Modular semantics for a UML statechart diagrams kernel and its extension to multicharts and branching time model-checking

Stefania Gnesi a,∗, Diego Latella b, Mieke Massink b

a Consiglio Nazionale delle Ricerche, Istituto di Elaborazione dell’Informazione, Via Moruzzi 1, Pisa, Italy
b Consiglio Nazionale delle Ricerche, Istituto CNUCE, Via Alfieri 1, S. Cataldo, Pisa, Italy

Received 5 February 2001; received in revised form 2 July 2001; accepted 3 July 2001

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 operational semantics for a behavioural subset of UML statechart diagrams (UMLSDs) including a formal proof of their correctness with respect to major UML semantics requirements concerning behavioural issues. We show how the modularity of our semantics definition can be exploited to define extensions, in particular we show an extension to models composed of collections of communicating statechart diagrams, which we call multicharts. Finally we present all the conceptual issues related to building a tool for action based branching time model-checking, for the automatic verification of formal correctness of UML multicharts. The approach we propose preserves all the information necessary to report the results of model-checking in terms of the original UMLSD specification. The reference verification environment used for this model-checking approach is JACK, where automata are represented in a standard format which facilitates the use of a collection of tools for automatic verification. © 2002 Elsevier Science Inc. All rights reserved.

Keywords: Formal specification; Formal semantics; Formal verification; Model-checking; UML; ACTL; JACK

1. Introduction and related work

The unified modelling language (UML) is a graphical modelling language for object-oriented software and systems [35]. It has been specifically designed for visualizing, specifying, constructing and documenting several aspects of – or views on – systems. Different diagrams are used for the description of the different views. UML is a semi-formal
language in the sense that its syntax and static semantics are defined formally, but its dynamic semantics is specified only informally. This poses difficulties for the development of sophisticated tools that go beyond syntax and static semantics checks of specifications such as tools for the formal analysis of dynamic behaviour.

In this paper, we concentrate on the analysis of dynamic behaviour of UML statechart diagrams (UMLSDs) based on model-checking techniques. These techniques allow the behaviour of a system to be analysed with respect to some correctness properties expressed by temporal logic formulas.

Our long term goal is to develop a high quality integrated environment for UML which supports formal verification via model-checking, formal test derivation, and integrated quantitative analysis including discrete event simulation.

Given the increasingly important role the UML is playing in the development of computer systems, including safety critical ones, it is important that high quality assurance standards are applied also for the design and implementation of support tools, in particular validation tools. This increases the confidence one can put on the verification and validation results that the tools produce. In this paper, we focus more on the design issues of such tools than on implementation ones. The methodology we follow to achieve the above goal is based on the use of (a) formal definition of the syntax and semantics of the particular notational subset of interest as well as (b) rigorous and, whenever feasible and/or convenient, formal proofs concerning features of the notation and/or of tool design or implementation. The formal definition of a notation provides an unambiguous description of the notation, which forms a fundamental step in the design of support tools and often can be used directly in their abstract specification. The rigorous proof of features of the notation greatly contributes in better understanding the notation itself and its conceptual implications. Furthermore, should the criticality of target applications, or the complexity of the support tools require formal correctness proofs of the design or implementation of such tools, the formal definition of the notation will anyway be a necessary prerequisite. An example of a UMLSDs to PROMELA [23] compiler derived from the formal semantics of UMLSDs we discuss in the present paper is described in [25] where a proof of correctness is also given. The state/transitions enumeration algorithm we present in this paper, which defines the main additional component necessary for model-checking UMLSDs via an existing model-checker, uses directly the transition relation defined in the formal semantics. We do not address its correctness explicitly here since the algorithm is straightforward.

In conclusion, a central role is played by the definition of formal semantics for the notation. On the other hand, putting formal semantics at the centre of language and tools development should not result in a source of rigidity. Thus, the formal semantics should be as flexible as possible so that new extensions can be added in a modular and/or orthogonal way and central parts of the semantics definition can be reused without too much effort.

1.1. Behavioural semantics for UMLSDs

In this paper, we focus on UMLSDs, which are meant for describing dynamic aspects of system behaviour. Following the above mentioned methodology the first step is the definition of a formal semantics for a kernel of UMLSDs, which we presented in [27]. We call such a kernel a “behavioural subset” of UMLSDs, since it includes all the interesting conceptual issues related to concurrency in the dynamic behaviour, like sequentialization, non-determinism and parallelism. The proposed semantics associates a labelled transition
system to each UMLSD, thus making classical model-checking algorithms applicable. Besides the above mentioned behavioural concepts, our semantics covers UMLSDs specific transition priorities, state refinement and inter-level transitions. More specifically, for dealing with priorities, we use a notion of “priority schema”, on which the semantics definition is “parametric”. Then we instantiate them on the UML specific one. Different instantiations bring to different priority schemas, including the original classical Statecharts one. Moreover, since the UML definition of the external environment is only partially defined, our semantics definition is parametric w.r.t. the environment as well.

The operational semantics presented in [27], for which in Section 3 we show correctness with respect to major behavioural requirements, assumed a system modelled by a single UMLSD. In this paper, we extend that semantics to the case of $N$ UMLSDs communicating via input-queues reusing the core semantics for a single UMLSD. We call such collections of UMLSDs *UML multicharts*. The extension to multicharts is in line with the official definition of the UML, although we are not completely convinced of the methodological soundness of modelling a given system with more than a single UMLSD – a discussion on this issue can be found in [25]. Nevertheless the high modularity of our semantics definition and of our approach makes it extremely easy for us to upgrade our semantics to the case of more than one single UMLSD, so we think the extension is worth, given that UML models are very likely to use multiple UMLSDs.

In our current work, we do not consider some important aspects of UMLSDs such as different types of states, events, actions and transitions. Most of them do not pose any conceptual difficulties in relation to the behavioural aspects of the semantics. Including them at this stage would however obfuscate the behavioural aspects of the notation and render the presentation of the semantics unnecessarily complicated. In Section 2, a precise description is given of the aspects of UMLSDs that are covered in this paper. Other limitations, namely those related to the object-oriented features of UML are more serious and require further study. Nevertheless, a basic formal semantics model and related tools, even for a restricted language, is an essential step for any further extension with the above mentioned features. The definition of a sound “basic” kernel of a notation has already proven a valuable and fruitful methodology and is now quite standard practice in many fields of concurrency theory, like process-algebra. Novel semantics concepts, like e.g. those related to stochastic behaviour [18], can be experimented on such a kernel and new mathematical theories, like e.g. those for behavioural preorders and equivalences [28], can be developed on it. Experimental tools can be developed on the basis of – or even derived from – such theories and concepts. Once enough experience has been gained with the kernel, its extension to other important features of the notation can be supported by a safer conceptual, theoretical and experimental background. Thus, in our opinion, a sound formal semantics for UMLSDs is a necessary condition for extending the considered notation with the inclusion of object-oriented features like classes and subclasses. The formal semantics serves as a necessary starting point for the definition of behavioural (ordering) relations which can play a role in the definition of the notion of sub-behaviour, connected to the notion of sub-classes [3]. A first attempt towards this direction is suggested in [28].

Several other approaches have been proposed in the literature for the definition of formal semantics of UMLSDs, e.g. [4,29,30,38], and much more has been done for classical statecharts. To the best of our knowledge, both Refs. [38] and [4] do not deal with transition priorities and moreover state refinement is not considered in [38].

The approach we followed for our definition of the semantics is similar to that proposed in [32] for classical statecharts but it takes into consideration the peculiarities of the UM-
LSDs relevant for the considered subset of the notation. On the other hand, it shares the relative simplicity of the work proposed in [32].

In [29,30] all interesting aspects of UMLSDs semantics are covered, but no correctness result for the proposed semantics is provided. More emphasis is put on implementation related issues as the work constitutes a basis for a PROMELA/SPIN based model-checker for UMLSDs. In [29,30] a “flat” representation of UMLSDs is used and the authors claim that such a representation is better suited for model-checking purposes than the hierarchical one used in [27]. We do not share such an opinion: using a hierarchical representation for UMLSDs (abstract syntax) has no negative impact on tools development. Rather, it helps considerably in carrying out correctness proofs; all interesting results in our work are proven inductively and such proofs heavily exploit the hierarchical structure of our representation, which is also the basis of the structure of our semantics deduction system – an example of such proofs will be provided later in this paper. Moreover, the same arguments hold for the specification and implementation of support tools since it is well known that hierarchical definitions greatly help in developing modular implementations – and addressing/proving their correctness.

1.2. Action based branching time model-checking

The benefits of branching time logics have been widely recognized in the literature and it is also well known that linear time and branching time logics are incomparable as far as their expressive power is concerned, i.e., there are interesting properties of system behaviour which can be expressed in linear time logics but cannot be expressed in branching time logics, and vice versa [5].

In this paper, we discuss a formal approach to model-checking UMLSDs based on the JACK verification environment [2]. JACK includes AMC, a model-checker for ACTL [13], a branching time temporal logic suitable to express properties of reactive systems whose behaviour is characterized by the actions they perform and whose semantics is defined by means of labelled transition systems.

The possibility of reasoning about system behaviour using directly the observable actions its components may perform, instead of their internal states has been proven of considerable help in several concurrency frameworks like e.g. process algebras and may provide a convenient way to express action based properties also for automata that traditionally have been associated with state-based temporal logics [1,12,15]. We think it will turn out useful also in the context of UMLSDs, since the actions (actually the events, in UML terminology) labelling statechart transitions obviously play a major conceptual role in modelling systems by means of this notation.

We are not aware of other proposals of (action) branching time model-checking for UMLSDs. There is more work in the context of Linear Time, state based, model-checking for statecharts. Linear time model-checking of classical statecharts is addressed in [9,33]. In [9] logical properties are expressed in a graphical language which is then translated into a linear time logic. In [33] classical statecharts are translated into PROMELA, the modelling language of the SPIN model-checker. As mentioned before, linear-time model-checking of UMLSDs is addressed in [25,26,29,30]. In [25,26] a translation to PROMELA is given based on the semantics proposed in [27] and is proven correct with respect to such semantics.

A preliminary version of the work on branching time action based model checking for UMLSDs presented in this paper has been published as Ref. [17].
1.3. Organization of the paper

The present paper is organized as follows: in Section 2 the subset of UMLSDs which we consider in this paper is introduced informally, together with its translation into the intermediate representation of hierarchical automata. Hierarchical automata and their operational UML-semantics are defined in Section 3, where the correctness theorem of the semantics is also given; the operational semantics is then extended to collections of UMLSDs. Section 4 addresses branching time action based model-checking of UMLSDs using the JACK environment. Some conclusions are drawn in Section 5. The detailed proof of the correctness theorem is given in Appendix A.

2. Basic concepts

UMLSDs are a (object-oriented) variant of classical Harel statecharts [20,21]. The statecharts formalism itself is an extension of traditional state transition diagrams. In this section, we briefly describe those features of UMLSDs which are of interest for this paper. We describe them by means of the example of Fig. 1, a variant of the example used in [32,33].

One of the main notions of statecharts is that of state refinement. In Fig. 1 state S of a TV system TV_SYS is refined into two automata. The left-hand one, TV, is composed of two states, ON and OFF, while the right-hand one, the USER, consists of just one state, representing a “chaotic” user. State OFF is again refined into an automaton with two states. State ON is a composite state and in particular it is said to be an AND-state or concurrent. Symmetrically, state POWER is an OR-state, as well as IMAGE and SOUND.

A transition connects a source to a target state. The transition is labelled by a trigger event, a Boolean guard and a sequence of actions. In our example, only trigger/action pairs are used, where the action consists in generating one (output) event. When a transition does not generate any output, the label consists only of the trigger. Finally, when a transition does not need any particular event for being triggered, the notation -action is used.

“System states” are modelled by configurations, which are sets of states. For instance, the following are configurations of our sample system: \{S, U, OFF, DIS\}, \{S, U, OFF, STB\}, \{S, U, ON, SHW, SND\}.

A transition is enabled and can fire if and only if its source state is in the current configuration, its trigger is offered by the external environment and the guard is satisfied. In this case, if the transition fires, the source state is left, the actions are executed, and the target state is entered.

![Fig. 1. Example of a UMLSD.](image-url)
In our example, if event on is given as input to the machine and the current configuration is \{S, OFF, STB, U\}, state OFF is left (together with state STB), the transition labelled by on is fired together with a transition of the user, and state ON is entered. In particular, ON being composite, we also have to say which are the particular sub-states which are reached. In the case at hand they are the default ones, i.e., the initial states of IMAGE and SOUND, namely, SHW and SND. Depending on which transition of the user is (non-deterministically) taken, a different event will be delivered to the environment; for instance if the lower-most transition is taken, then event txt will be delivered.

In the general case, some target sub-states can be explicitly specified. In our example, when the current configuration contains ON and event off is offered, the configuration resulting from firing the related transition will be \{S, OFF, STB, U\}, where STB is explicitly pointed to by the transition. Such a transition is called an inter-level transition and can in general have more than one target in order to explicitly point to all relevant states (fork transitions).

Symmetrically, also the transition from STB to ON is an inter-level one. Firing it requires the system to be in a configuration containing STB. Inter-level transitions can also have more than one source state, the meaning being that all such states must be in the current configuration for the transition to be enabled (join transitions). Compound transitions can be either join or fork transitions.

In general, more than one event can be available in the environment. The UML semantics assumes a dispatcher which selects one event at a time from the environment, modelled as a queue, and offers it to the state machine. More than one transition can be enabled at this point. Some of them can be in conflict: this happens when the intersection of the sets of states left by the transitions is not empty. Some conflicts can be resolved using priorities. Roughly speaking a transition has higher priority than another transition if its source state is a sub-state of the source of the other one. If the conflict cannot be resolved using priorities, then any of the conflicting enabled transitions may be fired. Due to concurrent states, it is possible that more than a single transition is fired as a reaction to a given event, as is the case for all transitions at top level, in our example.

In particular, according to the semantics of UML [35] the set of transitions that will fire must fulfill all of the following requirements:

- it must be maximal with respect to set inclusion,
- it must contain only transitions which are currently enabled,
- it must contain only transitions which are not mutually conflicting, and
- it must be such that no enabled transition outside the set has higher priority than a transition in the set.

When the effects of all such transitions and related actions are complete a new event is selected by the dispatcher and a new cycle is started.

The general idea within the UML community is to model each object behaviour with a different statechart diagram. Thus the overall model of behaviour, at a system level, will be specified as a collection of statechart diagrams each associated with its state machine equipped with its input-queue. We call such collections multicharts. In this extended context, at the beginning of each cycle the dispatcher is also in charge of choosing the particular input-queue from which the event to be processed is to be selected and to feed it to the associated machine. The nature of such a choice is left unspecified by our semantics, i.e., it is left completely non-deterministic.
Symmetrically, once an event is generated as an output action, it will be copied to the input-queue of each and every machine to which it has been addressed. This implies the existence of a proper naming/addressing schema in the context of Multicharts, possibly in conjunction with object methods names. In this paper, we do not discuss any detail of such schemas. The extended semantics will be explicitly addressed in Section 3.2.

As stated in Section 1, the semantic model we use in this paper is a slight variant of the one defined in [27]. This model is defined for a quite restricted subset of UMLSDs, which, nevertheless includes all the interesting conceptual issues related to concurrency in the dynamic behaviour, like sequentialization, non-determinism and parallelism. In this paper, we shall refer to the same subset of the notation as in [27]. More specifically, we do not consider history, action and activity states; we restrict events to signal ones 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 as are branch transitions; also variables and data are not allowed so that actions are required to be just (a sequence of) events. We also abstract from entry and exit actions of states.

The first step of our approach is a purely syntactical one and consists in translating UMLSDs into what is usually called a hierarchical automaton (HA). HAs can be seen as an abstract syntax for UMLSDs 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.

In the sequel we will be concerned only with HAs since the translation from UMLSDs to HAs is conceptually simple and purely syntactical [27].

3. Hierarchical automata

In this section, we recall the notion of HAs as defined in [27,32] and their UML operational semantics given in [27]. Only the relevant definitions of HAs and their UML operational semantics are given here. We refer to [27] for details. Notice that our formal development has been based on the UML definition given in [36,37]. The main concepts, notions and results we discuss in the present paper remain valid also for the next version of UML, defined in [35]. The first notion we need to define is that of (sequential) automaton.1

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

The labels in $\lambda_A$ have a particular structure which we shall discuss later. Moreover, we assume that all transition labels are unique. This can be easily achieved by just assigning them arbitrary unique names, as we shall do in the rest of this paper. For sequential

1 In the following we shall freely use a functional-like notation in our definitions where: (i) currying will be used in function application, i.e., $f a_1 a_2 \cdots a_n$ will be used instead of $f(a_1, a_2, \ldots, a_n)$ and function application will be considered left-associative; (ii) for function $f : X \longrightarrow Y$ and $Z \subseteq X$, $f[Z] = \{y \in Y | \exists x \in Z. y = f x\}$, $\text{rng} f$ denotes the range of $f$ and $f|Z$ is the restriction of $f$ to $Z$; (iii) by $\exists_1 x$. $P x$ we mean “there exists a unique $x$ such that $P x$".
automaton $A$ let functions $\text{SRC}, \text{TGT} : \delta_A \rightarrow \sigma_A$ be defined as $\text{SRC}(s, l, s') = s$ and $\text{TGT}(s, l, s') = s'$. HAs are defined as follows:

**Definition 2 (Hierarchical automata).** An HA $H$ is a 3-tuple $(F, E, \rho)$, 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 $\rho : \bigcup_{A \in F} \sigma_A \rightarrow 2^F$ imposes a tree structure to $F$, i.e., (i) there exists a unique root automaton $A_{\text{root}} \in F$ such that $A_{\text{root}} \not\in \bigcup \text{rng } \rho$, (ii) every non-root automaton has exactly one ancestor state: $\bigcup \text{rng } \rho = F \setminus \{A_{\text{root}}\}$ and $\forall A \in F \setminus \{A_{\text{root}}\}. \exists_{1s} s' \in \bigcup_{A' \in F \setminus \{A\}} \sigma_{A'}$. $A \in (\rho s)$, 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 label $l$ of a transition $t = (s, l, s')$ of (sequential) automaton $A$ is a tuple $(\text{EV}_t, \text{SR}_t, \text{AC}_t, \text{G}_t, \text{TD}_t)$. The meaning of the components of the labels is intimately connected with the use of HAs as an abstract syntax for UMLSDs. It is thus convenient to explain such a meaning by means of an example which shows how our sample UMLSD is mapped into an HA.

The HA representing our sample UMLSD is shown in Fig. 2. The complete information related to its transition labels is given in Table 1 where $u_1, \ldots, u_7$ are their unique names. It is easy to see that there is a clear correspondence between the states of the two structures. Also the refinement of a state into one or more sub-states in the statechart is properly represented by the refinement function $\rho$; in our example we have $\rho(\text{OFF}) = \{\text{POWER}\}$, $\rho(\text{ON}) = \{\text{IMAGE}, \text{SOUND}\}$ and $\rho(s) = \emptyset$ for any other state $s$. In the figure, this is represented by dotted arrows. Initial states are indicated by thick boxes.

Non-inter-level transitions are represented in the obvious way. Consider now the inter-level transition from STB to ON in Fig. 1. Such a transition is represented in the HA by the transition from OFF (the highest ancestor of STB “crossed” by the transition in the statechart) to ON, labelled by $\text{on}$. The indication of the fact that the real “origin” of such a transition is state STB is encoded in the label of the transition (see Table 1). In particular, it is encoded in what is called the source restriction (SR) of the transition. The source restriction of transition $\text{on}$ is STB. In general, for join transitions the source restriction is a set of states. The label also contains the event (EV) which triggers the transition and the corresponding actions (AC) to be performed when the transition is fired. Finally, the label

![Fig. 2. Example of a hierarchical automaton.](image-url)
| t   | off1 | off2 | on  | d  | st | v   | sh | m  | sn | u₀ | u₁ | u₂ | u₃ | u₄ | u₅ | u₆ | u₇ |
|-----|------|------|-----|----|----|-----|----|----|----|----|----|----|----|----|----|----|
| EV t| off  | out | on  |    |    | txt |    |    |    |    |    |    |    |    |    |    |
| SR t| ∅    | ∅   |      |    |    |    |    |    |    |    |    |    |    |    |    |    |
| TD t| {STB}| {DIS}| {SHW, SND} |    |    |    |    |    |    |    |    |    |    |    |    |    |
| AC t| ϵ    | ϵ   | ϵ   |    |    |    |    |    |    |    |    |    |    |    |    |    |
of a transition contains the so called target determinator (TD) and the optional guard (G). The target determinator explicitly lists all the basic (i.e., non refined) states which must be reached when a transition is fired. For example, the transition from ON to STB in Fig. 1 is represented in Fig. 2 by the transition labelled by off, the target determinator of which is \{STB\}. Similarly, the TD of the transition labelled by on is \{SHW, SND\}. The guard is a Boolean expression expressing a condition which must evaluate to true for the transition to be enabled to fire (in our example we do not deal with guards for simplicity; we will just assume all of them be the constant true).

An algorithm for translating UMLSDs to HAs can be found in [27]. In the sequel we shall implicitly make reference to a generic HA $H = (F, E, \rho)$. Every sequential automaton $A \in F$ characterizes an HA in its turn: intuitively, such an HA is composed by all those sequential automata which lay below $A$, including $A$ itself, and has a refinement function $\rho_A$ which is a proper restriction of $\rho$.

**Definition 3.** For $A \in F$ the automata, states and transitions under $A$ are defined, respectively, as

\[
\mathcal{A} A = \{A\} \cup (\bigcup_{A' \in (\bigcup_{s \in \sigma(A)}(\mathcal{A} A'))}(\mathcal{A} A')),
\mathcal{S} A = \bigcup_{A' \in \mathcal{A} A} \sigma A', \quad \text{and}
\mathcal{T} A = \bigcup_{A' \in \mathcal{A} A} \delta A'
\]

The definition of sub-hierarchical automaton follows:

**Definition 4 (Sub-hierarchical automata).** For $A \in F$, $(F A, E, \rho A)$, where $F A = (\mathcal{A} A)$, and $\rho A = \rho|_{(\mathcal{S} A)}$, is the HA characterized by $A$.

In the sequel for $A \in F$ we shall refer to $A$ both as a sequential automaton and as the sub-hierarchical automaton of $H$ it characterizes, the role being clear from the context. $H$ will be identified with $A_{\text{root}}$. Sequential Automata will be considered a degenerate case of HAs. The notions of conflicting transitions, transition priority and orthogonal states are defined below. For a deeper discussion on their properties the reader is referred to [24]. Both the notion of conflict and that of priority are based on the notion of state precedence:

**Definition 5 (State precedence).** For $s, s' \in \mathcal{S} H$, $s \prec s'$ iff $s' \in \mathcal{S} (\rho s)$. Let also $\preceq$ denote the reflexive closure of $\prec$.

The following definition establishes when two transitions are conflicting:

**Definition 6 (Conflicting transitions).** For $t, t' \in (\mathcal{T} H)$, $t$ is conflicting with $t'$, written $t \# t'$, iff $t \neq t'$ and $(\text{SRC } t \preceq \text{SRC } t') \lor (\text{SRC } t' \preceq \text{SRC } t)$.

The following definition characterizes those structures which can be used for imposing priorities on transitions.

**Definition 7 (Priority schema).** A Priority Schema is a triple $(\Pi, \sqsubseteq, \pi)$ with $(\Pi, \sqsubseteq)$ a partial order and $\pi : (\mathcal{T} H) \rightarrow \Pi$ such that: $\forall t, t' \in (\mathcal{T} H). (\pi t \sqsubseteq \pi t') \land t \neq t' \Rightarrow t \# t'$.

We say that $t$ has lower priority than (the same priority as) $t'$ iff $\pi t \sqsubseteq \pi t'$. 
A possible choice for a priority schema is given below. It is based on state precedence and generalizes the requirement that transitions originating from “inner” states have priority over those higher in the state hierarchy [35,37]. To that purpose we first need the notion of orthogonal states.

**Definition 8 (Orthogonal states).** Two states \( s, s' \in \mathcal{S} \) are orthogonal, written \( s \parallel s' \), iff
\[
\exists s'' \in (\mathcal{S}), A, A' \in (\rho s''), A \neq A' \land s \in \mathcal{A} A \land s' \in \mathcal{A} A'.
\]

The following lemma relates orthogonality with conflict and priority and can be proved directly for the relevant definitions:

**Lemma 1.** For \( t, t' \in (\mathcal{T}) \) the following holds: \((\text{SRC } t) \parallel (\text{SRC } t')\) implies \( \neg (t \# t') \) and \( \pi t \nsubseteq \pi t' \).

Let \((\text{PWO, } \preceq^s, f)\) such that:
- \(\text{PWO} = \{ S \subseteq (\mathcal{S}) \mid \forall s, s' \in S. (s \neq s' \Rightarrow s \parallel s')\}\).
- For all \( S, S' \subseteq \mathcal{S} \), \( S \preceq^s S' \) iff \( \forall s \in S. \exists s' \in S'. s \preceq s' \).
- \( f t = [s \mid s \in (\text{SRC } t) \land (\text{SR } t) = \emptyset] \cup (\text{SR } t) \).

It can be easily shown that \((\text{PWO, } \preceq^s, f)\) is a priority schema [24].

In the remaining of this section we shall deal with the semantics of HAs. A configuration denotes a global state of an HA, composed of local states of component sequential automata:

**Definition 9 (Configurations).** A configuration of \( H \) is a set \( C \subseteq (\mathcal{S}) \) such that (i) \( \exists \sigma s' \in \sigma_{A \text{root}}. s \in C \) and (ii) \( \forall s, A. s \in C \land A \in \rho s \Rightarrow \exists \sigma s' \in A. s' \in C \).

For \( A \in F \) the set of all configurations of \( A \) is denoted by \( \text{Conf}_A \). The operational semantics of an HA is defined as a labelled transition system (LTS), which is a set of states related by a transition relation.

**Definition 10 (Labelled transition system).** An LTS is a 4-tuple \( Ts = (S, s^0, \text{Act}, \rightarrow) \) where \( S \) is a set of states, \( s^0 \) is the initial status, \( \text{Act} \) is a finite and non-empty set of actions, \( \rightarrow \subseteq S \times \text{Act} \times S \) is the state transition relation.

In the context of UMLSDs, states are called statuses and the transition relation is called the STEP relation. We shall use states and statuses as synonymous, when this will not cause confusion. The (STEP) transitions are labelled by the set of the labels of those transitions of the sequential automata which have been fired in the HA. Each status is composed of a configuration and the current environment with which the HA is supposed to interact. While in classical statecharts the environment is modelled by a set, in the definition of UMLSDs the particular nature of the environment is not specified (actually it is stated to be a queue, but the management policy of such a queue is not defined). We choose not to fix particular semantics such as sets, or bags or FIFO queues, etc., but to model the environment in a policy-independent way. In the following definition, we assume that for set \( X, \Theta X \) denotes the set of all structures of a certain kind (like FIFO queues, or bags, or sets) over \( X \) and we assume to have basic operations for inserting and removing elements from such structures. In particular \((\text{add } \& e)\) denotes the structure obtained by adding \( e \) to environment \( \& \). Similarly, \((\text{join } \& \&')\) denotes the environment obtained by merging \( \& \).
with $\mathcal{E}'$. The predicate $\text{is}\_\text{join}^n_{j=1} \mathcal{E}_j \mathcal{J}$ states that $\mathcal{J}$ is a possible join of $\mathcal{E}_1 \cdots \mathcal{E}_n$ and it is a way for expressing non-deterministic merge of $\mathcal{E}_1 \cdots \mathcal{E}_n$. Moreover, by $(\text{Sel} \ \mathcal{E} \ e \ \mathcal{E}')$ we mean that $\mathcal{E}'$ is the environment resulting from selecting $e$ from $\mathcal{E}$, the selection policy depending on the choice for the particular semantics of the environment. Finally, $\text{nil}$ is the empty structure and given sequence $r \in \mathcal{X}^*$, $(\text{new} \ r)$ is the structure containing the elements of $r$ (again, the existence and nature of any relation among the elements of $(\text{new} \ r)$ depends on the semantics of the particular structure). So, for instance, if sets are chosen, then $(\text{add} \ \mathcal{E} \ e) = \mathcal{E} \cup \{e\}$, $(\text{join} \ \mathcal{E} \ \mathcal{E}') = \mathcal{E} \cup \mathcal{E}'$ and, for $e \in \mathcal{E}$, $(\text{Sel} \ \mathcal{E} \ e \ \mathcal{E}') \equiv (\mathcal{E}' = \mathcal{E}\setminus\{e\})$.

The semantics of UML does not specify what happens when a queue is empty. Our approach in this paper is to make automata stutter in such a situation, except when there are transitions which do not need any trigger for firing. To that purpose, we require both $(\text{Sel} \ \text{nil} \rightarrow \text{nil}) \equiv \text{true}$ and $- \in \mathcal{E}$ for all $\mathcal{E}$ (including the empty environment) to hold.

3.1. Operational semantics: single UMLSD

The following definition characterizes the operational semantics of an HA as an LTS for the case that we deal with a single statechart diagram. The definition is followed by several lemmas and a main theorem that state the correctness of the formal operational semantics defined here and the behavioural requirements informally defined in [35,37] and recalled in Section 2.

**Definition 11** (Operational semantics of HAs). The operational semantics of an HA $H$ is the LTS $T_s = (S, s^0, Act, \rightarrow)$ where (i) $S = \text{Conf}_H \times (\Theta \ E)$ is the set of statuses of $T_s$, (ii) $s^0 = (C_0, E_0) \in S$ is the initial status, (iii) $Act \subseteq 2^T H$ is the set of actions of $T_s$ and (iv) $\rightarrow \subseteq S \times Act \times S$ is the transition relation defined in the sequel.

A transition of $T_s$ denotes a maximal set of non-conflicting transitions of the sequential automata of $H$ which respect priorities. The $\rightarrow$ relation is defined by means of a deduction system. In this paper, we consider only closed systems, where the environment can interact only with the HA and no external manipulation is allowed on it [27]. The rule follows:

**Definition 12** (Transition relation).

$$
\frac{(\text{Sel} \ \mathcal{E} \ e \ \mathcal{E}'')}{H \uparrow \emptyset :: (\mathcal{E}, \{e\}) \xrightarrow{L} (\mathcal{E}', \mathcal{E}')} \quad (\mathcal{E}, \mathcal{E}) \xrightarrow{L} (\mathcal{E}', (\text{join} \ \mathcal{E}'', \mathcal{E}'))
$$

The above rule formally stipulates that (1) from the fact that event $e$ is a possible selection for the input event from the current environment $\mathcal{E}$ (first premiss) and (2) the fact the HA $H$, on the current configuration $\mathcal{E}$ and on such input event $e$ fires all and only the transitions in set $L -$ moving to configuration $\mathcal{E}'$ and producing events in $\mathcal{E}'$ (second premiss, see below) – a STEP transition from the current status $(\mathcal{E}, \mathcal{E})$, firing the set $L$ of transitions, can be deduced. The next configuration will obviously be $\mathcal{E}'$ while the next environment will be given by inserting the newly generated events $\mathcal{E}'$ into the old queue $\mathcal{E}$ deprived of element $e$ (namely $\mathcal{E}'''$, by definition of Sel).
In the above rule we make use of an auxiliary relation, namely $A \uparrow P :: (\mathcal{C}, \mathcal{E}) \xrightarrow{L} (\mathcal{C}', \mathcal{E}')$. Such a relation models labelled transitions of the HA $A$, and $L$ is the set containing the transitions of the sequential automata of $A$ which are selected to fire. We call $\xrightarrow{L}$ the step-transition relation in order to avoid confusion with transitions of sequential automata. When confusion may arise, we call the latter sequential transitions. $P$ is a set of transitions. It represents a constraint on each of the transitions fired in the step, namely that it must not be the case that there is a transition in $P$ with a higher priority. So, informally, $A \uparrow P :: (\mathcal{C}, \mathcal{E}) \xrightarrow{L} (\mathcal{C}', \mathcal{E}')$ should be read as “$A$, on status $(\mathcal{C}, \mathcal{E})$ can perform $L$ moving to status $(\mathcal{C}', \mathcal{E}')$, when required to perform transitions with priorities not smaller than any in $P$”. Obviously, no restriction is made on the priorities for $H$ in Definition 12, but set $P$ will be used to record the transitions a certain automaton can do when considering its sub-automata. More specifically, for sequential automaton $A$, $P$ will accumulate all transitions which are enabled in the ancestors of $A$. The deduction system for $\xrightarrow{L}$, which we call the core semantics, is shown in Fig. 3 where the following auxiliary functions are used:

**Definition 13 (Enabled transitions).** For $A \in F$, generic set of states $\mathcal{C} \in \text{Conf}_H$ and generic environment $\mathcal{E} \in (\Theta E)$,

(i) the set of all the enabled local transitions of $A$ in $(\mathcal{C}, \mathcal{E})$, $LE_A \mathcal{C} \mathcal{E}$ is defined as:

$$LE_A \mathcal{C} \mathcal{E} = \{ t \in \delta_A \mid [(\text{SRC} t)] \cup (\text{SR} t) \subseteq \mathcal{C} \land (\text{EV} t) \in \mathcal{E} \land (\mathcal{C}, \mathcal{E}) \models (G t)\}$$

(ii) the set of all enabled transitions of $A$ in $(\mathcal{C}, \mathcal{E})$ considered as an HA, i.e., including those of descendents of $A$, $E_A \mathcal{C} \mathcal{E}$ is defined as follows:

$$E_A \mathcal{C} \mathcal{E} = \bigcup_{A' \in (\Theta A)} LE_{A'} \mathcal{C} \mathcal{E}.$$ 

The following two lemmata will turn useful for proving the main theorem on the operational semantics:

**Lemma 2.** For $s \in \mathcal{C} \cap \sigma_A$ and $A_j \in \rho_A s$ the following holds: $E_{A_j} \mathcal{C} \mathcal{E} = (E_A \mathcal{C} \mathcal{E}) \cap (\mathcal{F} A_j)$.

**Lemma 3.** (i) $E_A \mathcal{C} \mathcal{E} \subseteq (\mathcal{F} A)$; (ii) $E_A \mathcal{C} \mathcal{E} = LE_A \mathcal{C} \mathcal{E} \cup (\bigcup_{A' \in \rho_A} E_{A'} \mathcal{C} \mathcal{E})$.

**Remark 1.** An obvious consequence of the above lemma is that $E_{A_j} \mathcal{C} \mathcal{E} \subseteq E_A \mathcal{C} \mathcal{E}$.

Finally, $A \uparrow P :: (\mathcal{C}, \mathcal{E}) \xrightarrow{L}$ will stand for: there exist $\mathcal{C}'$ and $\mathcal{E}'$ such that $A \uparrow P :: (\mathcal{C}, \mathcal{E}) \xrightarrow{L} (\mathcal{C}', \mathcal{E}')$. Finally, for state $s$ and set $S \subseteq \mathcal{F} (\rho s)$, such that $s \preceq s''$ for all $s'' \in S$, the closure of $S$, $(\mathcal{C} s S)$, is defined as the set $\{s' \mid \exists s'' \in S. s \preceq s' \preceq s''\}$.

In the core semantics, the progress rule establishes that if there is a transition of $A$ enabled and the priority of such a transition is “high enough” then the transition fires and a new status is reached accordingly. The composition rule stipulates how automaton $A$

\footnote{$((\mathcal{C}, \mathcal{E}) \models g$ means that guard $g$ is true of status $(\mathcal{C}, \mathcal{E})$. Its formalization is immaterial for the purposes of the present paper. Here we only point out that obviously no guard on events is satisfied in any status $(\mathcal{C}, [-])$. In the deduction rules, we will relax the requirement $\mathcal{C} \in \text{Conf}_A$ and we will assume $\mathcal{C} \in \text{Conf}_H$. This allows the use of guards which make reference to non-local states.}
delegates the execution of transitions to its sub-automata and these transitions are propagated upwards. Finally, if there is no transition of \( A \) enabled with priority “high enough” and moreover no sub-automata exist to which the execution of transitions can be delegated, then \( A \) has to “stutter”, as enforced by the stuttering rule. In our example, when \( \{S, ON, U, SHW, MTE\} \) is the current configuration and \( out \) is offered by the environment, the progress rule can be applied to the HA characterized by TV for \( off_2 \). As a result, TV enters configuration \( \{OFF, DIS\} \). Similarly, the progress rule can be applied to USER for firing transition, say, \( u_5 \). Using the composition rule we then deduce that starting from the status \( \{S, ON, U, SHW, MTE\}, \{out\} \) TV_SYS produces a step-transition labelled by \( \{off_2, u_5\} \) while moving to configuration \( \{S, OFF, DIS, U\} \) and delivering \( mte \) to the environment.

The following lemma expresses a safety property of the operational semantics w.r.t. \( P \): it essentially states that only transitions with a “high enough” priority are fired.

**Lemma 4.** For all \( L \in 2^{(\mathcal{F} \times H)} \), \( t \in L \), the following holds: \( A \uparrow P :: (\emptyset, \emptyset) \xrightarrow{L} (\emptyset, \emptyset') \) if and only if \( \emptyset \).
\((\mathcal{C}, \mathcal{E}) \xrightarrow{L_j} (\mathcal{C}_j, \mathcal{E}_j)\), according to (3) of the composition rule. The following steps prove the assert:

\[ t \in L_j \land A_j \uparrow P' :: (\mathcal{C}, \mathcal{E}) \xrightarrow{L_j} (\mathcal{C}_j, \mathcal{E}_j) \]

\[ \Rightarrow \quad \text{[Induction Hypothesis]} \]

\[ \not\exists t' \in P \cup LE_A \mathcal{C} \mathcal{E} . \pi t \sqsubseteq \pi t' \]

\[ \Rightarrow \quad \text{[Set Theory]} \]

\[ \not\exists t' \in P . \pi t \sqsubseteq \pi t'. \quad \square \]

The following theorem shows that our operational semantics satisfies the major behavioural requirements informally defined in [35,37] and briefly recalled in Section 2.

**Theorem 1.** For all \( L \subseteq (\mathcal{T}, A) \), \( A \uparrow P :: (\mathcal{C}, \mathcal{E}) \xrightarrow{L} \) if and only if \( L \) is a maximal set, under set inclusion, which satisfies all the following properties: (i) \( L \) is conflict-free, i.e., \( \forall t, t' \in L . \neg t \# t' \); (ii) all transitions in \( L \) are enabled in the current status, i.e., \( L \subseteq E_A \mathcal{C} \mathcal{E} \); (iii) there is no transition outside \( L \) which is enabled in the current status and which has higher priority than a transition in \( L \), i.e., \( \forall t \in L . \not\exists t' \in E_A \mathcal{C} \mathcal{E} . \pi t \sqsubseteq \pi t' \); and (iv) all transitions in \( L \) respect \( P \), i.e., \( \forall t \in L . \not\exists t' \in P . \pi t \sqsubseteq \pi t' \).

**Proof.** The proof is carried out for the two implications separately. The direct implication is proved by structural induction, while the reverse implication is proved by derivation induction. The detailed proof can be found in Appendix A. \( \square \)

### 3.2. Operational semantics: multicharts

In this section, we shall show how the modularity of our semantics definition allows us to re-use the core semantics in order to define multicharts semantics. Dealing with a multichart implies the use of a naming and addressing schema in order to deliver output events to the correct queues. In this paper, we assume the existence of a naming schema and we equip events with explicit information about its destination.

More specifically, a multichart is a tuple \((SD_1, \ldots, SD_k)\) of UMLSDs, where each \( SD_j \) is uniquely identified by \( j \) and where all actions labelling transitions are of the following form: \((e, w)\) with \( e \) an event and \( w \subseteq \{1, \ldots, k\} \). The intended meaning of the destination \( w \) is the obvious one: \( e \) has to be delivered to the input-queue of each \( SD_j \) for \( j \in w \). Moreover, we assume that the sets of states of \( SD_i \) and \( SD_j \) are disjoint for \( i \neq j \).

Given multichart \((SD_1, \ldots, SD_k)\), let \((H_1, \ldots, H_k)\) the tuple of HAs associated to \( SD_1, \ldots, SD_k \), each provided with its input queue \( \mathcal{E}_j \). We assume that all the sequential automata of all HAs in the tuple are distinct and the sets of their states as well as transitions are disjoint. We call such a tuple a MultiHA. The behaviour of the system is modelled as the interleaving of the STEPs of the component HAs.

**Definition 14 (Operational semantics of MultiHAs).** The operational semantics of a MultiHA \((H_1, \ldots, H_k)\) is the LTS \( Ts = (S, s^0, Act, \rightarrow) \) where (i) \( S = Conf_{H_1} \times \cdots \times Conf_{H_k} \times (\Theta E_1) \times \cdots \times (\Theta E_k) \) is the set of global statuses of \( Ts \), (ii) \( s^0 = (\mathcal{C}0_1, \ldots, \mathcal{C}0_k, \mathcal{E}0_1, \ldots, \mathcal{E}0_k) \in S \) is the initial global status, (iii) \( Act \subseteq 2^{\mathcal{C}H_1} \cup \cdots \cup 2^{\mathcal{E}H_k} \) is the set of actions of \( Ts \) and (iv) \( \rightarrow \subseteq S \times Act \times S \) is the step-transition relation defined in the sequel.
In order to define the new step-transition relation all what is needed is just to replace the top level rule given in Definition 12 by a different one, while the core semantics given in Fig. 3 remains unchanged. The new top level rule makes use of two further operations on environments, \( \downarrow \) and \( \text{ext} \). By \( (Y \downarrow z) \) we mean the new structure obtained by first removing from \( Y \) those elements \((e, w)\) such that \( z \not\in w \) and then cleaning up each element of the resulting structure by removing its destination \( w \). For instance, for FIFO queue \( Y = ((a, \{1, 2\}), (c, \{3\}), (d, \{1, 3\})) \) we have \((Y \downarrow 1) = (a, d), (Y \downarrow 3) = (c, d)\). Function \( \text{ext} \) is defined by \( \text{ext} z XY = \text{join} X (Y \downarrow z) \).

**Definition 15 (Transition relation of MultiHAs).**

\[
\begin{align*}
& j \in \{1, \ldots, k\} \quad \text{(0)} \\
& (\text{Sel} \not\in e e'' \not\in) \quad \text{(1)} \\
& H_j \uparrow \emptyset : (e_1, e_j, e_k) \xrightarrow{L} (e_j', e') \quad \text{(2)} \\
& (e_1, \ldots, e_j, \ldots, e_k, (\text{ext} 1 e_1 e'), \ldots, (\text{ext} J e'' e') \cdots (\text{ext} k e_k e'))
\end{align*}
\]

As should be clear from Definition 15, a step-transition of the MultiHA coincides with a step-transition of one of its components, say \( H_j \). The new status differs from the old one because of the change in the configuration of \( H_j \), from \( e_j \) to \( e_j' \), and the change in the queues. In particular, the selected event is dequeued from the queue \( e_j \) of \( H_j \), leading to \( e'' \) and all queues of the system are updated as a result of the actions \( e_j' \) of the step-transition of \( H_j \).

4. Model-checking UML statecharts diagrams

A formal semantics of a language is not only useful as a reference, precise and unambiguous definition of behaviour, but even more for the development of reliable tools to support formal verification of models expressed in the language. One important class of tools are the model-checking tools. Model-checking techniques [14] have been defined to verify system properties, expressed as temporal logic formulas, on finite state models of the behaviour of systems. Once a model of a system has been generated, the properties are automatically verified by model-checking tools and therefore this kind of tools can be easily used also by non-expert users. Many prototype verification environments are currently available (e.g. [2,7,8,16]). In case of systems which are described in terms of the actions they perform and the related state changes it is convenient to use also an action-based temporal logics to express their properties. In this paper, we choose the branching time, action based, temporal logic ACTL [13] and the model-checkers available in the JACK environment [2].

4.1. JACK and ACTL

JACK [2] is an environment based on the use of process algebras, automata and temporal logic formalisms, which supports many phases of the system development process.

The idea behind the JACK environment is to integrate different specification and verification tools in order to provide an environment in which a user can choose from several
verification tools by means of a user-friendly graphic interface. This last feature is quite an important one since it is nowadays widely recognized that there is no single specification and verification technique which can cover all aspects of system design in a satisfactory way; rather, different techniques and tools match different stages of design.

The FC2 format, i.e., the common representation format used in JACK for data, makes it possible to exchange information among the tools integrated in the environment and to easily add other tools. The FC2 format allows a labelled transition system (i.e., an automaton) to be represented by means of a set of tables that keep the information about state names, arc labels, and transition relations between states. The format allows nets of automata to be represented as well.

The editing tools integrated in JACK (MAUTO and ATG) allow specifications be described both in textual form and in graphical form, by drawing automata. Moreover, the tools provide sophisticated graphical procedures for the description of specifications as networks of processes. This supports hierarchical specification development.

Once the specification of a system has been written, JACK permits the construction of the automaton corresponding to the behaviour of the overall system, by using either MAUTO or FC2LINK and HOGGAR (which is a tool based on binary decision diagrams); this is the model of the system. Moreover, by using MAUTO or HOGGAR, automata can be minimized with respect to various (bisimulation) equivalences. The model of a system can be further analysed by means of the action based model-checker AMC that is included in the JACK tool set. The AMC allows the expression and verification of action based branching time temporal logic formulas (ACTL).3

The main advantage in using JACK stems from its independence from the particular notation used for modelling the systems of interest. This is a consequence of the fact that all JACK tools share the common FC2 format. Given a new notation equipped with its formal LTS based semantics, its embedding in JACK amounts to implementing such semantics using FC2. Obviously also the syntactical embedding at the front-end side must be performed. This last issue is out of the scope of this paper and presents no particular problem except the typical language engineering ones. Once the new notation is embedded in JACK the full power of the tools the environment provides is available.

ACTL [13] is a branching time temporal logic suitable to express properties of reactive systems whose behaviour is characterized by the actions they perform and whose semantics is defined by means of LTSs. In fact, ACTL embeds the idea of “evolution in time by actions” and is suitable for describing the various possible temporal sequences of actions that characterize a system behaviour. The logic can be used to define both liveness (something good eventually happens) and safety (nothing bad can happen) properties of reactive systems.

The syntax of ACTL is given by the following grammar, where $\phi$ denotes a state formula and $\mu$ and $\mu'$ are action formulas:

$$\phi ::= \text{true} \mid \neg \phi \mid \phi \land \phi' \mid [\mu] \phi \mid \text{AG} \phi \mid A[\phi] \mid U[\mu'] \phi' \mid E[\phi] \mid U[\mu'] \phi'. $$

The only atomic proposition allowed in state formulas in the above grammar is true. The reason for that is that currently we deal only with action-based formulas and no further

3 Detailed information about JACK is available at http://rep1.iei.pi.cnr.it/projects/JACK. At the same address, there is a downloadable version of the JACK package.
state-based information is considered. Action formulas are defined by the grammar given below, where $\alpha$ is an element of the set of observable actions of the system:\footnote{In the context of this paper we are not concerned with internal actions, usually denoted by $\tau$, since they are not necessary for modelling UMLSDs steps.}

$$\mu ::= \text{true} \mid \alpha \mid \mu \land \mu \mid \neg \mu.$$  

The formal semantics of ACTL is given in [13]. Here we provide only an informal description of such semantics. In the following, we will often use $\mu$ as a shorthand for “an action satisfying $\mu$”.

Any state satisfies $\text{true}$. A state satisfies $\neg \phi$ if and only if it does not satisfy $\phi$; it satisfies $\phi \& \phi'$ if and only if it satisfies both $\phi$ and $\phi'$. A state satisfies $[\alpha] \phi$ if all states directly reachable from such a state via $\alpha$ satisfy $\phi$. The meaning of $\text{AG} \phi$ is that $\phi$ is true now and always in the future.

A state $P$ satisfies $A[\phi \{\mu\} U \{\mu'\} \phi']$ if and only if in each path exiting from $P$, $\mu'$ will be eventually executed. It is also required $\phi'$ to hold after $\mu'$, and all the intermediate states to satisfy $\phi$; finally, before $\mu'$ only $\mu$ actions can be executed. The formula $E[\phi \{\mu\} U \{\mu'\} \phi']$ has the same meaning, except that it requires at least one path beginning in $P$, and not all of them, to satisfy the given constraint. A useful formula is $A[\phi \{\text{true}\} U \{\mu'\} \phi']$ where the first action formula is true, this means that any action can be executed before $\mu'$.

Some derived operators can be defined: $\text{false}$ stands for $\neg \text{true}$; $\phi \mid \phi'$ – respectively, $\mu \lor \mu'$ – stands for $\neg (\neg \phi \& \neg \phi')$ – respectively, $\neg (\neg \mu \land \neg \mu')$; $\langle \mu \rangle \phi$ stands for $\neg [\mu] \neg \phi$; finally, $\text{EF} \phi$ stands for $\neg \text{AG} \neg \phi$ (this is the eventually operator, whose meaning is that $\phi$ will be true sometime in the future). The meaning of $\langle \mu \rangle \phi$, $\text{AG} \phi$ and $A[\phi \{\mu\} U \{\mu'\} \phi']$ is graphically shown in Fig. 4.

As an example of an ACTL formula expressing a property of the TV system presented in Section 2, consider the following: $\text{AGEF} \langle \text{out} \rangle \text{true}$. The formula says that “in every (A) computation, in every (G) status of such a computation there exists (E) a (sub-)computation, starting from such a status, where action out is eventually (F) performed”. In other words, the formula expresses the requirement that from each status there exists a finite sequence of steps for powering the TV off.
This is an example of those properties which are not expressible in any linear time temporal logics. Needless to say, properties like the above one are particularly well suited for describing the existence of “safe” states reachable from any state of a system and so they play a major role in the validation of, among others, dependable systems.

Model-checking for ACTL can be performed in linear time with respect to the size of the LTS and the number of temporal operators in the formula to be checked.

For the purpose of model-checking UMLSDs, observable actions may take any of the following forms: e/ or simply e, denoting trigger event e, /a referring to action a and the most general e/a for trigger event e producing action a.

The difference between state-based and action-based formulas is less strong than that between branching-time and linear-time in the sense that in principle each state-based formula can be expressed as an action based formula and vice versa [13]. However, often one form may be more convenient than the other depending on the kind of property to express and the model. Often, when properties clearly concern the temporal order of actions that may occur in a system, the action based formulas are more appropriate. In the following example the property that the TV set USER can perform an off action only after s/he performed an on action is expressed.

\[ \sim E[\text{true}\{\sim /on\} U /\text{off} \text{true}] \]

Literally it says that there is no path (\(\sim E\)) where no on action (\(\sim /on\)) occurs until (\(U\)) an off action happens (/off).

If we want to require that the USER can never turn the TV set off when it is not on, i.e., no two consecutive off actions can occur without an on action having occurred in between them, we need to verify in addition that:

\[ \text{AG}([/\text{off}] \sim E[\text{true}\{\sim /on\} U /\text{off} \text{true}] \text{true}) \]

Stating literally that always in every state (\(\text{AG}\)) when an off action has occurred ([/off]) there is no path (\(\sim E\)) where no on action (\(\sim /on\)) occurs until (\(U\)) an off action happens (/off).

So, the two expressions together cover both the situation before the first occurrence of an off action and that after the first occurrence of an off action.

4.2. Building the semantics automaton

In order to model-check a multichart specification we have to generate the semantics automaton of the related MultiHA. The LTS thus obtained will then be used as the model on which to prove the satisfiability of the ACTL formulas expressing the desired properties of the system.

In Fig. 5 an algorithm for building the semantics automaton of a MultiHA is given. The algorithm uses directly the operational semantics defined in Section 3 and is just a customization to such semantics of standard state/transition enumeration algorithms for building a complete state-space using a transition relation. Consequently, we prefer to keep the description of the algorithm rather informal, given in a kind of pseudo-Pascal notation, where set notation is freely used. The algorithm is straightforward, thus we refrain from giving a rigorous correctness proof. Rather, we briefly describe the main steps of the algorithm in the sequel, pointing out arguments supporting its correctness.

The input to the algorithm is a MultiHA \(H\), composed of \(k\) HAs, and its initial global state \((\emptyset_0, \varepsilon_0)\), of type \text{array}[1..k] of Config \(\times\) \text{array}[1..k] of Queue. It uses relations \(A \uparrow\)
Types
HA (* hierarchical automata *)
State (* states of seq. automata *)
Transition (* transition (labels) of seq. automata *)
Event (* events *)
Destination (* destinations *)
Queue = queue_of Event (* no particular policy fixed *)
DQueue = queue_of Event × Destination
Config = set_of State
Status = array[1..k] of Config × array[1..k] of Queue
Act = set_of Transition
S-Transition = Status × Act × Status (* STEP-transitions *)

Variables
H : array[1..k] of HA
S : set_of Status (* currently generated statuses *)
St : set_of Status (* currently analyzed statuses *)
Tr : set_of S-Transition (* currently generated STEP-transitions *)
C, G : array[1..k] of Config
C' : Config
e : Event
E, F : array[1..k] of Queue
E' : DQueue
E'' : Queue

Initialization
H := ...
S := {(C₀, E₀)};
St := Ø;
Tr := Ø;
repeat
begin
select any element from S and assign it to (C, E);
S := S \ {(C, E)};
St := St ∪ {(C, E)};
forall j in {1, ..., k} do
forall (e, E'') such that (Sel E'[j] e E'') do
forall L, C', E' such that H[j] ↑ Ø :: (C[j], {e}) → L (C', E') do
begin
forall h in {1, ..., k} do
begin
G[h] := if h = j then C' else C[h];
F[h] := if h = j then ext h E'' E' else ext h E[h] E';
end;
S := S ∪ {(G, F)} \ St;
Tr := Tr ∪ {(C, E), L, (G, F)}
end
until S = Ø
end

Fig. 5. Semantics automaton algorithm.
$P :: (⟨C, \{e\}) \xrightarrow{L} (⟨C', \{e'\})$ and $(Sel \ e \ e')$, defined in Section 3, as Boolean functions. Also function $ext$ on event queues is directly used in the algorithm.

At each main iteration, a previously generated global state $(C, \mathcal{E})$ is taken (and removed) from the set stored in variable $S$, analysed and added to the set stored in variable $St$, initialized to $\emptyset$. So $S$ holds the set of states which have been generated but not yet analysed. The initial global state $(C_0, \mathcal{E}_0)$ is the first to be generated at initialization. When analysing global state $(C, \mathcal{E})$, all possible step-transitions, for all possible events which can be selected from the input queues $\mathcal{E}[j]$ of every component, $H[j]$, of $H$ are computed and added to the set stored in variable $Tr$, initialized to $\emptyset$. As it can easily be seen from the definition of the algorithm, these are all and only the step-transitions originating from $(C, \mathcal{E})$ according to Definition 15. For each such step-transition of the $j$th component $HA$ of $H$, the corresponding next global state is computed, taking care that the only configuration which is changed is that of such a component and that the event which triggered the step is properly removed from the input queue from which it had been taken. All input queues are updated with the result events of the step-transition. Each next global state generated as above is added to the set stored in variable $S$ unless already analysed. The termination condition is reached when there are no more (new) global states to analyse. At this point, the output of the algorithm is given by the values of variables $St$ and $Tr$ holding, respectively, the set of global states and the set of step-transitions of the semantics automaton.

The algorithm terminates only if the semantics automaton of the MultiHA is finite, i.e., there is a finite number of global states and a finite number of step-transitions. Notice that, although the number of possible configurations of any UMLSD is always finite, the particular semantics chosen for the input queue may lead to an infinite number of global states. If this happens, traditional model-checking is obviously not a viable approach to validation, unless an implementation related maximal size of the queues is fixed and its impact on the correctness of the validation verdicts is assessed.

When the semantics LTS is finite, the algorithm generates it completely. In the very worst case, the number of states and transitions of such an LTS may grow exponentially with the number $k$ of component UMLSDs of the Multichart, the number $p$ of components of each UMLSD and the length $n$ of the input queues.

The negative impact of (the buffer length of) asynchronous communication on the size of the state-space is well known (see e.g., [22]). The impact of the number of component UMLSDs on the overall size of the LTS is just an instance of the state-explosion problem with interleaving semantics. The extent to which the number of components of each UMLSD affects the overall size of the LTS is very much related to the nesting structure of the UMLSD. The very worst case is represented by a UMLSD composed by $p$ parallel AND-states, each composed of $s$ basic OR-states and such that, within each AND-state there is a transition from every (OR-)substate to every other (OR-)substate for each input event. The total number of configurations for such a UMLSD is $s^p$, which, for a multichart of $k$ components with input queues of length $n$ may give raise to a number of states of the order of $s^{pk}v^n$, where $v$ is the number of events in the multichart.

Anyway, such a situation is rather unrealistic because any good UMLSD model should not be totally “flat”, unless composed by a very small number of relatively small machines. Rather, it should be characterized by a high degree of nesting, since state refinement and hierarchy are powerful abstraction mechanisms provided by UMLSDs. Nesting induces state dependencies in the configurations, thus limiting their total number drastically. Moreover, usually only critical parts of the system under consideration are subject to formal
validation, so, in general, the number of statechart components of a multichart should not be too high, being such critical parts typically relatively simple. With respect to the impact of asynchronous communication, the only solution is to keep the length of the queues short.

The application of the algorithm to our single statechart example yields an automaton with more than 40 statuses and a few hundreds of step-transitions. In order to be able to draw any sensible picture in this paper, in the sequel we will refer to a simplified version of the example, which is shown in Figs. 6 and 7. Essentially, we abstract from the video-text functionality and we replace the user with a less liberal behaviour. Now the user is supposed to issue an on command before issuing snd, mte and off ones. Similarly, after having issued an on signal, (s)he cannot request to power the TV set off (out) before having switched it off (off). Transition u0 of the original user is split into transitions u0I and u0A, both labelled by (−, ∅, ∅, ε) in the obvious way. The information related to all other transitions can easily be derived from Table 1. The automaton obtained from the HA of Fig. 7 is shown in Fig. 8. For readability reasons, in the figure statuses are labelled with the environment value (top) and the current configuration (bottom) – notice that in this example the event queue is invariantly at most one element long. We use a single statechart example to highlight the model-checking aspects rather than the multichart aspects. Note however that the algorithm is defined for MultiHAs.
4.3. Implementing the semantics automata

Since we are using LTS semantics we can easily represent the semantics automata as FC2 objects. In the operational semantics, statuses are composed of configuration-environment pairs. Step-transitions are labelled by sets of labels of transitions from the sequential automata (we shall call the latter labels *sequential labels* in the sequel). A proper coding of statuses and step-transition labels is necessary. The simplest one is to use strings both for naming the states of the LTS and for labelling its step-transitions. So, for instance, the status \((\{OFF, STB, UI\}, out)\) is trivially coded to a state called ‘OFF:STB:UI$out’, with $ being used as a separator and letting the elements of sets being ordered alphabetically. Analogously, the step-transition label \(\{st, u1\}\) is implemented as the list ‘st:u1’. In the following, for status \((C, E)\) (respectively, step-transition label \(L\)) we let \([| (C, E) |]\) (respectively, \([| L |]\)) denote its coding.

4.4. Transforming logical formulas

Once the LTS has been generated as an FC2 file, the action model-checker AMC can be used to formally verify the UML dynamic description of a system. In the following we show the steps necessary to that purpose using our running example. The generalization to generic multicharts is straightforward.

The first step is to express properties using ACTL. For instance, in our example of Fig. 6, one could be interested in proving that the TV set USER can perform the first *off* action,
only after (s)he performed an on action. The ACTL formula below formalizes the above property; more specifically it states that there exists no path where an off action occurs as output action (/off) with no on action having occurred as output (\sim /on) before:  
\[ \sim E[true\{\sim /on\}U{/off}true]. \]

The above formula cannot directly be checked against the LTS of Fig. 8 since the labels in the transitions of the UMLSD of Fig. 6 have been coded into proper sequential labels in the HA of Fig. 7. Such sequential labels in turn occur as elements in the labels of the step-transitions in the semantics LTS. So, in order to check the above property against the LTS two transformations are needed: the observable actions referring to the UMLSD must be mapped into labels of its HA and then into those of the LTS. This mapping can be completely automatic and must guarantee that the semantics of the logic formula is preserved. In the present section we will briefly show how this transformation can be done, leaving out all the actual implementation details.

The first transformation generates an intermediate formula and is due to purely syntactical issues, namely the fact that we use HAs as abstract syntax for UMLSDs. The actual mapping is performed using the transition labels table of the HA (see Table 1). In particular, for each observable action \( \alpha \) occurring in the formula, the set \( SL(\alpha) = \{t_1, \ldots, t_{|SL(\alpha)|}\} \) of the (sequential labels of) the transitions in which the event/action occur as specified by \( \alpha \) is computed. This requires a complete sequential scan of the table, with a cost in time which is linear with the total number of sequential transitions. The memory overhead cost is negligible.

In our example the observable actions are /off and /on. There is only one sequential label, \( u3 \) (respectively, \( u4 \)), where off (respectively, on) is used as output action, i.e., AC \( u3 = \text{off} \) (respectively, AC \( u4 = \text{on} \)). Notice that if in the formula an observable action of the form \( e/a \) occurs, then the set of sequential labels to be computed is that of those \( x \) such that both \( EVx = e \) and \( ACx = a \) hold.

Using the above information, the intermediate formula is generated by replacing in the original formula every occurrence of each observable action \( \alpha \) with the following formula 
\[ t_1 \lor \cdots \lor t_{|SL(\alpha)|} \]

namely the logical disjunction of the sequential labels contained in \( SL(\alpha) \). Such \( t_1, \ldots, t_{|SL(\alpha)|} \) will be the atomic propositions of the generated intermediate formula.

The meaning should be clear: any transition (of any sequential automaton) which is labelled with an element of the above set obviously satisfies the atomic action formula \( \alpha \). This guarantees that this first transformation preserves the meaning of the formula. Our sample formula is converted into the following one:
\[ \sim E[true\{\sim u4\}U{u3}true]. \]

The second transformation has to do with the semantics of UMLSDs. We recall that each step-transition corresponds to the firing of a set of transitions of sequential automata and is labelled by such a set. Let us consider a (sequential) transition \( t_j \) occurring in the intermediate formula as atomic proposition. Intuitively such an atomic proposition holds whenever \( t_j \) is fired, regardless of the particular STEP in which this happens. Consequently, if \( L(t_j) = \{L_1, \ldots, L_{|L(t_j)|}\} \) is the set of the labels of all the step-transitions in the...
semantics LTS which contain \( t_j \) as an element, the final formula is obtained by replacing in the intermediate formula every occurrence of each sequential label \( t_j \) with the formula

\[
\begin{align*}
\| L_1 \| \lor \cdots \lor \| L_{|L(t_j)|} \|
\end{align*}
\]

denoting the logical disjunction of the (coding of) the step-transition labels contained in \( L(t_j) \). Such \( \| L_1 \|, \ldots, \| L_{|L(t_j)|} \| \) will be the atomic propositions of the generated formula.

The information for computing sets \( L(t_j) \) is fully available once the semantics LTS is computed and the collection of such sets for all the atomic propositions appearing in the intermediate formula can be computed with a single visit of the LTS. Obviously, the size of the LTS affects the efficiency of the translation.

Applying this last step to the intermediate formula of our running example yields to

\[
\neg E[\text{true}\{\neg (u4 \lor st:u4 \lor d:u4 \lor off1:u4)\} \cup \{u3 \lor on:u3 \lor m:u3 \lor sn:u3\}\text{true}]
\]

Since all the observable actions of the above formula are labels of the LTS, the formula is ready for model-checking using JACK. Both this formula and the stronger requirement discussed in Section 4.1 are satisfied by the UMLSD of Fig. 6.

Note that the transformations preserve all the necessary information that is needed to report back the results in a form that relates to the original UMLSD specification. This information consists of the configurations and the original transitions that are used to present the trace of a possible counterexample when a property is not satisfied by the model.

5. Conclusions

In this paper, we presented a formal proof of the correctness of the operational semantics for a behavioural subset of UMLSDs proposed in [27], with respect to major requirements formulated in the official UML semantics (informal) definition that concern behavioural aspects of UMLSDs. In particular we proved that each STEP of the operational semantics fires a maximal set of currently enabled, non-conflicting transitions such that there is no enabled transition outside the set which has higher priority than a transition in the set.

Structural and derivation induction are heavily used in the proofs. This shows the advantages of a hierarchical and recursive definition of the operational semantics. The rigorous proof of features of the notation greatly contributes in better understanding the notation itself and its conceptual implications, specially with respect to semantics issues, thus increasing the confidence on its formal definition and providing arguments for its correctness. Other formal operational semantics have been proposed for UMLSDs in the literature, but to our knowledge, none of them have been equipped with any correctness proof.

The proposed semantics definition uses a core kernel that is amenable to extensions with important other features of UMLSDs. One such a feature is the use of several statecharts communicating over different queues. We have shown that the modularity of our semantics for single statecharts allows a straightforward extension to the use of multiple statecharts.

In [18] another extension to the single statechart semantics has been proposed. There we describe an orthogonal extension of UMLSDs with stochastically timed variables. In particular, following the stochastic automata approach of D’Argenio [10,11], we enriched UMLSDs with random clocks which can be set when states are entered and which can be used as guards for transitions: a transition can fire only when all clocks guarding it reach zero. The operational semantics definition has been extended in order to deal with random clocks. The extension is technically simple and allows us to use powerful analysis
techniques, including discrete event simulation. Furthermore, it is \textit{orthogonal} in the sense that the automaton of the basic, untimed, operational semantics is the same as that of the stochastic semantics, once clock information is removed. Orthogonality can be proven by derivation induction \cite{18} and sets the formal link of the extension with the original semantics. Although in \cite{18} we only deal with single statecharts the extension can be combined in a straightforward way with the multichart one following the same approach as for single untimed UMLSD described in this paper.

Another extension of our operational semantics is defined in \cite{28} where we develop a formal \textit{testing theory} for UMLSDs. A new formal operational semantics is defined, which uses the same core semantics as in the present paper but which is better suited for testing theory. The new semantics is proved consistent with our original ones and is guaranteed to generate only \textit{finite} state machines. Proper testing pre-orders and equivalences are defined which allow us to equate/distinguish systems on the basis of their interaction with the surrounding environment, abstracting from their internal structure. The resulting theory can be used as a formal framework for the derivation of test suites from system specifications. Finally, we provide a way for effective automatic verification of testing equivalence of our statecharts, based on existing techniques and \textit{tools}.

Our experience shows thus that our definition of the semantics of UMLSDs greatly facilitates formal reasoning about UMLSDs as well as the definition of extensions/modifications of the notation and its semantics. In particular, it allows us to easily formalize the relations between the extensions and the original semantics. Proving such relations turns out to be relatively easy as well.

In this paper, a model-checking approach to the formal verification of UML statechart diagrams based on the JACK verification environment has also been presented. The approach is based on ACTL, a branching time action-based temporal logic, and on our formal semantics of UMLSDs. ACTL embeds the idea of “evolution in time by actions” and is suitable for describing the various possible temporal sequences of actions that characterize a system behavior. To our knowledge, this is the first work addressing \textit{branching time} and \textit{action based} model-checking of UMLSDs.

The work presented in this paper forms the basis for the implementation of a hierarchical automata branching time model-checking facility within the verification environment JACK, which will be then extended in order to deal with the graphical nature of UMLSDs. The AMC model-checker provided by JACK requires the full state-space to be computed in order to perform its analysis. This may result in an exponential blow-up. This is quite a common problem when the full state-space enumeration approach is taken. For this reason, alternative model-checking approaches are being developed. In particular, in the context of JACK, efficient on-the-fly model-checking algorithms are being developed for systems modelled as networks of automata \cite{19}. Consequently, a next step in our work with UMLSDs will be the representation of hierarchical automata as networks of automata which cooperate for simulating the state hierarchy as well as transition priorities. Moreover, we are currently investigating the application of on-the-fly techniques to UMLSDs.

On the other hand, many successful model-checking experiments of real-life system models have been reported in the literature where full state-space enumeration is used, despite the state explosion problem in the worst case. The success of such experiments usually stems from the proper use of abstraction in modelling the systems \cite{1,6}. We believe that similar approaches can be followed for UMLSDs, which, moreover, also provide “built-in” abstraction/refinement mechanisms like composite states and hierarchy.
Finally, once the various model-checking techniques will have been explored for the restricted subset of UML statechart diagrams used in this paper, extensions to such a subset will be considered. We think it would be particularly helpful to extend UMLSDs with variables and, correspondingly, enrich state formulas with predicates on the values of such variables. This way, both state-based and action-based model-checking would be available for verification of UMLSDs.

Appendix A

Proof of Theorem 1. The proof of the theorem is carried out by induction either on the length of the derivation for proving $A \uparrow P :: (\mathcal{E}, \mathcal{E}) \xrightarrow{L} (\mathcal{E}', \mathcal{E}')$ or on the structure of the subset of $F$ affected by $\mathcal{E}$. With respect to structural induction, let $F_\mathcal{E}$ be the set $\{A \in F \mid \mathcal{E} \cap \sigma_A \neq \emptyset\}$. It is easy to define a relation on $F_\mathcal{E}$ such that $X$ is related to $Y$ iff $s \in \mathcal{E} \cap \sigma_Y$ and $X \in (\rho_Y s)$. Notice that since $\mathcal{E}$ is a configuration, for each $A$ in $F_\mathcal{E}$ there is a unique state $s \in \sigma_A \cap \mathcal{E}$. 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 and the bottom elements are those $X$ such that $\rho s = \emptyset$ for $s \in \mathcal{E} \cap \sigma_X$ [31].

We prove the two implications separately.

Part 1. Direct implication: if $L$ is a maximal set which satisfies properties (i)–(iv), then $A \uparrow P :: (\mathcal{E}, \mathcal{E}) \xrightarrow{L}$. We proceed by induction on the structure of $F_\mathcal{E}$.

Base case ($\rho_A s = \emptyset$). If $L = \emptyset$, then the conditions for the stuttering rule are fulfilled, if instead $L \neq \emptyset$ then the conditions for the progress rule are fulfilled and the assert follows.

Induction step ($\rho_A s \neq \emptyset$). There are two cases depending on $\exists t \in L. t \in \delta_A$:

Case 1. $\exists t \in L. t \in \delta_A$. Notice first of all that, in order for $L$ to be conflict free, it must also be $L = \{t\}$, as it follows from the definitions of configuration and of conflicting transitions. Moreover, the hypothesis on $L$ make the conditions for the application of the progress rule fulfilled for $t$ and the assert follows.

Case 2. $\not\exists t \in L. t \in \delta_A$. Let $L_j = L \cap (\mathcal{T} A_j)$ with $\{A_1, \ldots, A_n\} = \rho s$. We prove that each $L_j$ is a maximal set which satisfies (i)–(iv) w.r.t. $A_j$ and $P \cup LE_A \not\subseteq \mathcal{E}$. We first prove that $L_j$ satisfies (i)–(iv).

(i)

$L_j \subseteq L$

$\Rightarrow$  

$[L \text{ sat. (i)}]$  

$L_j \text{ sat. (i)}$

(ii)

$t \in L_j$

$\Rightarrow$  

[definition of $L_j$]

$t \in L \cap (\mathcal{T} A_j)$

$\Rightarrow$  

[L sat. (ii)]

$t \in (E_A \mathcal{E} \mathcal{E}) \cap (\mathcal{T} A_j)$

$\Rightarrow$  

[Lemma 2]

$t \in E_{A_j} \not\subseteq \mathcal{E}$
(iii) By contradiction:
\[ t \in L_j \land t' \in E_{A_j} \land \pi t \sqsubseteq \pi t' \]
\[ \Rightarrow \{ L_j \subseteq L \text{ and Remark 1} \} \]
\[ t \in L \land t' \in E_{A_j} \land \pi t \sqsubseteq \pi t' \]
\[ \Rightarrow \{ L \text{ sat. (iii) w.r.t. } A \} \]
false

(iv) By contradiction:

Suppose \( t \in L_j \land t' \in (P \cup LE_{A_j} \land \pi t \sqsubseteq \pi t' \). Either \( t' \in P \) or \( t' \in LE_{A_j} \):

Case 1. \( t' \in P \)
\[ t \in L_j \land t' \in P \land \pi t \sqsubseteq \pi t' \]
\[ \Rightarrow \{ L_j \subseteq L \} \]
\[ t \in L \land t' \in P \land \pi t \sqsubseteq \pi t' \]
\[ \Rightarrow \{ L \text{ sat. (iv) w.r.t. } P \} \]
false

Case 2. \( t' \in LE_{A_j} \)
\[ t \in L_j \land t' \in LE_{A_j} \land \pi t \sqsubseteq \pi t' \]
\[ \Rightarrow \{ \text{Lemma 3 (ii)} \} \]
\[ t \in L \land t' \in E_{A_j} \land \pi t \sqsubseteq \pi t' \]
\[ \Rightarrow \{ L \text{ sat. (iii) w.r.t. } A \} \]
false

Now we prove that \( L_j \) is maximal. We proceed again by contradiction. Suppose there exists \( L'_j \) such that \( L_j \subset L'_j \) and \( L'_j \) satisfies (i)–(iv) w.r.t. \( A_j \) and \( P \cup LE_{A_j} \). Let \( L' = L'_j \cup \bigcup_{k=1,k\neq j}^{n} L_k \). Clearly \( L \subset L' \) and in the following we show that \( L' \) would satisfy (i) to (iv) w.r.t. \( A \) and \( P \).

(i)
true
\[ \Rightarrow \{ L'_j \text{ sat. (ii) w.r.t. } A_j; \text{ Lemma 3 (i)} \} \]
\[ L'_j \subseteq (\mathcal{F} A_j) \]
\[ \Rightarrow \{ \text{Composition rule; definition of } \mathcal{F}, ||; \text{ Lemma 1} \} \]
\[ \forall t \in L'_j, t' \in L_k, \neg(t \# t') \]
\[ \Rightarrow \{ \text{definition of } L'; L'_j \text{ and } L_k \text{ sat. (i)} \} \]
\[ L'\text{sat. (i)} \]

(ii) \( L' \subseteq E_A \land \pi t \sqsubseteq \pi t' \) since \( L'_j \) sat. (ii) w.r.t. \( A_j \) and Lemma 3 (ii) applies; moreover \( \bigcup_{k\neq j} L_k \subseteq L \) and \( L \) sat. (i) w.r.t. \( A \).

(iii) First notice that
1. \( \forall t \in L_k. \ \forall t' \in E_A \land \pi t \sqsubseteq \pi t' \) for \( k \neq j \) since \( L_k \subseteq L \) which sat. (iii) w.r.t. \( A \).
2. \( \forall t \in L'_j. \ \forall t' \in E_{A_j} \land \pi t \sqsubseteq \pi t' \) since \( L'_j \) sat. (iii) w.r.t. \( A_j \).
3. \( \forall t \in L'_j. \ \forall t' \in E_{A_k} \land \pi t \sqsubseteq \pi t' \) for \( k \neq j \) because of Lemma 1.
4. \( \forall t \in L_j. \ \forall t' \in E_A \land \pi t \sqsubseteq \pi t' \) since \( L'_j \) sat. (iv) w.r.t. \( P \cup LE_A \).
Points (1)–(4) above, together with Lemma 3 (ii) prove that \( L' \) satisfies (iii) w.r.t. \( A \).

(iv) \( \forall t \in L'. \ \exists t' \in P. \ \pi t \sqsubseteq \pi t' \) since \( L \) sat. (iv) w.r.t. \( P \) and \( L_j' \) sat. (iv) w.r.t. \( P \cup L A \in \mathcal{C} \). All the above shows that \( L' \) would satisfy (i)–(iv) w.r.t. \( A \) and \( P \) and \( L \subset L' \) which would contradict the hypothesis on \( L \).

So, by the Induction Hypothesis \( \bigwedge_{j=1}^{n} A_j \uparrow P \cup L A \in \mathcal{C} \vdash (\mathcal{C}, \mathcal{E}) \xrightarrow{L_j} (\mathcal{C}_j, \mathcal{E}_j) \). Moreover, if \( \bigcup_{j=1}^{n} L_j = \emptyset \), then \( L = \emptyset \) which, by hypothesis on \( L \) implies in turn \( \forall t \in L A \in \mathcal{C} \). \( \exists t' \in P. \ \pi t \sqsubseteq \pi t' \). This means that the composition rule can be applied and the assert is proven.

Part 2. Reverse implication: if \( A \uparrow P :: (\mathcal{C}, \mathcal{E}) \xrightarrow{L} \) then \( L \) is a maximal set which satisfies properties (i)–(iv). We proceed by induction on the length \( d \) of the derivation for \( A \uparrow P :: (\mathcal{C}, \mathcal{E}) \xrightarrow{L} (\mathcal{C}', \mathcal{E}') \).

Base case \((d = 1)\). If the derivation has length 1, then only the progress rule or the stuttering rule could have been applied. In both cases the assert follows directly from the conditions of the rules.

Induction step \((d > 1)\). In this case the composition rule must have been applied in the last step of the derivation. By the Induction Hypothesis, every \( L_j \) satisfies (i)–(iv) w.r.t. \( A_j \) and \( P \cup L A \in \mathcal{C} \). We show that \( L = \bigcup_{j=1}^{n} L_j \) is a maximal set which satisfies (i)–(iv) w.r.t. \( A \) and \( P \).

We first show that \( L \) satisfies (i) to (iv) w.r.t. \( A \) and \( P \).

(i) \( L \) sat. (i) since each \( L_j \) sat. (i) by Induction Hypothesis and for \( i \neq j \) if \( t \in L_i \) and \( t' \in L_j \), then \( \neg(t \# t') \) because of (SRC \( t \parallel SRC t' \)) and Lemma 1.

(ii)

\[
\Rightarrow \quad L_j \text{ sat. (ii) w.r.t. } A_j
\]

\[
\bigwedge_{j=1}^{n} L_j \subseteq E_{A_j} \in \mathcal{C} \in \mathcal{E}
\]

\[
\Rightarrow \quad \{ \text{Remark 1; } L = \bigcup_{j=1}^{n} L_j \}
\]

\[
L \subseteq E_{A} \in \mathcal{C} \in \mathcal{E}
\]

(iii) By contradiction:

Suppose w.l.g. \( t \in L_j \) and \( t' \in E_{A} \in \mathcal{C} \in \mathcal{E} \) with \( \pi t \sqsubseteq \pi t' \). By Lemma 3 (ii) either \( t' \in L A \in \mathcal{C} \) or \( t' \in E_{A_k} \in \mathcal{C} \) for some \( k \) such that \( A_k \in \rho s \).

Case 1. \( t' \in L A \in \mathcal{C} \in \mathcal{E} \)

\[
t \in L \land t' \in L A \in \mathcal{C} \in \mathcal{E} \land \pi t \sqsubseteq \pi t'
\]

\[
\Rightarrow \quad \{ \text{Composition rule} \}
\]

\[
t \in L_j \land t' \in L A \in \mathcal{C} \in \mathcal{E} \land A_j \uparrow P \cup L A \in \mathcal{C} \in \mathcal{E} :: (\mathcal{C}, \mathcal{E}) \xrightarrow{L_j} \land \pi t \sqsubseteq \pi t'
\]

\[
\Rightarrow \quad \{ \text{Lemma 4} \}
\]

false

Case 2. \( t' \in E_{A_k} \in \mathcal{C} \) for some \( k \) such that \( A_k \in \rho s \).

Notice first that it cannot be \( k = j \) since \( L_j \) is conflict free and \( \pi t \sqsubseteq \pi t' \) implies \( t \# t' \) by definition of priority schema. On the other hand, it cannot be \( k \neq j \) since \( \pi t \sqsubseteq \pi t' \) would violate Lemma 1.
(iv) By contradiction:

\[ t \in L \land t' \in P \land \pi t \sqsubseteq \pi t' \]
\[ \Rightarrow \{ \text{Composition rule and set theory} \} \]
\[ \exists L_j . t \in L_j \land t' \in P \cup LE_A \land \pi t \sqsubseteq \pi t' \]
\[ \Rightarrow \{ L_j \text{ sat. (iv) w.r.t. } P \cup LE_A \} \]
\[ \text{false} \]

Now we prove that \( L \) is maximal. We proceed again by contradiction. Suppose there exists \( L' \) such that \( L \subset L' \) and \( L' \) satisfies (i)–(iv) w.r.t. \( A \) and \( P \). First observe that \( L' \subseteq \bigcup_{j=1}^{n} (T A_j) \) in fact:

\[ \text{true} \]
\[ \Rightarrow \{ L \subseteq L', L' \text{ sat. (i)}, \text{ and } L' \text{ sat. (ii)} \} \]
\[ L' \cap (LE_A \land \pi t \subseteq \pi t') = \emptyset \land L' \subseteq L \]
\[ \Rightarrow \{ \text{Lemma 3 (ii)} \} \]
\[ L' \subseteq \bigcup_{j=1}^{n} E A_j \land \pi t \subseteq \pi t' \]
\[ \Rightarrow \{ \text{Lemma 3 (i)} \} \]
\[ L' \subseteq \bigcup_{j=1}^{n} (T A_j) \]

Let, for \( k = 1, \ldots, n, \) \( L_k' = L' \cap (T A_k) \). Obviously there must exist a \( j \) such that \( L_j \subseteq L_k' \) since \( \bigcup_{j=1}^{n} L_j = L \subseteq L' \). We show that \( L_j' \) satisfies (i)–(iv) w.r.t. \( A_j \) and \( P \cup LE_A \land \pi t \subseteq \pi t' \) which contradicts the induction hypothesis (maximality of \( L_j \)).

(i)
\[ L_j' \subseteq L' \]
\[ \Rightarrow \{ L' \text{ sat. (i)} \} \]
\[ L_j' \text{ sat. (i)} \]

(ii)
\[ t \in L'_j \]
\[ \Rightarrow \{ \text{definition of } L'_j \} \]
\[ t \in L' \cap (T A'_j) \]
\[ \Rightarrow \{ L' \text{ sat. (ii)} \} \]
\[ t \in (E A \land \pi t' \subseteq \pi t \land \pi t') \]
\[ \Rightarrow \{ \text{Lemma 2} \} \]
\[ t \in E A_j \land \pi t' \subseteq \pi t' \]

(iii) By contradiction:
\[ t \in L'_j \land t' \in E A_j \land \pi t \subseteq \pi t' \]
\[ \Rightarrow \{ L'_j \subseteq L' \text{ and Remark 1} \} \]
\[ t \in L' \land t' \in E A \land \pi t \subseteq \pi t' \]
\[ \Rightarrow \{ L' \text{ sat. (iii)} \} \]
\[ \text{false} \]

(iv) By contradiction:

Suppose \( t \in L'_j \land t' \in (P \cup LE_A \land \pi t' \subseteq \pi t' \). Either \( t' \in P \) or \( t' \in LE_A \land \pi t \subseteq \pi t' \).

Case 1. \( t' \in P \)
\[ t \in L'_j \land t' \in P \land \pi t \subseteq \pi t' \]
\[ \Rightarrow \{ L'_j \subseteq L' \} \]
\( t \in L' \land t' \in P \land \pi t \sqsubseteq \pi t' \)

\[ \Rightarrow \{ L' \text{ sat. (iv) w.r.t } P \} \]

Case 2. \( t' \in LE_A \not\in \mathcal{E} \)

\[ t \in L_j' \land t' \in LE_A \not\in \mathcal{E} \land \pi t \sqsubseteq \pi t' \]

\[ \Rightarrow \{ L_j' \subseteq L' \} \]

\[ t \in L \land t' \in LE_A \not\in \mathcal{E} \land \pi t \sqsubseteq \pi t' \]

\[ \Rightarrow \{ \text{Lemma 3 (ii)} t \in L' \land t' \in E_A \not\in \mathcal{E} \land \pi t \sqsubseteq \pi t' \} \]

\[ \Rightarrow \{ L' \text{ sat. (iii) w.r.t } A \} \]

false

So \( L \) is maximal. \( \Box \)

Acknowledgements

The work described in this paper has been performed in the context of the ESPRIT Project no. 27439-HIDE. One author of this paper has been partially funded by the MURST Project SALADIN (Architetture Software e Linguaggi per Coordinare Componenti Distribuite e Mobili). The authors thank Istvan Majzik for the fruitful discussions on the UML and its semantics.

References


