Statechart Diagrams provide a graphical notation for describing dynamic aspects of system behaviour within the Unified Modeling Language (UML). In this paper we present a branching time model-checking approach to the automatic verification of formal correctness of UML Statechart Diagrams specifications.

We use a formal operational semantics for building a labeled transition system (automaton) which is then used as a model to be checked against correctness requirements expressed in the action based temporal logics ACTL. Our reference verification environment is JACK, where automata are represented in a standard format, which facilitates the use of different tools for automatic verification.

1. Introduction

The Unified Modeling Language (UML) is a graphical modeling language for object-oriented software and systems [8, 9]. 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.

In this paper we focus on UML Statechart Diagrams, which are meant for describing dynamic aspects of system behaviour. In particular we discuss a simple model-checking approach to the formal verification of UML Statechart Diagrams based on the JACK verification environment [1]. The UML is a semi-formal language, since its syntax and static semantics (the model elements, their interconnection and well-formedness) are defined formally, but its dynamic semantics are specified only informally [9]. Several approaches have been proposed in the literature for the definition of a formal semantics of UML Statechart Diagrams, e.g. [14, 2, 11]. The work described in this paper is based on the operational semantics proposed in [11], which follows an approach similar to the one proposed in [12] for classical statecharts and shares its relative simplicity in the world of formal approaches to statechart semantics. To our understanding, transition priorities are dealt with neither in [14], where also state hierarchy is not allowed, nor in [2], where model checking is addressed. Both transition priorities and state hierarchy constitute main issues in our work.

Linear time model checking of classical statecharts is addressed in [4, 13]. In [4] logical properties are expressed in a graphical language which is then translated into a linear time logic. In [13] classical statecharts are translated into PROMELA, the modeling language of the SPIN model-checker. Linear-time model checking of UML Statechart Diagrams is addressed in [10] where a translation to PROMELA is given based on the semantics proposed in [11].

JACK [1] 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, independently developed at different research institutes (I.E.I.- C.N.R. and the University of Rome “La Sapienza” in Italy, and INRIA in France), 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.

JACK includes also a model-checker for ACTL [5], 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 benefits of branching time logics have been widely recognized in the literature and it is also wellknown that linear time and branching time logics are incomparable as far as the 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 [3].

The present paper is organized as follows: in Sect. 2 the subset of UML Statechart Diagrams 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 recalled in Sect. 3. Sect. 4 addresses model-checking of UML Statechart Diagrams using the JACK environment. Some conclusions are drawn in Sect. 5.

2. Basic concepts

UML statecharts are a (object-oriented) variant of classical Harel statecharts [6, 7]. The statecharts formalism itself is an extension of traditional state transition diagrams. In this section we briefly describe those features of UML statechart diagrams 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 [12, 13].

![Figure 1. Example of a UML statechart](image)

One of the main notions of statecharts is the notion 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 concurrent. Also $S$ is a concurrent state.

A transition connects a source to a target state. The transition is labeled 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 event for being triggered, the notation /action is used.

"System states” are modeled 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.

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 labeled 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 lowest transition is taken, then event txt will be delivered.

In the general case, some target substates 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 interlevel 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 interlevel one. Firing it requires the system to be in a configuration containing STB. Interlevel 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, modeled 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 substate 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 the set of transitions that will fire is a maximal set of enabled, non-conflicting transitions, 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 first step of our approach is a purely syntactical one and consists in translating the statechart diagrams into what is usually called a hierarchical automaton. Hierarchical Automata can be seen as an abstract syntax for statechart diagrams 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. Our sample statechart diagram is mapped into the hierarchical automaton of Fig. 2. It is easy to see that the hierarchical automaton of Fig. 2 can be taken as an alternative representation for the statechart of Fig. 1. In fact there is a clear correspondence between the states of the two structures. Also the refinement of a state into one or more substates in the statechart is properly represented by the refinement function \( \rho \); in our example we have \( \rho(S) = \{TV, USER\} \), \( \rho(OFF) = \{POWER\} \), \( \rho(ON) = \{IMAGE, SOUND\} \) 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-interlevel transitions are represented in the obvious way. Consider now the interlevel transition from \( STB \) to \( ON \) in Fig. 1. Such a transition is represented in the hierarchical automaton by the transition from \( OFF \) (the highest ancestor of \( STB \) "crossed" by the transition in the statechart) to \( ON \), labeled by \( on \). The indication of the fact that the real "origin" of such a transition is state \( STB \) is encoded in the label of the transition (not shown in the figure). In particular, it is encoded in what is called the source restriction (SR) of the transition. The source restriction of transition \( on \) 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 of a transition contains the so called target determinator (TD). 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 labeled by \( off1 \), the target determinator of which is \{\( STB \)\}. Similarly, the TD of the transition labeled by \( on \) is \{\( SHW, SND \)\}. The complete information related to the transition labels for the hierarchical automaton of Fig. 2 is given by Table 1.

In the sequel we will be concerned only with Hierarchical Automata since the translation from statechart diagrams to Hierarchical Automata is conceptually simple and purely syntactical [11].

As stated in Sect. 1, the semantic model we use in this paper is a slight variant of the one defined in [11]. This model is defined for a quite restricted subset of UML Statechart Diagrams, still including 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. More specifically, we do not consider history, action and activity states; we restrict events to signal and call 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 above restrictions are made essentially for simplicity since, in our opinion, most of them do not have any strong impact on the semantics and translation at a conceptual level.

Other limitations, namely the fact that we do not deal with the object-oriented features of UML statechart diagrams, e.g. sub-behaviours, etc, are more serious and we leave them for further study, together with extensions like deterministic/stochastic time. Basic formal semantics and related tools, even for a restricted language, are an essential step for any further extension with the above mentioned features.

3. Hierarchical Automata

In this section we recall the notion of Hierarchical Automata as defined in [12, 11] and their UML operational semantics given in [11]. Only the relevant definitions are given. We refer to [11] for details. The first notion is that of (sequential) automaton.

---

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. \( f(a_1, a_2, \ldots, a_n) \) will be used instead of \( f(a_1, a_2, \ldots, a_n) \) and function
events range i.e. by just assigning them arbitrary unique names. For sequential automata with mutually disjoint sets of states, \( \lambda_A \) is a finite set of transition labels and \( \delta_A \subseteq \sigma_A \times \lambda_A \times \sigma_A \) is the transition relation.

As mentioned in the previous section, the labels in \( \lambda_A \) have a particular structure. Moreover, we assume that all transition labels are unique. This can be easily be achieved by just assigning them arbitrary unique names. For sequential automaton \( A \) let functions \( SRC, TGT : \delta_A \rightarrow \sigma_A \) be defined as \( SRC(s, l, s') = s \) and \( TGT(s, l, s') = s' \). Hierarchical Automata are defined as follows:

**Def. 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.

**Def. 2 (Hierarchical Automata)** A hierarchical automaton \( H \) is a 4-tuple \((F, E, \rho, \Lambda)\), 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 \( A_{\text{root}} \not\in \bigcup \rho \), (ii) every non-root automaton has exactly one ancestor state: \( \bigcup \rho \cap F \subseteq \{A_{\text{root}}\} \) and \( \forall A \in F \setminus \{A_{\text{root}}\}, \exists s \in \bigcup_{A_{\text{parent}} \in F \setminus \{A_{\text{root}}\}} \sigma_{A_{\text{parent}} iron} A \in (\rho s) \) (iii) there are no cycles: \( \forall S \subseteq \bigcup_{A \in F} \sigma_A, \exists s \in S, s \cap \bigcup_{A_{\text{parent}} \in F} \sigma_A = \emptyset \). Finally, \( \Lambda = \bigcup_{A \in F} \lambda_A \).

We say that a state \( s \) for which \( \rho s = \emptyset \) holds is a basic state. With reference to the hierarchical automaton presented in application will be considered left-associative; (ii) for function \( f : X \rightarrow Y \) and \( Z \subseteq X, f Z = \{y \in Y \mid \exists x \in Z, y = fx\}, \rho \text{ng} f \) denotes the range of \( f \) and \( f|_Z \) is the restriction of \( f \) to \( Z \).

![Example of a Hierarchical Automaton](image)

**Figure 2. Example of a Hierarchical Automaton**

**Table 1. Transition Labels**

<table>
<thead>
<tr>
<th>t</th>
<th>off1</th>
<th>off2</th>
<th>on</th>
<th>d</th>
<th>st</th>
<th>v</th>
<th>sh</th>
<th>m</th>
<th>sn</th>
<th>u0</th>
<th>u1</th>
<th>u2</th>
<th>u3</th>
<th>u4</th>
<th>u5</th>
<th>u6</th>
<th>u7</th>
</tr>
</thead>
<tbody>
<tr>
<td>EV t</td>
<td>off</td>
<td>out</td>
<td>on</td>
<td>out</td>
<td>in</td>
<td>txt</td>
<td>txt</td>
<td>mtc</td>
<td>sn</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>SR t</td>
<td>@</td>
<td>@</td>
<td>{STB}</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>TD t</td>
<td>{STB}</td>
<td>{DIS}</td>
<td>{SHW, SND}</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>AC t</td>
<td>( \epsilon )</td>
<td>( \epsilon )</td>
<td>( \epsilon )</td>
<td>( \epsilon )</td>
<td>( \epsilon )</td>
<td>( \epsilon )</td>
<td>( \epsilon )</td>
<td>( \epsilon )</td>
<td>( \epsilon )</td>
<td>( \epsilon )</td>
<td>( \epsilon )</td>
<td>( \epsilon )</td>
<td>( \epsilon )</td>
<td>( \epsilon )</td>
<td>( \epsilon )</td>
<td>( \epsilon )</td>
<td>( \epsilon )</td>
</tr>
</tbody>
</table>

Our example for the hierarchical automaton in Fig. 2 we have: \( F = \{TV\_SYS, TV, USER, POWER, IMAGE, SOUND\}, \rho S = \{TV, USER\}, \rho OFF = \{POWER\}, \rho ON = \{IMAGE, SOUND\}, \) and all other states are basic.

In the sequel we shall implicitly make reference to a generic hierarchical automaton \( H = (F, \rho, \Lambda) \).

Every sequential automaton \( A \in F \) characterizes a hierarchical automaton in its turn: intuitively, such a hierarchical automaton 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 \( \rho \).

**Def. 3** For \( A \in F \) the automata, states, labels and transitions under \( A \) are defined respectively as \( A A = \{A\} \cup \bigcup_{A' \in F} (\sigma_{A'}) \),

\( S A = \bigcup_{A' \in A} \sigma_{A'} \),

\( \Lambda A = \bigcup_{A' \in A} \lambda_{A'} \), and

\( \mathcal{T} A = \bigcup_{A' \in A} \delta_{A'} \).

The definition of sub-hierarchical automaton follows:

**Def. 4** (Sub-Hierarchical Automata) For \( A \in F \) (\( F_A, E_A, \rho_A, \Lambda_A \)), where \( F_A = (A A), \lambda_A = (\Lambda A), \) and \( \rho_A = \rho|_{(S A)} \), is the hierarchical automaton 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 Hierarchical Automata.
Def. 5 (State Precedence) For \( s, s' \in S \) and \( s \prec s' \) iff \( s' \in \text{closure}(s) \). Let also \( \preceq \) denote the reflexive closure of \( \prec \).

The notion of conflict between transitions needs to be extended in order to deal with state hierarchy. The interested reader is referred to [11]. When transitions \( t \) and \( t' \) are in conflict we write \( t \not\prec t' \). Priorities are assigned to transitions via a function \( \pi \) and a partial order \( \subseteq \) based on state precedence is defined on them. So, we say that \( t \) has lower priority than (the same priority as) \( t' \) iff \( \pi t \subseteq \pi t' \). A configuration denotes a global state of a hierarchical automaton, composed of local states of component sequential automata:

Def. 6 (Configurations) A configuration of \( H \) is a set \( C \subseteq (S, H) \) such that (i) \( \exists s \in \sigma_{A,e} \cdot s \in C \) and (ii) \( \forall s, A. s \in C \land A \in \rho s \Rightarrow 3_s s' \in A. s' \subseteq C \)

For \( A \in F \) the set of all configurations of \( A \) is denoted by \( \text{Conf}_A \). The operational semantics of an hierarchical automaton is defined as a Labeled Transition System, which is a set of states related by a transition relation. In the context of Statecharts Diagrams, states are called statuses and the transition relation is called the STEP relation. The STEP-transitions are labeled by the set of the labels of those transitions of the sequential automata which have been fired in the hierarchical automaton. Each status is composed of a configuration and the current environment with which the hierarchical automaton is supposed to interact. While in classical statecharts the environment is modeled by a set, in the definition of UML statecharts diagrams 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 any particular semantics such as a set, or a bag or a FIFO queue etc., but to model the environment in a policy-independent way. In the following definition we assume that for set \( F \) of transitions via a function \( \pi \) and a partial order \( \subseteq \) based on state precedence is defined on them. So, we say that \( t \) has lower priority than (the same priority as) \( t' \) iff \( \pi t \subseteq \pi t' \). A configuration denotes a global state of a hierarchical automaton, composed of local states of component sequential automata:

Def. 7 (Operational semantics of Hierarchical Automata) The operational semantics of an hierarchical automaton \( H \) is the LTS \( Ts = (S, s^0, \text{Act}, \rightarrow) \) where (i) \( S = \text{Conf}_H \times (\Theta \ E) \) is the set of statuses of \( Ts \), (ii) \( s^0 = (C_0, E_0) \in S \) is the initial status, (iii) \( \text{Act} \subseteq 2^\Lambda \) is the set of actions of \( Ts \) and (iv) \( \rightarrow \subseteq S \times \text{Act} \times S \) is the transition relation defined in the sequel.

A transition of \( Ts \) 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 hierarchical automaton, and no external manipulation is allowed on it [11]. The rule follows:

Def. 8 (Closed Systems)

\[
(H \uparrow \emptyset : (C, \{e\}) \xrightarrow{\text{Sel } e \in E} (C', E'))
\]

In the above rule we make use of an auxiliary relation, namely \( A \uparrow P : (C, E) \xrightarrow{\text{L}} (C', E') \). Such a relation models labeled transitions of the hierarchical automaton \( A \), and \( L \) is the set containing the transitions of the sequential automata of \( A \) which are selected to fire. We call \( \xrightarrow{\text{L}} \) the step transition relation in order to avoid confusion with transitions of sequential automata. \( P \) 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, \( A \uparrow P : (C, E) \xrightarrow{\text{L}} (C', E') \) should be read as “\( A \), on status \((C, E)\), can perform \( L \) moving to status \((C', E')\), when required to perform transitions with priorities not smaller than any in \( P \)”.

Def. 9 (Enabled Transitions) For \( A \in F \), set of states \( C \) and environment \( E \),

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 \(- \in E \) for all \( E \) (including the empty environment) to hold.

The following definition characterizes the operational semantics of a Hierarchical Automaton as a Labeled Transition System (LTS).
(i) the set of all the enabled local transitions of $A$ in $(C, E)$, $LE_A C E$ is defined as:

$$LE_A C E = \{ t \in \delta_A | \{(SRC t) \cup (SR t) \subseteq C \land (EV t) \in E \land (C, E) \vdash (G t)\}$$

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

$$E_A C E = \bigcup_{A' \in (A A)} LE_A C E$$

Moreover, $A \uparrow P :: (C, E)$ will stand for: there exists $C'$ such that $A \uparrow P :: (C, E) \rightarrow (C', E')$. Finally, for state $s$ and set $S \subseteq S (\rho s)$, such that $s \leq s''$ for all $s'' \in S$, the closure of $S (s S)$, is defined as the set $\{s' | \exists s'' \in S. s \leq s' \leq s''\}$.

**Progress rule**

$$t \in LE_A C E \rightarrow b t' \in P \cup E_A C E. \pi t \sqsubseteq \pi t'$$

$$A \uparrow P :: (C, E) \xrightarrow{\sigma} (c (TGT t) (TD t), \text{new}(AC t))$$

**Composition Rule**

$$\{s\} = C \cap \sigma_A$$

$$\rho A s = \{A_1, \ldots, A_n\} \neq \emptyset$$

$$\bigwedge_{j=1}^n A_j \uparrow P \cup LE_A C E :: (C, E) \xrightarrow{L} (C_j, E_j)$$

$$\bigcup_{j=1}^n L_j = \emptyset \Rightarrow (\forall t \in LE_A C E. \exists t' \in P. \pi t \sqsubseteq \pi t')$$

$$A \uparrow P :: (C, E) \xrightarrow{\bigcup_{j=1}^n L_j} \{s\} \cup \bigcup_{j=1}^n C_j, \text{join}_{j=1}^n E_j$$

**Stuttering rule**

$$\{s\} = C \cap \sigma_A$$

$$\rho A s = \emptyset$$

$$\forall t \in LE_A C E. \exists t' \in P. \pi t \sqsubseteq \pi t'$$

$$A \uparrow P :: (C, E) \xrightarrow{\sigma} (\{s\}, \text{nil})$$

Figure 3. Operational Semantics of UML-Hierarchical Automata.

In the operational 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 $A$ 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 hierarchical automaton characterized by $TV$ for off f2. As a result, $TV$ enters configuration $\{OFF, DIS\}$. Similarly, the Progress Rule can be applied to USER for firing transition, say, u5. Using the Composition Rule we then deduce that starting from the status $\{(S, ON, U, SHW, MTE), \{out\}\}$ $TV\_SYS$ produces a step-transition labeled by $\{of, f2, u5\}$ while moving to configuration $\{S, OFF, DIS, U\}$ and delivering mte to the environment.

The following result [11] shows that our operational semantics satisfies the requirements informally defined in [9].

**Theorem 1** For all $L \subseteq (T A), A \uparrow P :: (C, E) \xrightarrow{L}$ 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. $\forall t, t' \in L. t \neq t'$; (ii) all transitions in $L$ are enabled in the current status, i.e. $L \subseteq E_A C E$; (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. $\forall t \in L. \exists t' \in P. \pi t \sqsubseteq \pi t'$; and (iv) all transitions in $L$ respect $P$, i.e. $\forall t \in L. \exists t' \in P. \pi t \sqsubseteq \pi t'$.

4. Model-checking Statecharts Diagrams

4.1. JACK and ACTL

JACK [1] 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 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 Labeled 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. ACTL can be used to describe temporal properties and model checking can be performed, by using AMC, to check whether systems (i.e. their models) satisfy the properties.

ACTL [5] 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 LTS’s. The logic can be used to define both liveness (something good eventually happens) and safety (nothing bad can happen) properties of reactive systems. An account of the syntax and informal semantics of ACTL is given in Table 2.

As an example of ACTL formula expressing a property of our sample system, consider the following:

\[ A G E F < \]

The formula stipulates 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. The above 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 major role in the validation of, among others, dependable systems.

The main advantage in using JACK stems from its independence from the particular notation used for modeling 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 in implementing such a 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.

### 4.2. Building the semantics automaton

In the following an algorithm for building the semantics automaton of a statechart diagram is given. The algorithm uses the operational semantics defined in Sect. 3.

The description of the algorithm is rather informal, given in a kind of pseudo-pascal notation, where, on the other hand, set notation is freely used.

The input to the algorithm is a hierarchical automaton \( H \) and its initial state \( (C_0, E_0) \). It uses relations \( A \uparrow P :: (C, \{ e \}) \leftrightarrow (C', E') \) and \( (S e \in E) \), defined in Sect. 3, as boolean functions. Also function \( \text{join} \) on event queues is used. The termination condition is reached when there are no more (new) states to analyze. At this point, the output of the algorithm is given by the values of variables \( S_t \) and \( T_r \) holding respectively the set of states and the set of step-transitions of the semantics automaton.

The application of the algorithm to our 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 Fig. 5. Essentially, we abstract from the videotext functionality and we replace the user with a less liberal behaviour. Now the user is supposed to issue an \textit{on} command before

```plaintext
Types
HA (* hierarchical automata *)
State (* states of seq. automata *)
Transition (* transition (labels) of seq. automata *)
Event (* events *)
Queue = queue_of Event (* no particular policy fixed *)
Init = set_of State
Status = Config \times Queue
Act = set_of Transition
S-Transition = Status \times Act \times Status (* STEP-transitions *)
Variables
\( H, HA \)
S : set_of Status (* currently generated statuses *)
St : set_of Status (* currently analyzed statuses *)
Tr : set_of S-Transition (* currently generated transitions *)
C, C' : Config
E, E' : Queue
\( e : Event \)
\( \text{Initialization} \)
\( H := \ldots (* \text{the input hierarchical automaton} *); \)
S := \{(C_0, E_0)\};
St := \emptyset;
Tr := \emptyset;
repeat
begin
select any element from \( S \) and assign it to \( (C, E) \);
S := S \setminus \{(C, E)\};
St := St \cup \{(C, E)\};
for \( (e, E') \) in \( \{(x, Y) | (Sel E x Y)\} \) do
if \( H \uparrow \emptyset :: (C, \{ e \}) \leftrightarrow (C', E') \)
then begin
S := S \cup \{(C', [join E', E']\) \setminus St\};
Tr := Tr \cup \{(C, E), (C', join E', E')\}\}
end
until S = \emptyset

Figure 4. Semantics automaton algorithm
```

---

#### Table 2: Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HA</td>
<td>Hierarchical automata</td>
</tr>
<tr>
<td>State</td>
<td>States of sequential automata</td>
</tr>
<tr>
<td>Transition</td>
<td>Transition (labels) of seq. automata</td>
</tr>
<tr>
<td>Event</td>
<td>Events</td>
</tr>
<tr>
<td>Queue</td>
<td>Queue of Event</td>
</tr>
<tr>
<td>Init</td>
<td>Set of State</td>
</tr>
<tr>
<td>Status</td>
<td>Set of States</td>
</tr>
<tr>
<td>Act</td>
<td>Set of Transition</td>
</tr>
<tr>
<td>S-Transition</td>
<td>Status \times Act \times Status (* STEP-transitions *)</td>
</tr>
</tbody>
</table>

#### Figure 4: Semantics automaton algorithm
Action formulas

\[ \chi ::= \begin{cases} 
true & \text{“any observable action”} \\
false & \text{“no observable action”} \\
a & \text{“the observable action } a \text{”} \\
\sim \chi & \text{“any observable action different from } \chi \text{”} \\
\chi | \chi' & \text{“either } \chi \text{ or } \chi' \text{”} \\
\chi \& \chi' & \text{“both } \chi \text{ and } \chi' \text{”} 
\end{cases} \]

State formulas

\[ \phi ::= \begin{cases} 
true & \text{“any behaviour is possible”} \\
false & \text{“no behaviour is possible”} \\
\sim \phi & \text{“} \phi \text{ is impossible”} \\
\phi \& \phi' & \text{“} \phi \text{ and } \phi' \text{”} \\
E \gamma & \text{“there exists a possible execution in which } \gamma \text{”} \\
A \gamma & \text{“for each of the the possible executions } \gamma \text{”} \\
< \chi > \phi & \text{“there exists a next state reachable by an action that satisfies } \chi \text{, in which } \phi \text{ holds”} \\
[\chi] \phi & \text{“for all next states reachable with actions that satisfy } \chi \text{, } \phi \text{ holds”} 
\end{cases} \]

Path formulas

\[ \gamma ::= [\phi(\chi) \cup (\chi') \phi'] \text{ “actions that satisfy } \chi \text{ are performed and } \phi \text{ holds until an action that satisfies } \chi' \text{ has been performed and then } \phi' \text{ holds”} \]

\[ [\phi(\chi) \cup \phi'] \text{ “actions that satisfy } \chi \text{ are performed and } \phi \text{ holds until } \phi' \text{ holds”} \]

\[ G \phi \text{ “in every future state } \phi \text{ holds”} \]

\[ F \phi \text{ “there exists a future state in which } \phi \text{ holds”} \]

Satisfaction relation for action formulas

\[ a \models b \iff a = b \text{ (a and b actions)} \]

\[ a \models \sim \chi \iff a \not\models \chi \text{ action formula} \]

\[ a \models \chi | \chi' \iff a \models \chi \text{ or } a \models \chi' \]

\[ a \models \chi \& \chi' \iff a \models \chi \text{ and } a \models \chi' \]

Table 2. ACTL Syntax and informal Semantics

issuing snd, mte and of f ones. Similarly, after having issued an on signal, (s)he cannot request to power the TV set off (out) before having switched it off (of f). Transition u0 of the original user is split into transitions u0I and u0A, both labeled by (−, 0, 0, ε) in the obvious way. The automaton obtained from the hierarchical automaton of Fig. 5 is shown in Fig. 6. For readability reasons, in the figure statuses are labeled 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.

4.3. Implementing semantics automata

Since we are using a LTS semantics we can easily represent the semantics automata as FC2 objects. The only issue
which deserves some explanation is the relation between the labels of the transitions in the sequential automata, which we shall call *sequential labels* in the sequel, and those labeling the step-transitions in the semantics automata, i.e. those of the LTS. From a theoretical point of view, the latter are just sets over the former. In practice, they are represented in FC2 as words (lists) over the former. For instance, the step-transition label \[s_t, u_1] \] is implemented as the list \["s_t", "u_1"] composed of the two elements "s_t" and "u_1".

On the other hand, a specifier should not be supposed to know what are the transitions which are fired together, but should be allowed to make reference to the sequential labels, like "s_t" or "u_1", since, in the end, these are the objects of the specification (s)he wrote. For instance, in our example, the specifier/verifier might be concerned with proving that the TV set \[USER\] can perform an \[off\] action, i.e. transition \[u_3\], only after an \[on\] action, i.e. \[u_4\], has been performed, and this regardless of the fact that transition \[u_3\] is fired together with \[s_n\] or any other transition of the \[TV_SET\].

In other words, the verifier should be allowed to write the following formula: 
\[
\sim E[true] \& u_4 \cup \{u_3\}\text{true} \& AG([u_3] \sim E[true] \& u_4 \cup \{u_3\}\text{true})
\]

To that purpose, the labels of the LTS are interpreted as logical conjunctions of sequential labels. These last labels, whenever mentioned in a temporal logic formula, are translated into proper disjunctions on LTS labels, under the above interpretation. More precisely, a sequential label is translated into the disjunction of all the LTS labels in which it occurs as an element. This information is fully available once the LTS is computed. For instance, an occurrence of \[u_3\] in a formula is translated as \[u_3 \lor s_n, u_3\]. Notice that the

\[3\] Strictly speaking, the specifier will make reference to the Statechart Diagram and its elements and not to the hierarchical automaton which represents it. So, another translation step is indeed needed, which is performed at the user front-end level, as mentioned above.
labels in the above disjunct are labels of the LTS transitions.

As an example, let’s reconsider the formula \( A \bigwedge E \bigcirc F < out > true \) discussed in section 4.1. The equivalent formula to be considered with reference to the hierarchical automaton of Fig. 5, is \( A \bigwedge E \bigcirc F < of f 2 | d > true \). This formula can then automatically be translated into \( A \bigwedge E \bigcirc F < d, u0 | d, u1 | d, u2 | d, u4 > true \) making reference to the corresponding LTS in Fig. 6. This last formula is in the right form to be directly used for model checking by the AMC model checker of JACK on the model of Fig. 6.

5. Conclusions

In this paper a simple model-checking approach to the formal verification of UML Statechart Diagrams based on the JACK verification environment has been presented. The approach is based on branching time temporal logics and on a formal semantics of UML Statechart Diagrams described in [11]. To our knowledge, this is the first work addressing branching time model-checking of UML Statechart Diagrams.

The work presented in this paper will be 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 UML Statechart Diagrams.

The next step in this work will be the representation of Hierarchical Automata as networks of automata which cooperate for simulating the state hierarchy as well as transition priorities. This way, the efficient model-checking algorithms JACK provides for networks of automata will be available also for Hierarchical Automata.

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.

6. Acknowledgements

The work described in this paper has been performed in the context of the ESPRIT Project n. 27439 - HIDE. The authors thank Istvan Majzik for the fruitful discussions on the UML and its semantics.

References