Under consideration for publication in Formal Aspects of Computing

Automatic Verification of a behavioural subset of UML Statechart Diagrams using the SPIN model-checker[1]

Diego Latella[1], Istvan Majzik[2] and Mieke Massink[1]

[1]Consiglio Nazionale delle Ricerche
Area della Ricerca di Pisa, Istituto CNUCE
Via Alfieri 1, Loc. Ghezzano, I56010 S. Giuliano T. (PI), ITALY
[2]Technical University of Budapest
Dept. of Measurement and Information Systems
Pazmany st. 1/d, H1521 Budapest, HUNGARY

Keywords: UML Statechart Diagrams; Model-checking; Program Transformation; PROMELA; SPIN

Abstract. Statechart Diagrams provide a graphical notation for describing dynamic aspects of system behaviour within the Unified Modelling Language (UML). In this paper we present a translation from a subset of UML Statechart Diagrams - covering essential aspects of both concurrent behaviour, like sequentialisation, parallelism, non-determinism and priority, and state refinement - into PROMELA, the specification language of the SPIN model checker. SPIN is one of the most advanced analysis and verification tools available nowadays. Our translation allows for the automatic verification of UML Statechart Diagrams. The translation is simple, proven correct, and promising in terms of state space representation efficiency.

1. Introduction and related work

The Unified Modelling Language (UML) is a graphical modelling language for object-oriented software and systems [FS97, Rat97a, Rat97b, RJB99]. It has been specifically designed for visualising, specifying, constructing and documenting

[1] The work described in this paper has been performed in the context of the ESPRIT Project n. 27439 - HIDE

Correspondence and offprint requests to: D. Latella, Diego.Latella@cnuce.cnr.it

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 describe a translation from UML Statechart Diagrams into PROMELA, the specification language of the SPIN model checker [Hol91, Hol97].

SPIN is one of the most advanced analysis and verification tools available nowadays, and an automatic translation from UML Statechart Diagrams to PROMELA allows the UML model designer to automatically verify correctness properties of UML Statechart Diagrams specifications.

The UML is a semi-formal language, since its syntax and static semantics (the model elements, their interconnection and well-formed-ness) are defined precisely, but its dynamic semantics are not specified formally [Rat97b]. Several approaches have been proposed in the literature for the definition of a formal semantics of UML Statechart Diagrams, e.g. [WB98, BW97, LMM99, LP99], and much more has been done for classical statecharts.

To the best of our knowledge, transition priorities are dealt with neither in [WB98], where also state refinement is not allowed, nor in [BW97], where model checking is addressed. Both transition priorities and state refinement constitute main issues in our work.

In [LMM99] we proposed an operational semantics for a subset of UML Statechart Diagrams covering state refinement, several transition priorities schemas, including the UML specific one, and essential aspects of concurrent behaviour, e.g. sequentialisation, parallelism and non-determinism. The approach followed in [LMM99] is similar to that proposed in [MLS97] for classical statecharts but it takes into consideration the peculiarities of the UML Statechart Diagrams relevant for the considered subset of the notation. On the other hand, it shares the relative simplicity of the work proposed in [MLS97].

In [LP99] all interesting aspects of UML Statechart Diagrams semantics are covered. Unfortunately, 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 UML Statechart Diagrams. In [LP99] a "flat" representation of UML Statechart Diagrams is used and the authors claim that such a representation is better suited for model-checking purposes than the hierarchical one used in [LMM99]. As we shall show in the present paper, using a hierarchical representation for UML Statechart Diagrams (syntax), not only has no negative impact on tools development, but, rather, it helps very much in carrying on correctness proofs; all interesting results presented in [LMM99] and in the present paper 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.

The work described in the present paper is based on the operational semantics we proposed in [LMM99]. Such a semantical model is defined for a restricted subset of UML Statechart Diagrams, still including all the interesting conceptual issues related to concurrency in the dynamic behaviour, like sequentialisation, non-determinism and parallelism. It also covers state refinement.

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 events, 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 behavioural aspects of the semantics and translation at a conceptual level, but dealing with them would dramatically and uselessly complicate the notation involved.

Other limitations, namely the fact that we do not deal with the object-oriented features of UML statechart diagrams, e.g. sub-behaviours, are more serious and we leave them for further study, together with extensions like the incorporation of deterministic/stochastic time. A basic formal semantics model and related tools, even for a restricted language, are an essential step for any further extension with the above mentioned features. The definition of a sound "basic" subset or kernel of a notation, on which to stress novel semantics concepts as well as develop mathematical theories, like behavioural preorders or equivalences, and experiments with specific tools, like model-checkers, has already proven a valuable, safe and fruitful methodology and is now quite standard practice in many fields of concurrency theory, like process-algebra. Once concepts, theories and tools have been developed for a restricted notation their extension to other, important, features of the notation can be supported by a safer background. For instance, it is our opinion that a sound formal semantics for UML Statechart Diagrams is a necessary condition for extending the considered notation with the inclusion of object-oriented features like classes and subclasses. In fact 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 [BD99].

In designing our translation from UML Statechart Diagrams to PROMELA, we followed an approach similar to that of Mikk et al. [MLSH97]. Nevertheless, our work differs substantially from theirs. First of all, in [MLSH97] classical statecharts [Har87, Har96] are considered instead of UML statecharts. So, the complications induced by the UML transition priorities and their reverse relation with the hierarchical structure of the statechart are not present therein, whereas they are dealt with in our work. More specifically, we use a notion of *priority schema*, on which the semantics are *parametric*, and then we instantiate it with the UML specific one. A high degree of flexibility is thus achieved in a very simple way. Moreover, since the UML definition of the external environment is, in our opinion, only partially defined, our semantics definition as well as our translation are *parametric* with respect to the environment. In the above mentioned work of Mikk et al., the environment is instead represented as a set, as required by the classical statechart semantics. Moreover, due to some simple optimisations, the code generated by our translator is considerably simpler than that of the translation proposed in [MLSH97]. Also, we do not need to use pre- and post-variables, so that the code generated by our translator does not suffer from the memory duplication problem which in the above mentioned work requires specific optimisation techniques. Finally, we are not aware of any correctness result for the work presented in [MLSH97].

The present paper is organised 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 [LMM99] are reviewed in Sect. 3. The actual translation from Hierarchical Automata to PROMELA is defined and proved correct in Sect. 4, where also some

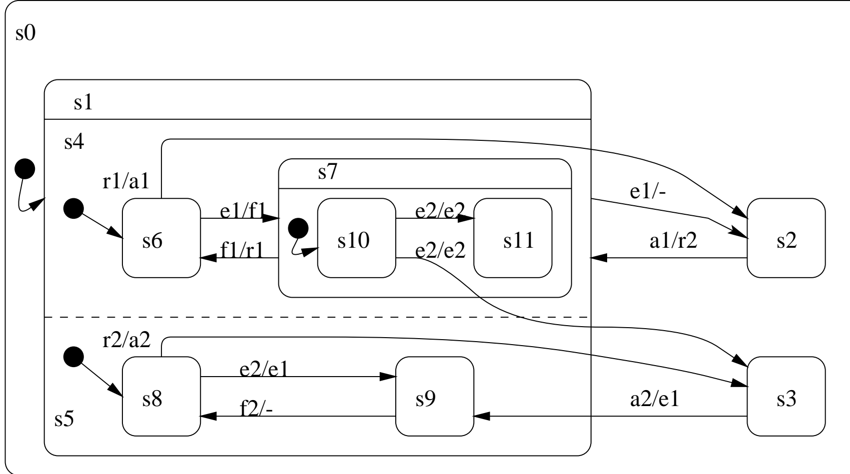

Fig. 1. Example of an UML statechart

simple but effective optimisations are discussed and some results of a case-study are presented. Conclusions are drawn in Sect. 5.

2. Basic concepts

UML Statechart Diagrams are a (object-oriented) variant of classical Harel statecharts [Har87, Har96]. 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. The detailed description of UML Statechart Diagrams can be found in [Rat97a] and [Rat97b].

One of the main notions of statecharts is that of state refinement. In Fig. 1 a UML Statechart Diagram is shown where state $s0$ is refined into an automaton consisting of three states, $s1$, $s2$, and $s3$. State $s1$ is further refined into two states, namely $s4$ and $s5$, each of them refined in turn into a distinct automaton. The same applies to state $s7$. States like $s0$, $s1$, $s4$, $s5$ and $s7$ are called *composite* and in particular $s1$ is a said to be *concurrent*.

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 an (output) event.

"System states" are modelled by *configurations*, which are sets of states. For instance, our system can be in any of the following configurations[2]: $\{s1, s6, s8\}$, $\{s1, s6, s9\}, \{s1, s10, s8\}, \{s1, s11, s8\}, \{s1, s10, s9\}, \{s1, s11, s9\}, \{s2\}, \{s3\}$.

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.

[2] For simplicity, in the following we will often skip mentioning state $s0, s4, s5$ and $s7$ explicitly.

In our example, if event $a1$ is given as input to the machine and the current configuration is $\{s2\}$, state $s2$ is left, event $r2$ is generated and delivered to the environment and state $s1$ is entered. In particular, $s1$ 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 $s4$ and $s5$, namely $s6$ and $s8$.

In the general case, some target sub-states can be explicitly specified. In our example, when the current configuration is $\{s3\}$ and event $a2$ is offered, the configuration resulting from firing the transition labelled by $a2/e1$ will be $\{s1, s6, s9\}$, where $s9$ 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 $s6$ to $s2$ and those from $s8$ to $s3$ and from $s10$ to $s3$ are inter-level ones. Firing, say, the first one requires the system to be in a configuration containing $s6$, regardless of the state in which $s5$ resides, and brings it to state $s2$. 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. In general, 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, in UML Statechart Diagrams, a transition has higher priority than another transition if its source state is a sub-state of the source of the other one. For instance, if the statechart of Fig. 1 is in a configuration containing both $s1$ and $s6$, and the event selected by the dispatcher is $e1$ then the transition from $s6$ to $s7$ will be fired since it has higher priority than the one to $s2$. If the conflict cannot be resolved using priorities, then any of the conflicting enabled transitions may be fired; this happens for instance when $s10$ is in the current configuration and $e2$ is offered by the environment.

Due to concurrent states, it is possible that more than a single transition is fired as a reaction to a given event. 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. In this sense the UML semantics does not allow "chain reactions" within the same step: events generated as a consequence of firing a step are not available to the machine during the same step, but they are available for being dispatched to the machine only from the next step on.

The first phase of our translation 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 automa-

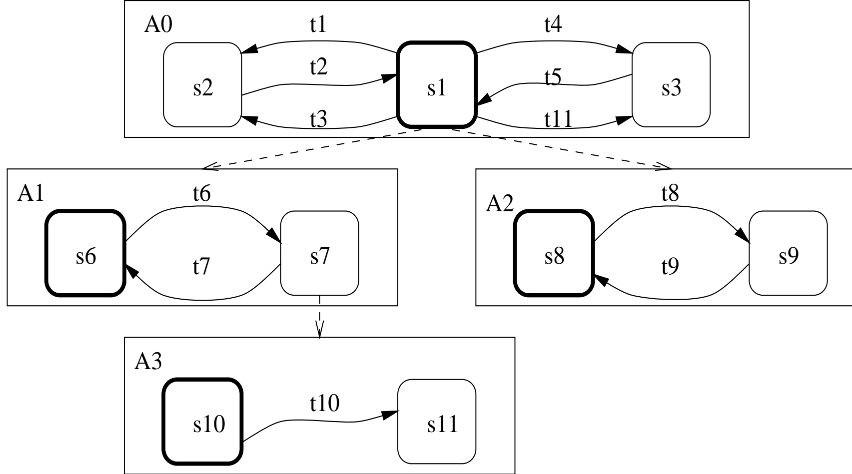

Fig. 2. Example of a Hierarchical Automaton

ton 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. Initial states are denoted by thick boxes. The refinement of a state into one or more sub-states in the statechart is properly represented by the refinement function ρ; in our example we have $\rho\ s1 = \{A1, A2\}$, $\rho\ s7 = \{A3\}$ and $\rho\ s = \emptyset$ for any other state s. In the figure this is represented by dashed arrows.

Non-inter-level transitions are represented in the obvious way. Consider now the inter-level transition from $s6$ to $s2$ in Fig. 1. Such a transition is represented in the hierarchical automaton by the transition from $s1$ (the highest ancestor of $s6$ "crossed" by the transition in the statechart) to $s2$, named $t1$. The indication of the fact that the real "origin" of such a transition is state $s6$ 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 $t1$ is $s6$. 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. Furthermore, 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 $s3$ to $s9$ in Fig. 1 is represented in Fig. 2 by the transition labelled $t5$, the target determinator of which is $\{s6, s9\}$. Finally, the label may contain an optional *guard* (G) which must evaluate to *true* in order for the transition to be enabled. Missing guards evaluate to *true* by default. From the above informal discussion it should be clear that all kinds of inter-level transitions across compound states in UML Statechart Diagrams have an adequate and clean representation in hierarchical automata.

The complete information related to the transition labels for the hierarchical automaton of Fig. 2 is given by Table 1, where guards are not shown since they are not used in the example.

In the sequel we will be concerned only with hierarchical automata since the

Table 1. Transition Labels

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

translation from statechart diagrams to hierarchical automata is conceptually simple and purely syntactical [LMM99].

3. Hierarchical Automata

In this section we review the notion of hierarchical automata as defined in [MLS97, LMM99] and their UML operational semantics given in [LMM99]. Only the relevant definitions are given. We refer to [LMM99] for details. The first notion is that of a (sequential) automaton[3].

Definition 3.1 (Sequential Automata). A sequential automaton A is a 4-tuple $(\sigma_A, s^0_A, \lambda_A, \delta_A)$ where σ_A is a finite set of *states* with $s^0_A \in \sigma_A$ the *initial state*, λ_A is a finite set of *transition labels* 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 λ_A have a particular structure. For transition t we shall require its label to be a 5-tuple (sr, ev, g, ac, td), where sr is the *source restriction*, ev is the *trigger* event, g is the *guard*, ac is the *actions* list and td is the *target determinator*.

In the sequel we shall use the following functions SRC, TGT, SR, EV, G, AC, TD, defined in the obvious way; for transition $t = (s, (sr, ev, g, ac, td), s')$, $SRC\ t = s, TGT\ t = s', SR\ t = sr, EV\ t = ev, G\ t = g, AC\ t = ac, TD\ t = td$.

Hierarchical Automata are defined as follows:

Definition 3.2 (Hierarchical Automata). A hierarchical automaton H is a 4-tuple (F, E, ρ, Λ), 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* $\rho : \bigcup_{A \in F} \sigma_A \longrightarrow 2^F$ imposes a tree structure to F, i.e. (i) there exists a unique root automaton $A_{root} \in F$ such that $A_{root} \notin \bigcup rng\ \rho$, (ii) every non-root automaton has exactly one ancestor state: $\bigcup rng\ \rho = F \setminus \{A_{root}\}$ and $\forall A \in F \setminus \{A_{root}\}.\ \exists_1 s \in \bigcup_{A' \in F \setminus \{A\}} \sigma_{A'}.\ A \in (\rho\ s)$ and (iii) there are no cycles: $\forall S \subseteq \bigcup_{A \in F} \sigma_A.\ \exists s \in S.\ S \cap \bigcup_{A \in \rho\, s} \sigma_A = \emptyset$; 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 Fig. 2 we have: $F = \{A0, A1, A2, A3\}$, $A0$ is the root automaton, $\rho\ s1 = \{A1, A2\}$, $\rho\ s7 = \{A3\}$, and all other states are

[3] In the following we will 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 application will be considered left-associative; (ii) for function $f : X \longrightarrow Y$ and $Z \subseteq X$, $f\ Z = \{y \in Y \mid \exists x \in Z.\ y = fx\}$, $rng\ f$ denotes the *range* of f and $f_{|Z}$ is the restriction of f to Z. Moreover the notation $\exists_1 x.\ P$ stands for "There exists a unique x such that P."

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

Every sequential automaton $A \in F$ characterises 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 ρ_A which is a proper restriction of ρ. A is the root automaton.

Definition 3.3. For $A \in F$ the *automata*, *states*, *labels* and *transitions under* A are defined respectively as

- $\mathcal{A}\ A = \{A\} \cup (\bigcup_{A' \in \left(\bigcup_{s \in \sigma_A} (\rho_A s)\right)} (\mathcal{A}\ A'))$
- $\mathcal{S}\ A = \bigcup_{A' \in \mathcal{A}\ A} \sigma_{A'}$
- $\Lambda\ A = \bigcup_{A' \in \mathcal{A}\ A} \lambda_{A'}$
- $\mathcal{T}\ A = \bigcup_{A' \in \mathcal{A}\ A} \delta_{A'}$

The definition of sub-hierarchical automaton follows:

Definition 3.4 (Sub-Hierarchical Automata). For $A \in F$, $(F_A, E, \rho_A, \Lambda_A)$, where $F_A = (\mathcal{A}\ A)$, $\Lambda_A = (\Lambda\ A)$, and $\rho_A = \rho_{|(\mathcal{S}\ A)}$, is the hierarchical automaton characterised 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 characterises, the role being clear from the context. H will be identified with A_{root}. Sequential Automata will be considered a degenerate case of Hierarchical Automata.

In the following we shall define the notions of *conflicting* transitions, *transition priority* and *orthogonal states*. For a more detailed discussion on their properties the reader is referred to [LMM98, LMM99]. Both the notion of conflict and that of priority are based on the notion of state precedence:

Definition 3.5 (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 3.6 (Conflicting Transitions). For $t, t' \in (\mathcal{T}\ H)$, t is conflicting with t', written $t\#t'$, iff $t \neq t'$ and $(SRC\ t \preceq SRC\ t') \vee (SRC\ t' \preceq SRC\ t)$

The following definition characterises those structures which can be used to impose priorities on transitions.

Definition 3.7 (Priority Schema). A *Priority Schema* is a triple $(\Pi, \sqsubseteq, \pi)$ with $(\Pi, \sqsubseteq)$ a *partial order* and $\pi : (\mathcal{T}\ H) \to \Pi$ such that: $\forall t, t' \in (\mathcal{T}\ H).\ (\pi\ t \sqsubseteq \pi\ t') \wedge t \neq t' \Rightarrow t\#t'$ 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 generalises the requirement that transitions originating from "inner" states have priority over those higher in the state hierarchy as required in UML Statechart Diagrams [Rat97b]. To that purpose we first need the notion of *orthogonal states*.

Definition 3.8 (Orthogonal States). Two states $s, s' \in \mathcal{S}\ H$ are *orthogonal*, written $s\ ||\ s'$, iff $\exists s'' \in (\mathcal{S}\ H), A, A' \in (\rho\ s'').\ A \neq A' \wedge s \in \mathcal{S}\ A \wedge s' \in \mathcal{S}\ A'$

Let $(PWO, \preceq^s, f)$ such that:

- $PWO = \{S \subseteq (\mathcal{S}\ H) \mid \forall s, s' \in S.\ (s \neq s' \Rightarrow s \mid\mid s')\}$ is the set of all the sets of Pair-Wise Orthogonal states of H.
- For all $S, S' \subseteq \mathcal{S}\ H$, $S \preceq^s S'$ iff $\forall s \in S.\ \exists s' \in S'.\ s \preceq s'$; $\preceq^s$ is a lifting of $\preceq$ to sets.
- $f\ t = \{s \mid s \in (SRC\ t) \wedge (SR\ t) = \emptyset\} \cup (SR\ t)$ is the priority function assigning priority to transitions.

It can be easily shown that $(PWO, \preceq^s, f)$ is a priority schema and that, for such a schema, transitions from "inner" states have higher priority than those originating from states "higher" in the state hierarchy.

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

Definition 3.9 (Configurations). A *configuration* of H is a set $C \subseteq (\mathcal{S}\ H)$ such that (i) $\exists_1 s \in \sigma_{A_{root}}.\ s \in C$ and (ii) $\forall s, A.\ s \in C \wedge A \in \rho\ s \Rightarrow \exists_1 s' \in A.\ s' \in C$.

For $A \in F$ the set of all configurations of A is denoted by Conf_A.

The operational semantics of a hierarchical automaton is defined as a Kripke Structure, which is a set of states related by a transition relation. In the context of Statechart Diagrams, states are called *statuses* and the transition relation is called the *STEP relation*. 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 modelled by a set, in the definition of UML statechart 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 multi-set or a FIFO queue etc., but to model the environment in a policy-independent way. In the following definition we assume that for set X, ΘX denotes the set of all structures of a certain kind (like FIFO queues, or multi-sets, or sets) over X and we assume to have basic operations for inserting and removing elements from such structures. In particular $(add\ \mathcal{E}\ e)$ denotes the structure obtained by adding e to environment $\mathcal{E}$. Similarly, $(join\ \mathcal{E}\ \mathcal{E}')$ denotes the environment obtained by merging $\mathcal{E}$ with $\mathcal{E}'$. The predicate $is_join_{j=1}^{n} \mathcal{E}_j\ \mathcal{J}$ states that $\mathcal{J}$ is a possible *join* of $\mathcal{E}_1 \ldots \mathcal{E}_n$ and it is a way for expressing non-deterministic merge of $\mathcal{E}_1 \ldots \mathcal{E}_n$. Moreover, by $(Sel\ \mathcal{E}\ e\ \mathcal{E}')$ 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, *nil* is the empty structure and given sequence $r \in X^*$, $(new\ r)$ is the structure containing the elements of r (again, the existence and nature of any relation among the elements of $(new\ r)$ depends on the semantics of the particular structure). So, for instance, if sets are chosen, then $(add\ \mathcal{E}\ e) = \mathcal{E} \cup \{e\}$, $(join\ \mathcal{E}\ \mathcal{E}') = \mathcal{E} \cup \mathcal{E}'$ and, for $e \in \mathcal{E}$, $(Sel\ \mathcal{E}\ e\ \mathcal{E}') \equiv (\mathcal{E}' = \mathcal{E} \setminus \{e\})$.

The semantics of UML does not specify what happens when a queue is empty. Our approach in this paper is to make the system block in such a case: this is in line with the behaviour of PROMELA processes when attempting to receive from an empty queue.

Definition 3.10 (Operational semantics of Hierarchical Automata). The operational semantics of an hierarchical automaton H is a Kripke structure $\mathbf{k} = (\mathbf{S}, \mathbf{s^0}, \longrightarrow)$ where (i) $\mathbf{S} = \mathrm{Conf}_H \times (\Theta\ E)$ is the set of statuses of $\mathbf{k}$, (ii) $\mathbf{s^0} = (\mathcal{C}_0, \mathcal{E}_0) \in \mathbf{S}$ is the initial status, (iii) $\longrightarrow \subseteq \mathbf{S} \times \mathbf{S}$ is the transition relation defined in the sequel.

A transition of $\mathbf{k}$ denotes a maximal set of non-conflicting transitions of the sequential automata of H which respect priorities. The $\longrightarrow$ 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 [LMM99]. The rule follows:

Definition 3.11 (Closed Systems).

$$\frac{\begin{array}{ll}(Sel\ \mathcal{E}\ e\ \mathcal{E}'') & (1)\\ H \uparrow \emptyset :: (\mathcal{C}, \{e\}) \xrightarrow{L} (\mathcal{C}', \mathcal{E}') & (2)\end{array}}{(\mathcal{C}, \mathcal{E}) \longrightarrow (\mathcal{C}', (join\ \mathcal{E}''\ \mathcal{E}'))}$$

In the above rule we make use of an auxiliary relation, namely $A \uparrow P :: (\mathcal{C}, \mathcal{E}) \xrightarrow{L} (\mathcal{C}', \mathcal{E}')$. Such a relation models labelled 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 $\xrightarrow{L}$ 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, $A \uparrow P :: (\mathcal{C}, \mathcal{E}) \xrightarrow{L} (\mathcal{C}', \mathcal{E}')$ 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 Def. 3.11, but set P will be used to record the transitions a certain automaton can fire when considering its sub-automata. More specifically, for a sequential automaton A, P will accumulate (the priority information of) all transitions which are enabled in the ancestors of A. The deduction system for $\xrightarrow{L}$ is shown in Fig. 3 where the following auxiliary functions are used:

Definition 3.12 (Enabled Transitions). For $A \in F$, set of states $\mathcal{C}$ and environment $\mathcal{E}$,
(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 follows[4]:

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

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

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

[4] $(\mathcal{C}, \mathcal{E}) \models g$ means that guard g is true of status $(\mathcal{C}, \mathcal{E})$. Its formalisation is immaterial for the purposes of the present paper. In the deduction rules, we will relax the requirement $\mathcal{C} \in \mathrm{Conf}_A$ and we will assume $\mathcal{C} \in \mathrm{Conf}_H$. This allows the use of guards which make reference to non local states.

Progress Rule

$$\frac{\begin{array}{ll} t \in LE_A\ \mathcal{C}\ \mathcal{E} & (1) \\ \not\exists t' \in P \cup E_A\ \mathcal{C}\ \mathcal{E}.\ \pi\ t \sqsubset \pi\ t' & (2) \end{array}}{A \uparrow P :: (\mathcal{C}, \mathcal{E}) \xrightarrow{\{t\}} (\mathbf{c}\ (TGT\ t)\ (TD\ t), new(ACt))}$$

Composition Rule

$$\frac{\begin{array}{ll} \{s\} = \mathcal{C} \cap \sigma_A & (1) \\ \rho_A\ s = \{A_1, \ldots, A_n\} \neq \emptyset & (2) \\ \left(\bigwedge_{j=1}^{n} A_j \uparrow (P \cup LE_A\ \mathcal{C}\ \mathcal{E}) :: (\mathcal{C}, \mathcal{E}) \xrightarrow{L_j} (\mathcal{C}_j, \mathcal{E}_j)\right) \wedge is_join_{j=1}^{n} \mathcal{E}_j\ \mathcal{J} & (3) \\ \left(\bigcup_{j=1}^{n} L_j = \emptyset\right) \Rightarrow (\forall t \in LE_A\ \mathcal{C}\ \mathcal{E}.\ \exists t' \in P.\ \pi t \sqsubset \pi t') & (4) \end{array}}{A \uparrow P :: (\mathcal{C}, \mathcal{E}) \xrightarrow{\bigcup_{j=1}^{n} L_j} (\{s\} \cup \bigcup_{j=1}^{n} \mathcal{C}_j, \mathcal{J})}$$

Stuttering Rule

$$\frac{\begin{array}{ll} \{s\} = \mathcal{C} \cap \sigma_A & (1) \\ \rho_A\ s = \emptyset & (2) \\ \forall t \in LE_A\ \mathcal{C}\ \mathcal{E}.\ \exists t' \in P.\ \pi t \sqsubset \pi t' & (3) \end{array}}{A \uparrow P :: (\mathcal{C}, \mathcal{E}) \xrightarrow{\emptyset} (\{s\}, nil)}$$

Fig. 3. Operational Semantics of UML-Hierarchical Automata.

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

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.

The following result [LMM98, LMM99] shows that our operational semantics satisfies the requirements informally defined in [Rat97b].

Theorem 3.1. 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. $\forall t \in L.\ \not\exists t' \in E_A\ \mathcal{C}\ \mathcal{E}.\ \pi t \sqsubset \pi t'$; and (iv) all transitions in L respect P, i.e. $\forall t \in L.\ \not\exists t' \in P.\ \pi t \sqsubset \pi t'$.

4. From Hierarchical Automata to PROMELA

In this section we describe the translation from Hierarchical Automata to PROMELA, the specification language for the "on-the-fly" model checker SPIN [Hol91, Hol97]. The choice of SPIN is mainly motivated by the variety of highly efficient state-space representation techniques and search algorithms it provides. Moreover, PROMELA provides standard data types and constructs which make models described in such a language understandable also to the practitioner designers. Finally, several translators from design and implementation languages to PROMELA have been proposed in the literature, so that validation at different development phases can now be supported by SPIN.

PROMELA is a process language, based on the interleaving model of computation. It is a C-like language extended with non-deterministic and loop guarded-constructs in the style of Dijkstra. Obviously, constructs to run processes and for inter-process communication are provided as well. Communication takes place via buffered, finite length, channels. A PROMELA specification is usually called a "PROMELA model".

Using PROMELA, one can define both a model describing the behaviour of a system, and a so called *never claim*. The never claim is a Büchi automaton which is run in synchronous product with the model and which is used for checking both safety and liveness properties. SPIN provides an automatic facility for generating Büchi automata from formulæ of a linear time temporal logic. For example, the formula $[] <> p$ where p is defined as $S2 == 1$ states that variable $S2$ will evaluate to 1 infinitely often. $[]$ and $<>$ are the SPIN notation for the temporal logics operators "forever" and "eventually"; SPIN offers also an "until" operator, besides the standard logic operators. If variable $S2$ models state $s2$ and in particular the boolean expression $S2 == 1$ models the fact that our sample hierarchical automaton is in a configuration which contains state $s2$, the above formula can be used for checking that such a state must be reached infinitely often, which is indeed the case in our example.

SPIN can also be used to simulate PROMELA models, i.e. to execute them, possibly with the controlled interference of the user, animating them via a useful graphic interface.

Finally, other verification means, like assertions to be verified in specific points of executions as well as invariants are provided by SPIN.

We do not discuss the details of PROMELA here. We shall just recall the main features of PROMELA constructs when we use them. For a detailed account of the language and the system we refer to the literature [Hol91, Hol97].

4.1. Base translation

The translation function takes an hierarchical automaton $H = (F, E, \rho, \Lambda)$ as input and generates PROMELA code as output. The translation is based on the operational semantics described in Sect. 3. The definition of these semantics is recursive in nature and obviously reflects the tree-like structure of hierarchical automata. This poses a first problem in our translation due to the very limited recursion capabilities of PROMELA. The solution we chose is not to use recursion at all at the PROMELA level and to limit its use in the translation function definition in order to exploit the tree-structure of the hierarchical automaton whenever possible.

In hierarchical automata, non-determinism arises both from conflicting transitions of the same sequential automaton and from the possibility of applying the Progress Rule or the Composition Rule. In the translation, non-determinism is modelled using the PROMELA non-deterministic construct, as we shall see later.

In the following we give a semi-formal definition of the translation. The reason why we present it in a semi-formal way is readability: we prefer to use a "dot notation", abstracting from PROMELA syntax details, rather than fully formalise them. A complete formalisation in ML can be found in [GL98], which, being ML executable, also provides an executable prototype translator.

4.1.1. Modelling the events and environment

In this work events are treated as uninterpreted symbols. We represent them as integer values and we name them for convenience. The related code is trivial, the `#define` directive of PROMELA being the same as in C. For set of events $E = \{e_1, \ldots, e_n\}$ we get the following PROMELA constants definition fragment:

#`define` e_1 1
⋮
#`define` e_n n

As already discussed in Sect. 3, the UML-semantics of statechart diagrams do not specify the semantics of the queue. For this reason, we leave the specifier free to choose the preferred structure from the following possibilities: a set, a multi-set, and a FIFO queue. The choice among these alternatives is an input parameter for the translator, which will generate the appropriate PROMELA code accordingly. Sets and multi-sets will be represented by their characteristic functions, while a FIFO queue is directly mapped into a PROMELA channel and it is up to the designer to specify its length LT. More specifically, for set E as above, if the environment is modelled by a set, such a set will be represented a boolean (bit) array of length n. For efficiency reasons, such an array is actually represented by n one-bit variables[5]:

`bit` $Qe_1, \ldots,\ Qe_n$;

with the obvious meaning: $Qe_j == 1$ if and only if event e_j is currently available in the environment. Analogously, a multi-set is represented by an integer array of length n.

A FIFO queue will be represented by generating the following code:

`chan` $Q = [LT]$ `of` $\{$`int`$\}$

As we shall see later, the choice of a FIFO queue has some impact also on the code generated for the actual firing of the transitions.

Finally, the selected event will be stored in the integer variable `Ev`, which obviously models e of Def. 3.11:

[5] This is because in SPIN bit arrays are implemented as byte arrays.

4.1.2. Modelling configurations

States are modelled in a straightforward way, namely by using a single bit variable per state. A configuration will be composed by those states corresponding to variables evaluating to 1. Let $\mathcal{S}\,H = \{s_1, \ldots\ s_z\}$; then the code to be generated for representing states is simply:

`bit` $S_1, \ldots\ S_z$;

where variable S_j represents state s_j via a bijection the definition of which is easy so that we leave it out here.

The code fragment generated for our example according to the above rules follows[6]:

```
#define a1 1
#define a2 2
#define e1 3
#define e2 4
#define f1 5
#define f2 6
#define r1 7
#define r2 8

bit Qa1, Qa2, Qe1, Qe2, Qf1, Qf2, Qr1, Qr2;

bit S1, S2, S3, S6, S7, S8, S9, S10, S11;
```

4.1.3. Modelling the STEP

Steps are generated by the *STEP* PROMELA process. The general structure of the *STEP* process consists of four phases:

1. selection of an event from the environment;
2. identification of the candidate transitions to fire - includes identification of enabled transitions and conflict resolution based on transition priority;
3. selection of those transitions, from the above ones, which will be fired - includes parallelism management and conflict resolution based on non determinism;
4. actual firing of the selected transitions - includes the identification of the resulting configuration and the generation of resulting events.

The generation of successive STEPs is obtained by including within a loop the code modelling the single STEP. Notice that atomicity of each step is guaranteed by means of the PROMELA **atomic** command: this in particular means that the only values available for verification are those which variables evaluate to *at the end* of each cycle. Any intermediate value taken *during* a cycle is hidden to the never claim as well as possible other monitoring processes which may run in parallel with the STEP process.

[6] The reader can see that here we decided to model the queue by a set. This is for simplicity reasons.

In the following we describe the four phases. As we show later on, the last three phases can be collapsed into a single one. Here we present them separated since this way the relationship with the operational semantics is more immediate.

The code to be generated for modelling the **selection of an event** from the queue depends on the choice of the designer w.r.t. modelling the queue. If the queue is modelled by a set, then the following code `select_event_set(H)` is generated:

`select_event_set(H)` $\stackrel{\text{def}}{=}$
`if`
$:: Qe_1 -> \ Ev \ = \ e_1; \ Qe_1 = 0$
$\vdots$
$:: Qe_n -> \ Ev \ = \ e_n; \ Qe_n = 0$
`fi`

The PROMELA semantics of the above non-deterministic construct is that the execution is blocked if none of the guards $Qe_1, \ldots, Qe_n$ evaluates to 1, otherwise the statements corresponding to any of those evaluating to 1 is executed, after which the `if` statement is left and control is passed to the next statement. By "corresponding" here we mean those statements contained between the arrow $->$ following the guard and the next $::$ or `fi` keyword. PROMELA provides also a special guard `else` which is satisfied if and only if all the others of the same `if`...`fi` statement evaluate to 0.

In the case of a multi-set, the test will obviously be $Qe_i >= 1$ and a decrement $Qe_i --$ will be used. If instead the designer chose to model the queue by a channel, the following input command $Q?Ev$ is generated.

The core of the translation lies in the identification of those transitions which are candidates for being fired and in selecting among them those which will actually be fired.

For each transition $t_j \in (\mathcal{T} \ H) = \{t_1, \ldots, t_v\}$ we use a variable $Cand_j$ to which a boolean expression will be assigned which evaluates to 1 if and only if t_j, is a **candidate for being fired**. Suppose $t_j \in \delta_A$, with $A \in F$. Such an assignment corresponds to implementing the premises of the Progress Rule, with P properly set, on the basis of the enabling situation of the transitions in the ancestors of A. We represent set P by its characteristic function:

`bit` $P_1, \ldots, P_v$

It is worth now recalling the premises of the Progress Rule where for notational convenience we have renamed t into t_j and we have split condition (2) into two distinct premises:

$$\begin{array}{ll} t_j \in LE_A \ \mathcal{C} \ \mathcal{E} & (1) \\ \nexists t' \in P. \ \pi \ t_j \sqsubset \pi \ t' & (2.1) \\ \nexists t' \in E_A \ \mathcal{C} \ \mathcal{E}. \ \pi \ t_j \sqsubset \pi \ t' & (2.2) \end{array}$$

Premise (1) is the enabling condition for t_j. Letting $SRC \ t_j = s_{j1}$, $SR \ t_j = \{s_{j2}, \ldots, s_{jk}\}$, $EV \ t_j = ev_j$ and $G \ t_j = g_j$, such a condition can be coded as the following PROMELA expression, where & denotes logical conjunction:

$S_{j1} \ \& \ \ldots \ \& \ S_{jk} \ \& \ (Ev == ev_j) \ \& \ g_j$

Notice that all the information on $SRC\ t_j$, $SR\ t_j$, $EV\ t_j$ and $G\ t_j$ is available to the translator since it is static information.

Let us now assume that, for generic transition t_x, P_x evaluates to 1 if and only if the enabling condition for t_x holds - this is achieved by an assignment similar to the above one, but relative to t_x. Let furthermore $\{t_{j1}, \dots, t_{js}\}$ be the set of all those transitions t_{ji} such that $t_{ji} \in (\mathcal{T}\ H) \setminus (\mathcal{T}\ A)$ and $\pi\ t_j \sqsubset \pi t_{ji}$; again, notice that $\pi\ t_j \sqsubset \pi t_{ji}$ is static information. Then, premise (2.1) is coded as follows:

$!P_{j1}\ \&\ \dots\ \&\ !P_{js}$

where ! denotes logical negation in PROMELA.
Finally, let $\{t_{j1}, \dots, t_{jm}\}$ be the set of those transitions t_{ji} such that:

- $SRC\ t_{ji} = SRC\ t_j$ or $SRC\ t_{ji} \in \mathcal{S}\ (\rho(SRC\ t_j))$, and
- $\pi\ t_j \sqsubset \pi t_{ji}$

It is easy to see that premise (2.2) corresponds to the conjunction of the *negations* of the enabling conditions for such transitions t_{ji}:

$S_{ji1}\ \&\ \dots\ \&\ S_{jik_i}\ \&\ (Ev == ev_{ji})\ \&\ g_{ji}$

where, as before, we have: $SRC\ t_{ji} = s_{ji1}$, $SR\ t_{ji} = \{s_{ji2}, \dots, s_{jik_i}\}$ and $EV\ t_{ji} = ev_{ji}$ and $G\ t_{ji} = g_{ji}$.

This brings us to the complete fragment of PROMELA code relative to the assignment to $Cand_j$:

$\underline{\texttt{candidate_designate}(t_j)} \stackrel{\text{def}}{=}$
$Cand_j = S_{j1}\ \&\ \dots\ \&\ S_{jk}\ \&\ (Ev == ev_j)\ \&\ g_j\ \&$
$!P_{j1}\ \&\ \dots\ \&\ !P_{js}\ \&$
$!(S_{j11}\ \&\ \dots\ \&\ S_{j1k_1}\ \&\ (Ev == ev_{j1})\ \&\ g_{j1})\ \&$
$\vdots$
$!(S_{jm1}\ \&\ \dots\ \&\ S_{jmk_m}\ \&\ (Ev == ev_{jm})\ \&\ g_{jm})$

As a concrete example of the above description the assignments for `Cand1`, `Cand3` and `Cand7` are given below[7]:

```
Cand1 = S1 & S6 & (Ev==r1)

Cand3 = S1 & (Ev==e1)             &
!(S1 & S6 & (Ev==r1) )            &     /* relative to t1 */
!(S1 & S8 & (Ev==r2) )            &     /* relative to t4 */
!(S6 & (Ev==e1) )                 &     /* relative to t6 */
!(S7 & (Ev==f1) )                 &     /* relative to t7 */
!(S8 & (Ev==e2) )                 &     /* relative to t8 */
!(S9 & (Ev==f2) )                 &     /* relative to t9 */
!(S10 & (Ev==e2) )                &     /* relative to t10 */
!(S1 & S10 & (Ev==e2) )                 /* relative to t11 */
```

[7] We recall here that $\pi\ t3 \sqsubset \pi\ t1, \pi\ t3 \sqsubset \pi\ t4, \pi\ t3 \sqsubset \pi\ t6, \pi\ t3 \sqsubset \pi\ t7, \pi\ t3 \sqsubset \pi\ t8, \pi\ t3 \sqsubset \pi\ t9, \pi\ t3 \sqsubset \pi\ t10, \pi\ t3 \sqsubset \pi\ t11, \pi\ t7 \sqsubset \pi\ t10, \pi\ t7 \sqsubset \pi\ t11$ and $\pi\ t \sqsubset \pi\ t'$ does not hold otherwise.

```
Cand7 = S7 & (Ev==f1)            &
!P11                             &
!(S10 & (Ev==e2) )               &      /* relative to t10 */
```

The complete code for the candidate transitions designation amounts to the sequencing of candidate_designate(t_j) for $j = 1, \ldots, v$.

In order to complete the description of the candidates designation step, we have to describe how variables P_j are set. In the operational semantics, set P is initialised to $\emptyset$ by Def. 3.11 and it is then "pumped up" with the enabled transitions of the automata visited when the step relation is deduced for the children of the refined states of such automata, via the Composition Rule. This results in P containing all and only the enabled transitions of the *ancestors* of $A \in F$ when applying the Progress Rule (or the Stuttering Rule) to it, i.e. when testing the enabling condition for an arbitrary transition t of A. Notice moreover that the only transitions of interest, among those in P, are those with a higher priority than t, as it can be seen from (2.1) of the Progress Rule.

In the PROMELA coding of premise (2.1), on the other hand, we consider set $\{t_{j1}, \ldots, t_{js}\}$ which indeed can only contain transitions of ancestors of A. In fact it cannot contain transitions of A or descendants of A (they would be elements of $\mathcal{T}\ A$), but also it cannot contain transitions of automata which run in parallel with A or an ancestor of A (these transitions cannot have higher priority than t's since such priorities are unrelated in the priority partial order, see [LMM99] Lemma 4). So, we just need to set P_{ji} to the enabling condition for t_{ji}, for $t_{ji} \in \{t_{j1}, \ldots, t_{js}\}$ as above. Moreover, since the value of P_{ji} depends only on the values of the state variables $S_1, \ldots\ S_z$ and variable Ev and since these values are not changed during the computation of the candidate transitions, the above assignments can be safely done for *all* the transitions in $\mathcal{T}\ H$ at the beginning of the cycle. More specifically, at the beginning of the cycle, just after variable Ev has been assigned a new value in the select_event(H) code, the set_P(t_j) code below is generated in sequence for $j = 1, \ldots, v$:

set_P(t_j) $\stackrel{\text{def}}{=}$
$P_j = S_{j1}\ \&\ \ldots\ \&\ S_{jk}\ \&\ (Ev == ev_j)\ \&\ g_j$

In our current example we have:

```
P1= S1 & S6 & (Ev==r1); P2= S2 & (Ev==a1); P3= S1 & (Ev==e1);
P4= S1 & S8 & (Ev==r2); P5= S3 & (Ev==a2); P6= S6 & (Ev==e1);
P7= S7 & (Ev==f1);      P8= S8 & (Ev==e2); P9= S9 & (Ev==f2);
P10=S10& (Ev==e2);      P11=S1 & S10& (Ev==e2)
```

This concludes the description of the candidate designation phase. Notice that this phase computes the set of *all* transitions which are enabled and which respect the priority constraints. Among these transitions, there are some which are in *conflict* and therefore cannot be fired in the same step. The actual **transition selection** phase takes care of resolving conflicts and selecting the transitions which will be eventually fired. This is done by assigning the value 1 to a second set of bit variables for the transitions, Sel_j, and exploits the PROMELA non-deterministic construct where the guard for assignment to Sel_j is obviously

$Cand_j$. The code for such assignments is generated recursively according to the tree structure of the hierarchical automaton.

Let $A \in F$ a hierarchical automaton with $\delta_A = \{t_1, \ldots, t_q\}$. If $\rho_A\ s = \emptyset$ for all $s \in \sigma_A$, i.e. if A is sequential, then the code for the selection phase for A, `select_transitions(A)`, is simply:

`select_transitions(A)` $\stackrel{\text{def}}{=}$
`if`
$:: Cand_1\ -> Sel_1 = 1$
$\vdots$
$:: Cand_q\ -> Sel_q = 1$
:: `else skip`
`fi`

If instead $\bigcup_{X \in \rho_A\ \sigma_A} X = \{A_1, \ldots, A_r\} \neq \emptyset$ then, letting $\delta_{A_j} = \{t_{j1}, \ldots, t_{jq_j}\}$ for $j = 1, \ldots, r$, the code for the selection phase for A, `select_transitions(A)` is recursively the following, where || denotes logical disjunction in PROMELA:

`select_transitions(A)` $\stackrel{\text{def}}{=}$
`if`
$::\quad Cand_1 \quad -> Sel_1 = 1$
$\vdots$
$::\quad Cand_q \quad -> Sel_q = 1$
$::\quad Cand_{11} \mid\mid \ldots \mid\mid Cand_{1q_1} \mid\mid \ldots \mid\mid Cand_{r1} \mid\mid \ldots \mid\mid Cand_{rq_r} ->$
    `select_transitions(A₁);`
    $\vdots$
    `select_transitions(Aᵣ)`
:: `else skip`
`fi`

From the above translation schema it is clear that a selection is made non-deterministically between those transitions which (are enabled, respect priority constraints and) are in conflict, whereas transitions which are not in conflict may have their Sel_j variable set to 1 at the same time, after the execution of the code generated by `select_transitions(A)` (this is due to the *sequencing* of the code `select_transitions(`A_1`)`, ... `select_transitions(`A_r`)`). Stuttering corresponds to the `else skip` branches. The code generated for our example follows:

```
if
::Cand1->Sel1=1
::Cand2->Sel2=1
::Cand3->Sel3=1
::Cand4->Sel4=1
::Cand5->Sel5=1
::Cand11->Sel11=1
::(Cand6||Cand7||Cand8||Cand9||Cand10)->
  if
  ::Cand6->Sel6=1
```

```
  ::Cand7->Sel7=1
  ::Cand10->
    if
    ::Cand10->Sel10
    ::else->skip
    fi
  ::else->skip
  fi;
  if
  ::Cand8->Sel8=1
  ::Cand9->Sel9=1
  ::else->skip
  fi
::else->skip
fi
```

So, if both `Cand6` and `Cand8` evaluate to 1, then *both* $t6$ and $t8$ will be selected for firing.

Finally, the actual **firing** of the selected transitions can occur. The code for firing a single transition t_j is self explanatory; below we for simplicity assume the event queue modeled by a set and an action consisting of generating one single output event:

<u>`fire_transition`</u>$(t_j) \stackrel{\text{def}}{=}$

`if`
$:: Sel_j -> S_{j1} = 0; \ldots; S_{js} = 0; S_{js+1} = 1; \ldots; S_{jh} = 1; Qe_j = 1$
:: `else skip`
`fi`

where $\bigcup_{A \in \rho(SRC\ t_j)} (\mathcal{S}\ A) \cup (SRC\ t_j) = \{S_{j1}, \ldots, S_{js}\}$, $AC\ t_j = e_j$, and $\mathbf{c}\ (TGT\ t_j)\ (TD\ t_j) = \{S_{js+1}, \ldots, S_{jh}\}$; of course the statement $Qe_j = 1$ is generated only if $AC\ t_j$ is nonempty. For instance, for $t1$ of our example we have:

```
if
::Sel1->S1=0;S2=1;S3=0;S6=0;S7=0;S8=0;S9=0;S10=0;S11=0;Qa1=1
::else->skip
fi
```

In the case of an environment represented by a set or a multiset, the order in which transitions are fired is irrelevant since no trace of the event delivery order will be kept in the environment. In this case the code to be generated is an arbitrary sequence of firing statements like the above one. In the case of a FIFO queue, instead, the code must take into consideration that parallel transitions can be fired in any order and then the resulting queues will be different. This can be easily done by running separate parallel PROMELA processes, each executing a statement like the above one or simulating this parallelism with a repetitive command and proper guards.

The last statements before ending a cycle and starting the next one are those for cleaning-up the auxiliary structures: all $P_j, Cand_j$ and Sel_j variables are set to 0.

```
translate(H) =def

#define e_1 1
:
#define e_n n
bit Qe_1, ..., Qe_n;
bit S_1, ... S_z;
int Ev;
bit Cand_1, ..., Cand_v;
bit Sel_1, ..., Sel_v;
bit P_1, ..., P_v;
process STEP()
{
do
:: atomic
{
select_event(H);
set_P(t_1);
:
set_P(t_v);
candidate_designate(t_1);
:
candidate_designate(t_v);
select_transitions(H);
fire_transition(t_1);
:
fire_transition(t_v);
cleanup(H);
}
od
}
init
{ initialize(H); run STEP()
}
```

Fig. 4. Schema of the translation from generic hierarchical automaton H to PROMELA

The PROMELA model is composed of a process, the STEP process, which is essentially a loop performing the above phases. Process STEP is run after all variables have been properly initialized in order to model the initial status. The schema of the translation function, assuming a set for environment is summarized in Fig.4 where the translation functions calls are underlined in order to avoid confusion with actual PROMELA code. Moreover, those (other) pieces of syntax depending on the input hierarchical automaton H are printed in emphasized mode. In the case of a multiset or a FIFO queue the schema is quite similar.

4.2. Correctness of the translation

In this section we shall prove the correctness of our translation function relative to the operational semantics of Sect 3. Although the proofs are rigorous, they are not fully formalized in order to limit notational complexity.

It should be clear to the reader that the key issue is the correctness of the PROMELA code generated for modeling the single step, and in particular the candidate designation and the transitions selection, since the correctness of the code generated for the event selection and for the actual firing of transitions is trivial to establish.

In the following, for hierarchical automaton H we let C_{cd} stand for the sequence of PROMELA statements for candidate designation in the translation of H generated by the following fragment of the translation function:

```
set_P(t_1);
⋮
set_P(t_v);
candidate_designate(t_1);
⋮
candidate_designate(t_v);
```

Similarly, let C_{st} be the code generated by `select_transitions(H)`. Furthermore we let C stand for $C_{cd}; C_{st}$. Before proving the correctness result we need some terminology related to PROMELA semantics. A *state vector* for a PROMELA model is composed by all the variables declared in the model, including the channels, plus a program counter for each process instantiation. A *state* is an assignment of values to the elements of the state vector. An *execution* Σ of a piece of code M starting from a state α is a possibly infinite sequence of states $\alpha_1, \ldots, \alpha_n$ where $\alpha_1 = \alpha$ and α_{i+1} is obtained from α_i by applying the operational semantics rules of PROMELA to the statements corresponding to the values of the program counters in α_i.

In general more than one single execution is allowed for the same code M from the same state α because the PROMELA semantics is based on the interleaving model of computation and the language includes non-deterministic statements.

It is worth noting here that the only source of non-determinism for code C above comes from C_{st} since C_{cd} is completely deterministic so, for given initial conditions, there exist only one execution for the latter. Moreover, it is easy to see that every execution of C always terminates.

We can now prove the following lemmata which, together, establish the correctness of the modeling of the step by the code generated by the translation function.

Lemma 4.1. Let $A \in F$, $P, L \subseteq (\mathcal{T}\ H)$, $\mathcal{C} \in \mathrm{Conf}_H$, $e \in E$, state α such that (i) P contains only enabled transitions of ancestors of A; (ii) for $j = 1, \ldots, z$ S_j evaluates to 1 in α if and only if the corresponding state s_j belongs to $\mathcal{C}$; (iii) Ev evaluates to e in α; (iv) for $j = 1, \ldots, v$ $Cand_j$ and Sel_j evaluate to 0 in α and g_j evaluates to 1 in α if and only if $(\mathcal{C}, \{e\}) \models g_j$. Then $A \uparrow P :: (\mathcal{C}, \{e\}) \xrightarrow{L}$ implies that there exists a finite execution Σ of C starting from α such that in its last state Sel_j evaluates to 1 for all $t_j \in L$.

Proof. By induction on the length d of the derivation for $A \uparrow P :: (\mathcal{C}, \mathcal{E}) \xrightarrow{L}$

$(\mathcal{C}', \mathcal{E}')$ for some $\mathcal{C}', \mathcal{E}'$ [Mil89].

Base case ($d = 1$):
If the derivation has length 1, then only the Progress Rule or the Stuttering Rule could have been applied. Suppose the Progress Rule has been applied. We first notice that for t_j in L $Cand_j$ will evaluate to 1 at the end of the execution of C_{cd}. This follows directly from the definition of `candidate_designate`(t_j), the hypothesis and the fact that premisses (1), (2.1) and (2.2) of the Progress Rule hold. Moreover, due to the semantics of the PROMELA non-deterministic command, there exists an execution of C_{st} at the end of which Sel_j will evaluate to 1. Then there exists an execution of $C_{cd}; C_{st}$ at the end of which Sel_j evaluates to 1. If instead the Stuttering Rule has been applied, then the assertion trivially holds from the fact that there always exists an execution for C under the hypothesis of the lemma.

Induction step ($d > 1$):
In this case the Composition Rule must have been applied in the last step of the derivation. From the Composition Rule we see that each L_i belongs to a step-transition which has been proven by a sub-derivation of length strictly less than d, so, by the Induction Hypothesis, there exist executions $\Sigma_1, \ldots, \Sigma_n$ of C such that, for $i = 1, \ldots, n$ we have that Sel_j evaluates to 1 for all $t_j \in L_i$ at the end of Σ_i. From the definition of `select_transitions(H)`, and in particular from the fact that the code it generates contains the *sequencing* of the transition selection code for each A_i, it follows that there must be an execution Σ of C such that Sel_j evaluates to 1 for each $t_j \in \bigcup_{i=1}^{n} L_i = L$. □

Lemma 4.2. Let state α be such that, for $j = 1, \ldots, v$ $Cand_j = 1$ if and only if $t_j \in \delta_A$, for $A \in F$ is enabled. Then for the end state α' of each execution of C_{st} the set $\{t_j \mid Sel_j \text{ evaluates to 1 in } \alpha'\}$ is a maximal set of non-conflicting transitions.

Proof. The assertion can be proved by induction on the structure of A.

Base case ($\bigcup_{X \in \rho_A \; \sigma_A} X = \emptyset$):
In the base case, namely if A is sequential, the assertion follows directly from the definition of `select_transitions(A)`. In fact from such a definition it follows that Sel_j will evaluate to 1 for just one transition t_j and $\{t_j\}$ is obviously a maximal set for a sequential automaton.

Induction step ($\bigcup_{X \in \rho_A \; \sigma_A} X \neq \emptyset$):
If A is not sequential, letting $\{A_1, \ldots, A_r\} = \bigcup_{X \in \rho_A \; \sigma_A} X$, we have, by the induction hypothesis, that each fragment `select_transitions`(A_i), for $i = 1, \ldots, r$, computes a maximal set and again from the definition of `select_transitions(A)` we get that at the end of an execution for C_{st} either Sel_j evaluates to 1 for a transition of A, or the set $\{t_j \mid Sel_j \text{ evaluates to 1 in } \alpha'\}$ is the union of the sets computed by `select_transitions`(A_i) during the execution, and so it is maximal as well. □

Lemma 4.3. Let $\mathcal{C} \in \mathrm{Conf}_H$, $e \in E$, state α such that (i) for $j = 1, \ldots, z$ S_j evaluates to 1 in α if and only if the corresponding state s_j belongs to $\mathcal{C}$; (ii) Ev evaluates to e in α ; (iii) for $j = 1, \ldots, v$ $Cand_j$ and Sel_j evaluate to 0 in α and and g_j evaluates to 1 in α if and only if $(\mathcal{C}, \{e\}) \models g_j$. Suppose

there exists an execution Σ of C which starts from α and stops in α_n and let $L = \{tj \mid Sel_j \text{ evaluates to 1 in } \alpha_n\}$. Then $H \uparrow \emptyset :: (\mathcal{C}, \{e\}) \xrightarrow{L}$ holds.

Proof. The argument here is simply observing the fact that, under the assumptions of the hypothesis, at the end of the execution of C_{cd}, variables $Cand_j$ characterize the set of *all* enabled transitions which respect priorities. Then the assertion follows from Lemma 4.2 and Theorem 3.1. □

Theorem 4.1. For hierarchical automaton $H, \mathcal{C} \in \text{Conf}_H$, $e \in E$, state α such that (i) for $j = 1, \ldots, z$ S_j evaluates to 1 if and only if the corresponding state s_j belongs to $\mathcal{C}$; (ii) Ev evaluates to e; (iii) for $j = 1, \ldots, v$ $Cand_j$ and Sel_j evaluate to 0 we have that $H \uparrow \emptyset :: (\mathcal{C}, \{e\}) \xrightarrow{L}$ holds if and only if there exists a finite execution Σ of C starting from α and such that in its last state Sel_j evaluates to 1 for all $t_j \in L$.

Proof. The assertion follows directly from Lemmata 4.1 and 4.3 □

4.3. Preliminary optimizations

By taking a look at the code generated following the rules described above, it is clear that variables $Cand_j$ and P_j are just temporary variables whose only purpose is to collect a value which is then immediately used. So they can be eliminated and their use be replaced by the expressions which are assigned to them.

A similar reasoning applies to the Sel_j variables: they can be eliminated and the code corresponding to the actual firing of transition t_j can be moved inside the nested `if...fi` transition selection statement, where Sel_j was set to 1.

On the basis of the above consideration a reduced version of PROMELA code can be generated which is much simpler than that presented in the previous sections and uses fewer variables: the only variables are those representing the configurations and the environment, namely the status.

The result of the optimized translation of our running example is given below where the macros $Cand_j$ are used only for readability purposes and could be eliminated:

```
#define a1 1
#define a2 2
#define e1 3
#define e2 4
#define f1 5
#define f2 6
#define r1 7
#define r2 8

bit S1, S2, S3, S6, S7, S8, S9, S10, S11;
bit Qa1, Qa2, Qe1, Qe2, Qf1, Qf2, Qr1, Qr2;
int Ev;

#define Cand1 (S1 & S6 & (Ev==r1))
#define Cand2 (S2 & (Ev==a1))
#define Cand3 (S1 & (Ev==e1) &     \
               !(S1  & S6 & (Ev==r1) )& \
               !(S1  & S8 & (Ev==r2) )& \
               !(S1  & S10 & (Ev==e2))& \
               !(S6  & (Ev==e1) )& \
```

```
               !(S7  & (Ev==f1) )& \
               !(S8  & (Ev==e2) )& \
               !(S9  & (Ev==f2) )& \
               !(S10 & (Ev==e2)))
#define Cand4 (S1 & S8 & (Ev==r2))
#define Cand5 (S3 & (Ev==a2))
#define Cand6 (S6 & (Ev==e1))
#define Cand7 (S7 & (Ev==f1) & \
               !(S1  & S10 & (Ev==e2)) &\
               !(S10 & (Ev==e2)))
#define Cand8 (S8 & (Ev==e2))
#define Cand9 (S9 & (Ev==f2))
#define Cand10 (S10 & (Ev==e2))
#define Cand11 (S1  & S10 & (Ev==e2))

proctype STEP()
{
do
::
atomic{
if
:: Qa1->Ev=a1;Qa1=0
:: Qa2->Ev=a2;Qa2=0
:: Qe1->Ev=e1;Qe1=0
:: Qe2->Ev=e2;Qe2=0
:: Qf1->Ev=f1;Qf1=0
:: Qf2->Ev=f2;Qf2=0
:: Qr1->Ev=r1;Qr1=0
:: Qr2->Ev=r2;Qr2=0
fi
;
if
::Cand1->S1=0;S2=1;S6=0;S7=0;S8=0;S9=0;S10=0;S11=0;Qa1=1
::Cand2->S1=1;S2=0;S3=0;S6=1;S8=1;Qr2=1
::Cand3->S1=0;S2=1;S6=0;S7=0;S8=0;S9=0;S10=0;S11=0
::Cand4->S1=0;S3=1;S6=0;S7=0;S8=0;S9=0;S10=0;S11=0;Qa2=1
::Cand5->S1=1;S3=0;S6=1;S9=1;Qe1=1
::Cand11->S1=0;S3=1;S6=0;S7=0;S8=0;S9=0;S10=0;S11=0;Qe2=1
::(Cand6||Cand7||Cand8||Cand9||Cand10)->
  if
  ::Cand6->S6=0;S7=1;S10=1;Qf1=1
  ::Cand7->S6=1;S7=0;S10=0;S11=0;Qr1=1
  ::Cand10->
    if
    ::Cand10->S10=0;S11=1;Qe1=1
    ::else->skip
    fi
  ::else->skip
  fi;
  if
  ::Cand8->S8=0;S9=1;Qe1=1
  ::Cand9->S8=1;S9=0
  ::else->skip
  fi
::else->skip
fi
}
od
}

init
```

```
{
atomic{
S1=1;S2=0;S3=0;S6=1;S7=0;S8=1;S9=0;S10=0;S11=0;
Qa1=0;Qa2=0;Qe1=0;Qe2=1;Qf1=0;Qf2=0;Qr1=0;Qr2=0;
};
run STEP()
}
```

Of course, also in this case, if a FIFO queue is used for modeling the environment, then the inner `if...fi` statements must be replaced by parallel processes, or proper code simulating the behaviour of such processes.

It should also be noticed that further standard code optimizations are possible. For instance the `if...fi` statement where only `Cand10` is tested could be replaced by `S10=0;S11=1;Qe1=1`. Moreover, further specific optimizations are possible. Configurations can be represented using basic states only, reducing the number of bit variables necessary for representing states. Such a number can then be further reduced to its base 2 logarithm by using proper encodings. The use of PROMELA **d_step** could help reducing the number of states. A **d_step** takes a list of statements and collapses the sequence of transitions resulting from their execution into a single one, so removing all intermediate states.

In any case, our models, both those generated by the base version and those obtained via the optimized one, result quite efficient in terms of number of states. A hint for understanding why this is the case can be given by analyzing the Kripke Structure of our simple example. It has 7 statuses while the PROMELA models both have 10 states (the base one having also a wider state vector than the reduced one). The presence of the 3 extra states can easily be explained. They are a consequence of the nested **if** statements that are executed "one at a time", thus generating an intermediate state whereas in the Kripke Structure the execution of the transitions of parallel automata (or their stuttering steps) "happen at the same time". This brings to a state-space overhead which grows linearly with the size of the hierarchical automaton. Further study and experimentation is needed in order to assess the efficiency of the code generated by our translation; nevertheless our first experiments are quite encouraging.

4.4. Implementation

Two implementations of the base translation exist: one has been developed in CNUCE and is written in Standard ML [GL98]. The other has been developed at the Technical University of Budapest and uses database technology [BDJS98]. Several experiments have been performed on this latter implementation. Here we briefly review the results of one of these experiments, namely the translation and verification of the Production Cell [LL95]. For a more detailed description of the experiments the reader is referred to [MJ99].

The particular version of the Production Cell used in the experiment is that defined in [BMM99], which, besides the standard components a feed belt, elevating rotary table, robot and press, includes a second (redundant) press and a human operator who is also able to repair the failures of the presses. The model of the system consists of 12 statecharts with a total number of 73 states and 89

transitions[8]. The translator, implemented in PL/SQL, ran on a Sun SPARCstation 20. It took about 180 sec. to generate the PROMELA code, including the time required to interface the front-end UML tool (MID Innovator version 6.1). The verification of the generated PROMELA model was performed using SPIN version 3.2.3 running on Linux PC (Intel Celeron 400MHz with 128Mbytes of RAM). The following results have been obtained for two validations:

```
Safety checking (safety, invalid endstates):
- time required to check: 0.68user 0.16system (seconds)
- length of state vector: 68 byte
- number of states: 1320 states, stored
- depth reached: 154046
- number of transitions: 1373 transitions (stored+matched)
- number of atomic steps: 441329
- required memory: 14.404 Mbyte

Liveness checking (liveness, non-progress cycles, where the progress
  means a processed blank removed from the production cell,
  the state marked by the "progress" label manually)
- time required to check: 0.23user 0.24system
- length of state vector: 72 byte
- number of states: 181 states, stored
- number of transitions: 185
- number of atomic steps: 29866
- required memory: 14.404 Mbyte
- result of the analysis: property failed
  (due to possible cycle of failure-reparation,
  without progress in processing the blank plates)
```

We conclude that the PROMELA model generated by the basic translation for the Production Cell is quite manageable.

5. Conclusions

In this paper we have shown a translation from UML Statechart Diagrams into PROMELA. We have defined a base translation, proved it correct and discussed some, preliminary, optimisations which substantially improve the resulting code.

Two implementations of the base translation exist: one has been developed in CNUCE and is written in Standard ML [GL98]. The other has been developed at the Technical University of Budapest and uses database technology [BDJS98].

Several experiments have been performed on this latter implementation. The interested reader is referred to [MJ99]; here we mentioned the results of only one such experiment, namely a version of the Production Cell [LL95], which, in our opinion, is rather indicative of the feasibility of our approach.

As we have seen in Sect. 4 the PROMELA models generated by the second version of the translator are simpler, shorter, and use fewer variables than those generated via the base one. Their sizes grow linearly with the size of the input hierarchical automaton, in particular with the number of transitions. This is a first improvement which is achieved by means of a few simple and safe optimisations. Further optimisations are possible and they are subject of our future research.

[8] Notice that these numbers refer to the *static* structure of the statechart diagram and not to its execution.

In this paper we considered a subset of UML, which essentially covers the aspects related to concurrency and state refinement. Issues of object-orientation need further, basic research. Other features like variables, history states or structured events, etc. can be included without drastic conceptual changes. For instance variables can be dealt with by adding proper store and name-binding functions to statuses. Completion transitions, which are a basic notion in the context of UML Statechart Diagrams, can be dealt with in a clean and modular way by defining a second deduction system for modelling completion steps. The structure of such a deduction system will be the same as that of the step-transition one (Fig. 3), and the two systems will be very similar in other respects. The final complete step-transition will be modelled by means of two further simple rules which merge step-transitions with completion runs, giving priority to the latter.

We consider the work presented in this paper as an essential first step towards a more complete model-checker for UML Statechart Diagrams.

It is worth pointing out here that our current semantic model and translation do not allow more than one statechart and one queue. On the other hand, it seems that the general idea of the UML designers is to associate a distinct statechart to each class or object and then let such statecharts communicate via queues. We have doubts about the methodological soundness of such an approach. Statechart Diagrams already allow concurrent behaviours to described by means of concurrent states, which act as a kind of "parallel composition operator" in the notation. The approach proposed by the UML designers adds to this an inter-object communication paradigm that uses concepts that lay *outside* the primitive composition operators. This does not follow generally accepted orthogonality principles of language design. In our opinion it would be better to have distinct statecharts modelling the behaviour of different objects within a system glued together into a single, parallel, statechart representing the system. If the interaction model which has been chosen for the interaction with the environment and, via the environment, between sub-machines of the statechart, namely the *queue plus the dispatcher*, turns out not to be expressive enough for modelling sub-machine interaction, then the issue is to find ways to enrich *such* an interaction model in order to make it satisfactory.

Nevertheless, should the UML designer stick to the first approach, it is straightforward to adapt our semantics as well as our translation. As far as the semantics is concerned we just need to add an extra (set of) rule(s) in order to deduce changes in a global status composed by the current configuration of each sub-machine and the current value of each queue, using the deduction system for the step transition we have been using in this paper, after proper addressing/naming conventions have been set and formalised. As far as the translation is concerned, it is almost trivial to change it in such a way that several STEP processes are generated, one per statechart diagram, and such processes communicate via FIFO channels or shared variables representing sets or multi-sets.

Finally, we want to point out that we can use the work presented in this paper as a starting point for the development of translations to enriched models such as timed-automata or stochastic-automata, for enriched versions of Statechart Diagrams with the aim of timed-model-checking or discrete event simulation. Of course, this will make sense only after a formal semantics for those enriched versions of Statechart Diagrams have been defined, which is one of our next areas for further study.

6. Acknowledgements

Istvan Majzik has been partially supported by the Hungarian Scientific Research Fund OTKA-F030533. Mieke Massink has been supported by the TACIT network under the European Union TMR Programme, Contract ERB FMRX CT97 0133. The authors would like to thank the anonymous reviewers for their interesting and constructive comments.

References

[BD99] H. Bowman and J. Derrik. A junction between state based and behavioural based specifications. In P. Ciancarini, A. Fantechi, and R. Gorrieri, editors, *IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Oriented Distributed Systems*. Kluwer Academic Publishers, 1999. ISBN 0-7923-8429-6.

[BDJS98] A. Borschet, M. Dal Cin, J. Javorszky, and C. Szasz. Specification of the HIDE environment. Technical Report HIDE/D3/TUB/1/v2, ESPRIT Project n. 27439 - High-Level Integrated Design Environment for Dependability HIDE, 1998.

[BMM99] A. Bondavalli, I. Majzik, and I. Mura. Automatic dependability analysis for supporting design decisions in UML. In A. Williams, editor, *Fourth IEEE International High-Assurance Systems Engineering Symposium*, pages 64–71. IEEE Computer Society Press, 1999. ISBN 0-7695-0418-3.

[BW97] J. Broersen and R. Wieringa. Interpreting UML-statecharts in a modal μ-calculus. Unpublished manuscript, 1997.

[FS97] M. Fowler and K. Scott. *UML Distilled. Applying the Standard Object Modeling Language*. Addison-Wesley, 1997. ISBN 0-201-32563-2.

[GL98] E. Giusti and D. Latella. Implementazione in SML di un traduttore da automi gerarchici a PROMELA. Technical Report CNUCE-B4-1998-018, Consiglio Nazionale delle Ricerche, Istituto CNUCE, 1998. In italian.

[Har87] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming. Elsevier*, 8(3):231–274, 1987.

[Har96] D. Harel. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996.

[Hol91] G. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991. ISBN0-13-539925-4.

[Hol97] G Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[LL95] C. Lewerentz and T. Lindner, editors. *Formal Development of Reactive Systems - Case Study Production Cell*, volume 891 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[LMM98] D. Latella, I. Majzik, and M. Massink. A simplified formal operational semantics for a subset of UML statechart diagrams. Technical Report HIDE/T1.2/PDCC/5/v1, ESPRIT Project n. 27439 - High-Level Integrated Design Environment for Dependability HIDE, 1998.

[LMM99] D. Latella, I. Majzik, and M. Massink. Towards a formal operational semantics of UML statechart diagrams. In P. Ciancarini, A. Fantechi, and R. Gorrieri, editors, *IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Oriented Distributed Systems*, pages 331–347. Kluwer Academic Publishers, 1999. ISBN 0-7923-8429-6.

[LP99] J. Lilius and I. Paltor Porres. The semantics of UML state machines. Technical Report 273, Turku Centre for Computer Science, 1999.

[Mil89] R. Milner. *Communication and Concurrency*. Series in Computer Science. Prantice Hall, 1989.

[MJ99] I. Majzik and J. Javorszky. Formal verification of UML statecharts: Case studies. Technical Report MITUB-TR-99-05, Dept. of Measurement and Information Systems - Technical University of Budapest, 1999.

[MLS97] E. Mikk, Y. Lakhnech, and M. Siegel. Hierarchical automata as model for statecharts. In R. Shyamasundar and K. Euda, editors, *Third Asian Computing Science*

Conference. Advances in Computing Sience - ASIAN'97, volume 1345 of *Lecture Notes in Computer Science*, pages 181–196. Springer-Verlag, 1997.

[MLSH97] E. Mikk, Y. Lakhnech, M. Siegel, and G. J. Holzmann. Implementing Statecharts in Promela/SPIN. Technical Report BL011272-971203-25TM, Bell Labs, Lucent Technologies, 1997.

[Rat97a] Rational Software and Microsoft and Hewlett-Packard and Oracle and Sterling Software and MCI Systemhouse and Unisys and ICON Computing and IntelliCorp and i-Logix and IBM and ObjecTime and Platinum Technology and Ptech and Taskon and Reich Technologies and Softeam. *UML Notation Guide, version 1.1*, 1997. Notation guide with diagram descriptions.

[Rat97b] Rational Software and Microsoft and Hewlett-Packard and Oracle and Sterling Software and MCI Systemhouse and Unisys and ICON Computing and IntelliCorp and i-Logix and IBM and ObjecTime and Platinum Technology and Ptech and Taskon and Reich Technologies and Softeam. *UML Semantics, version 1.1*, 1997. UML semantics with metamodel.

[RJB99] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999. ISBN 0-201-30998-X.

[WB98] R. Wieringa and J. Broersen. A minimal transition system semantics for lightweight class and behavior diagrams. In M. Broy, D. Coleman, T. Maibaum, and B. Rumpe, editors, *Proceedings of the ICSE98 Workshop on Precise Semantics for Software Modeling techniques*, 1998.