fact that $\mathcal{U}_j \xrightarrow{\tau}$ may hold. In conclusion, also in this case we reach a contradiction since the computation we can build involves the same $\mathcal{U}_j$, for $j \geq 0$, occurring in η, which is unsuccessful.

Case 2.2: $\exists n \geq 0.\ \forall m \geq n.$ the m-th element of γ' is τ
In this case γ is finite and by Lemma 6.2 we get $\mathcal{C}_{\mathrm{in}} \xrightarrow{\gamma}$. Therefore we can build an unsuccessful computation $\mathcal{U} \mid\mid \mathcal{C}_{\mathrm{in}} \rightsquigarrow \ldots$ as in Case 2.1 reaching a contradiction. □

Proof of Lemma 6.4

$$
\begin{array}{ll}
 & {}^{\Sigma}\mathsf{LTS}(H) \sqsubseteq_{\sim MAY} {}^{\Sigma}\mathsf{LTS}(H') \\
\Leftrightarrow & \{\text{Theorem 4.1}\} \\
 & {}^{\Sigma}\mathsf{LTS}(H) \ <<_{MAY}\ {}^{\Sigma}\mathsf{LTS}(H') \\
\Leftrightarrow & \{\text{Def. 4.6}\} \\
 & \mathsf{lan}\ {}^{\Sigma}\mathsf{LTS}(H) \subseteq \mathsf{lan}\ {}^{\Sigma}\mathsf{LTS}(H') \\
\Rightarrow & \{\text{Lemma 5.2}\} \\
\Leftarrow & \{\text{Lemma 5.3; } E \subseteq E'\} \\
 & {}^{\Sigma}\mathsf{LTS}(H) \sqsubseteq_{\mathsf{co}} \mathsf{LTS}(H')
\end{array}
$$
□

Proof of Lemma 6.5

$$
\begin{array}{ll}
 & {}^{\Sigma}\mathsf{LTS}(H) \sqsubseteq_{\sim MUST} {}^{\Sigma}\mathsf{LTS}(H') \\
\Rightarrow & \{\text{Corollary B.1}\} \\
 & \mathsf{lan}\ {}^{\Sigma}\mathsf{LTS}(H') \subseteq \mathsf{lan}\ {}^{\Sigma}\mathsf{LTS}(H) \\
\Rightarrow & \{\text{Lemma 5.2}\} \\
 & {}^{\Sigma}\mathsf{LTS}(H') \sqsubseteq_{\mathsf{co}} {}^{\Sigma}\mathsf{LTS}(H)
\end{array}
$$
□

Proof of Theorem 6.2

Part i -

$$
\begin{array}{ll}
 & {}^{\Sigma}\mathsf{LTS}(H'') \sqsubseteq_{\sim MAY} {}^{\Sigma}\mathsf{LTS}(H') \wedge {}^{\Sigma}\mathsf{LTS}(H') \sqsubseteq_{\mathsf{co}} {}^{\Sigma}\mathsf{LTS}(H) \\
\Leftrightarrow & \{\text{Theorem 4.1 (a)}\} \\
 & \mathsf{lan}\ {}^{\Sigma}\mathsf{LTS}(H'') \subseteq \mathsf{lan}\ {}^{\Sigma}\mathsf{LTS}(H') \wedge {}^{\Sigma}\mathsf{LTS}(H') \sqsubseteq_{\mathsf{co}} {}^{\Sigma}\mathsf{LTS}(H) \\
\Rightarrow & \{\text{Lemma 5.3; } E' \subseteq E\} \\
 & \mathsf{lan}\ {}^{\Sigma}\mathsf{LTS}(H'') \subseteq \mathsf{lan}\ {}^{\Sigma}\mathsf{LTS}(H') \wedge \mathsf{lan}\ {}^{\Sigma}\mathsf{LTS}(H') \subseteq \mathsf{lan}\ {}^{\Sigma}\mathsf{LTS}(H)
\end{array}
$$

$\Rightarrow$ {Set Theory}

$\mathsf{lan}\ ^{\Sigma}\mathrm{LTS}(H'') \subseteq \mathsf{lan}\ ^{\Sigma}\mathrm{LTS}(H)$

$\Rightarrow$ {Lemma 5.2}

$^{\Sigma}\mathrm{LTS}(H'') \sqsubseteq_{\mathsf{co}} {}^{\Sigma}\mathrm{LTS}(H)$

Part ii -

$^{\Sigma}\mathrm{LTS}(H') \underset{\sim}{\sqsubset}_{MUST} {}^{\Sigma}\mathrm{LTS}(H'') \wedge {}^{\Sigma}\mathrm{LTS}(H') \sqsubseteq_{\mathsf{co}} {}^{\Sigma}\mathrm{LTS}(H)$

$\Rightarrow$ {Corollary B.1}

$\mathsf{lan}\ ^{\Sigma}\mathrm{LTS}(H'') \subseteq \mathsf{lan}\ ^{\Sigma}\mathrm{LTS}(H') \wedge {}^{\Sigma}\mathrm{LTS}(H') \sqsubseteq_{\mathsf{co}} {}^{\Sigma}\mathrm{LTS}(H)$

$\Rightarrow$ {Lemma 5.3; $E' \subseteq E$}

$\mathsf{lan}\ ^{\Sigma}\mathrm{LTS}(H'') \subseteq \mathsf{lan}\ ^{\Sigma}\mathrm{LTS}(H') \wedge \mathsf{lan}\ ^{\Sigma}\mathrm{LTS}(H') \subseteq \mathsf{lan}\ ^{\Sigma}\mathrm{LTS}(H)$

$\Rightarrow$ {Set Theory}

$\mathsf{lan}\ ^{\Sigma}\mathrm{LTS}(H'') \subseteq \mathsf{lan}\ ^{\Sigma}\mathrm{LTS}(H)$

$\Rightarrow$ {Lemma 5.2}

$^{\Sigma}\mathrm{LTS}(H'') \sqsubseteq_{\mathsf{co}} {}^{\Sigma}\mathrm{LTS}(H)$

Part iii -

Directly follows from Part (ii) and the fact that $\underset{\sim}{\sqsubset}$ is stronger than $\underset{\sim}{\sqsubset}_{MUST}$ by Def. 4.3.

Part iv -

$^{\Sigma}\mathrm{LTS}(H') \sqsubseteq_{\mathsf{co}} {}^{\Sigma}\mathrm{LTS}(H) \wedge {}^{\Sigma}\mathrm{LTS}(H) \underset{\sim}{\sqsubset}_{MAY} {}^{\Sigma}\mathrm{LTS}(H'')$

$\Rightarrow$ {Lemma 5.3, $E' \subseteq E$}

$\mathsf{lan}\ ^{\Sigma}\mathrm{LTS}(H') \subseteq \mathsf{lan}\ ^{\Sigma}\mathrm{LTS}(H) \wedge {}^{\Sigma}\mathrm{LTS}(H) \underset{\sim}{\sqsubset}_{MAY} {}^{\Sigma}\mathrm{LTS}(H'')$

$\Rightarrow$ {Theorem 4.1 (a)}

$\mathsf{lan}\ ^{\Sigma}\mathrm{LTS}(H') \subseteq \mathsf{lan}\ ^{\Sigma}\mathrm{LTS}(H) \wedge \mathsf{lan}\ ^{\Sigma}\mathrm{LTS}(H) \subseteq \mathsf{lan}\ ^{\Sigma}\mathrm{LTS}(H'')$

$\Rightarrow$ {Set Theory}

$\mathsf{lan}\ ^{\Sigma}\mathrm{LTS}(H') \subseteq \mathsf{lan}\ ^{\Sigma}\mathrm{LTS}(H'')$

$\Rightarrow$ {Lemma 5.2}

$^{\Sigma}\mathrm{LTS}(H') \sqsubseteq_{\mathsf{co}} {}^{\Sigma}\mathrm{LTS}(H'')$

Part v -

$^{\Sigma}\mathrm{LTS}(H') \sqsubseteq_{\mathsf{co}} {}^{\Sigma}\mathrm{LTS}(H) \wedge {}^{\Sigma}\mathrm{LTS}(H'') \underset{\sim}{\sqsubset}_{MUST} {}^{\Sigma}\mathrm{LTS}(H)$

$\Rightarrow$ {Lemma 5.3; $E' \subseteq E$}

$\mathsf{lan}\ ^{\Sigma}\mathrm{LTS}(H') \subseteq \mathsf{lan}\ ^{\Sigma}\mathrm{LTS}(H) \wedge {}^{\Sigma}\mathrm{LTS}(H'') \underset{\sim}{\sqsubset}_{MUST} {}^{\Sigma}\mathrm{LTS}(H)$

$\Rightarrow$ {Corollary B.1}

$\mathsf{lan}\ ^{\Sigma}\mathrm{LTS}(H') \subseteq \mathsf{lan}\ ^{\Sigma}\mathrm{LTS}(H) \wedge \mathsf{lan}\ ^{\Sigma}\mathrm{LTS}(H) \subseteq \mathsf{lan}\ ^{\Sigma}\mathrm{LTS}(H'')$

$\Rightarrow$ {Set Theory}

$\mathsf{Ian}\ ^{\Sigma}\mathrm{LTS}(H') \subseteq \mathsf{Ian}\ ^{\Sigma}\mathrm{LTS}(H'')$

$\Rightarrow$ {Lemma 5.2}

$^{\Sigma}\mathrm{LTS}(H') \sqsubseteq_{\mathsf{co}} {}^{\Sigma}\mathrm{LTS}(H'')$

Part vi -

Directly follows from Part (v) and the fact that $\overset{\sqsubseteq}{\sim}$ is stronger than $\overset{\sqsubseteq}{\sim}_{MUST}$ by Def. 4.3. □

Proof of Proposition 6.1

$^{\Sigma}\mathrm{LTS}(H) \overset{\sqsubseteq}{\sim}_{MAY} {}^{\Sigma}\mathrm{LTS}(H')$

$\Leftrightarrow$ {Theorem 4.1}

$^{\Sigma}\mathrm{LTS}(H) <<_{MAY} {}^{\Sigma}\mathrm{LTS}(H')$

$\Leftrightarrow$ {Def. 4.6}

$\mathsf{Ian}\ ^{\Sigma}\mathrm{LTS}(H) \subseteq \mathsf{Ian}\ ^{\Sigma}\mathrm{LTS}(H')$

$\Rightarrow$ {$\mathsf{Ian}\ \mathrm{LTS}(H) \subseteq \mathsf{Ian}\ ^{\Sigma}\mathrm{LTS}(H)$ by Lemma 6.1 (iii)}

$\mathsf{Ian}\ \mathrm{LTS}(H) \subseteq \mathsf{Ian}\ ^{\Sigma}\mathrm{LTS}(H')$

$\Rightarrow$ {$(\mathsf{Ian}\ \mathrm{LTS}(H)) \setminus \Sigma = \mathsf{Ian}\ \mathrm{LTS}(H)$; Lemma 6.1 (ii)}

$\mathsf{Ian}\ \mathrm{LTS}(H) \subseteq \mathsf{Ian}\ \mathrm{LTS}(H')$

$\Leftrightarrow$ {Def. 4.6}

$\mathrm{LTS}(H) <<_{MAY} \mathrm{LTS}(H')$

$\Leftrightarrow$ {Theorem 4.1}

$\mathrm{LTS}(H) \overset{\sqsubseteq}{\sim}_{MAY} \mathrm{LTS}(H')$ □

Proof of Proposition 6.2

$^{\Sigma}\mathrm{LTS}(H) \overset{\sqsubseteq}{\sim}_{MUST} {}^{\Sigma}\mathrm{LTS}(H')$

$\Rightarrow$ {Corollary B.1}

$\mathsf{Ian}\ ^{\Sigma}\mathrm{LTS}(H') \subseteq \mathsf{Ian}\ ^{\Sigma}\mathrm{LTS}(H)$

$\Rightarrow$ {Lemma 5.2}

$^{\Sigma}\mathrm{LTS}(H') \sqsubseteq_{\mathsf{co}} {}^{\Sigma}\mathrm{LTS}(H)$

$\Rightarrow$ {Lemma 6.3[11]}

$\mathrm{LTS}(H) \overset{\sqsubseteq}{\sim}_{MUST} \mathrm{LTS}(H')$ □

Proof of Proposition 6.3

The proposition directly follows from Propositions 6.1 and 6.2. □

[11] The use of Lemma 6.3 requires $E = E'$

References

[1] A. Aho and J. Ullman. *Principles of Compiler Design.* Addison-Wesley, 1979.

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

[3] 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 *LNCS*, pages 107–121. Springer-Verlag, 1998.

[4] T. Bolognesi and M. Caneve. Squiggles - a tool for the analysis of LOTOS specifications. In K. Turner, editor, *Forte '88*, pages 201–216. North-Holland Publishing Company, 1989.

[5] 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.

[6] H. Bowman and J. Derrick. A junction between state based and behavioural based specifications. 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 213–239. Kluwer Academic Publishers, 1999. ISBN 0-7923-8429-6.

[7] 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.

[8] A. Cavarra, C. Crichton, J. Davies, A. Hartman, T. Jeron, and L. Mounier. Using UML automatic test generation. In J.P. Katoen and P. Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *LNCS*. Springer-Verlag, 2002.

[9] R. De Nicola. Extensional equivalences for transition systems. *Acta Informatica*, (24):211–237, 1987.

[10] A. Fantechi, S. Gnesi, and A. Maggiore. Enhancing test coverage by back-tracing model-checker counterexamples. In M. Pezzé, editor, *Proceedings of TACoS 2004*, volume 116 of *Electronic Notes in Theoretical Computer Science*, pages 199–211. Elsevier Science Publishers B.V., 2004.

[11] E. Farchi, A. Hartman, and Pinter S. Using a model-based test generator to test for standard conformance. *IBM SYSTEMS JOURNAL*, 41(1):89–109, 2002.

[12] 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.

[13] J. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu, and M. Sighireanu. CADP: A protocol validation and verification toolbox. In *Conference on Computer-Aided Verification*, volume 1102 of *LNCS*, pages 436–440. Springer-Verlag, 1996.

[14] 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).

[15] 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*, 51(1):43–75, 2002.

[16] S. Gnesi, D. Latella, and M. Massink. Formal Test-case Generation for UML Statecharts. In P. Bellini, S. Bohner, and B. Steffen, editors, *9th IEEE International Conference on Engineering of Complex Computer Systems*, pages 75–84. IEEE, IEEE Computer Society Press, 2004. ISBN 0-7695-2109-6.

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

[18] R. Hierons. Adaptive Testing of a Deterministic Implemenatation Against a Nondeterministic Finite State Machine. *The Computer Journal. Oxford University Press.*, 41(5):349–355, 1998.

[19] C.A.R. Hoare. *Communicating Sequential Processes.* Series in Computer Science. Prentice Hall, 1985.

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

[21] Y. Kim, H. Hong, D. Bae, and S. Cha. Test cases generation from UML state diagrams. *IEE Proc. Softw. IEE*, 146(4):187–192, 1999.

[22] R. Langerak. *Transformations and Semantics for LOTOS.* PhD thesis, University of Twente, 1992.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] D. Latella, M. Massink, H. Baumeister, and M. Wirsing. Mobile UML Statecharts with Localities. In C. Priami and P. Quaglia, editors, *Global Computing 2004*, volume 3267 of *LNCS*, pages 34–58. Springer-Verlag, 2005.

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

[29] N. Lynch and M. Tuttle. An Introduction to Input/Output Automata. *CWI Quarterly*, 2(3):219–246, 1989.

[30] Z. Manna, S. Ness, and J. Vuillemin. Inductive methods for proving properties of programs. *Communications of the ACM. ACM Press*, 16(8):491–502, 1973.

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

[32] E. Mikk, Y. Lakhnech, and M. Siegel. Hierarchical automata as model for statecharts. In R. Shyamasundar and K. Euda, editors, *Third Asian Computing Science Conference. Advances in Computing Sience - ASIAN'97*, volume 1345 of *LNCS*, pages 181–196. Springer-Verlag, 1997.

[33] R. Milner. *Communication and Concurrency.* Series in Computer Science. Prentice Hall, 1989.

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

[35] A. Petrenko, N. Yevtushenko, and G von Bochmann. Testing deterministic implementations from nondeterministic FSM specifications. In B. Baumgarten, H. Burkhardt, and A. Giessler, editors, *Testing of Communication Systems.*, pages 125–140. Chapman & Hall, 1996.

[36] S. Pickin, C. Jard, Y. Le Traon, T. Jeron, J. Jezequel, 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 *LNCS*. Springer-Verlag, 2002.

[37] 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.

[38] Chow T. Testing Software Design Modeled by Finite-State Machines. *IEEE Transactions on Software Engineering. IEEE CS*, SE-4(3):178–187, 1978.

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

[40] J. Tretmans. *A Formal Approach to Conformance Testing.* PhD thesis, University of Twente, 1992.

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

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