A Formal Semantics of UML StateCharts by Means of Timed Petri Nets

Youcef Hammal

LSI, Département d'Informatique, Faculté d'Electronique & Informatique,
Université des Sciences et de la Technologie Houari Boumediene,
BP 32, El-Alia 16111, Bab-Ezzouar, Algiers, Algeria
yhammal@wissal.dz

Abstract. This paper deals with the formalization of Unified Modeling language (UML) by means of Petri Nets. In order to improve the semantics of UML dynamic diagrams, we define a new method of embedding UML StateCharts into Interval Timed Petri Nets (ITPN). This method considers all kinds of hierarchical states together with the most of pseudo-states like history ones. Besides consistencies analysis, time intervals of ITPN model well event generation and dispatching delays making it possible to achieve performance and time properties analysis of complex systems.

1 Introduction

Complex systems are typically large and reactive systems containing a great number of different kinds of hardware and software components that may operate concurrently by means of various synchronization and communication mechanisms. They are also characterized by very high-level interactions with their environment and reactions to external stimuli must be achieved within given time intervals. Otherwise, violation of those temporal constrains produces critical consequences. Therefore, modeling and analysis of functional requirements are not enough and other concepts like reliability and safety become as important as the previous ones.

Obviously a multi-paradigm modeling language is needed to design and analyze such critical systems. Hence the Unified Modeling language offers a collection of visual, friendly and flexible notations for expressing the artifacts representing various aspects of complex systems ranging from business applications to real time systems.

This OMG standard language [13] is based on object-orientation involving high-quality design concepts such as abstraction, encapsulation and decomposition of systems into objects. Besides, UML is also supported by wide-established tools and environments for specification, design and automatic code generation. Moreover its extensibility faculties make it easier to improve UML notations to address issues of critical systems with respect to the profile for schedulability, performance and time specification [14]. However in spite of the precise syntactic aspects of UML notations, their semantics remain too imprecise and lack verifiability capabilities [6][8]. So considering the semi-formal aspect of UML, it is imperative to define mapping methods of UML diagrams into other formal modeling languages to take

F. Wang (Ed.): FORTE 2005, LNCS 3731, pp. 38 – 52, 2005.

advantage of their tools developed for analysis, simulation and verification of parts of produced models. Thus, the lack of formality can be overcome by providing a dynamic semantics to UML by means of various rigorous mathematical formalisms such as Petri Nets and Formal Specification Techniques (FST).

In this way our approach focuses on ascribing behavioral diagrams of UML (mainly StateCharts) with formal semantics in terms of temporal nets in order to allow both the verification of consistency of the different dynamic diagrams and the analysis of timeliness and performance of an UML model.

In this paper we deal with UML StateCharts that are state machines increasing the modeling power of classical state transition diagrams by introducing superstates and the hierarchical decomposition of superstates. On the other hand, the target formalism is a derived Petri net [1] enhanced with time intervals to represent timing information about event transmission delays. We proceed with our method by steps to overcome the arising difficulties relating to the boundary-crossing arcs that transgress the good compositional properties of StateCharts like those described in [16].

The paper is structured as follows: The next section presents the related work and the motivation of our work. Section 3 shows the basic features of UML StateCharts and in section 4 we present the interval timed Petri nets. Then in section 5, the patterns of our translation method are given for more comprehensibility and section 6 presents the main algorithms which map StateCharts into ITPN formalism. Finally a conclusion is given in section 7 where some remarks and future works are outlined.

2 Motivation and Related Work

The UML specification document [13] provides the description of any UML diagram in three parts: the abstract syntax, the well-formedness rules for the abstract syntax and then its semantics. The well-formedness rules are formulated in the Object Constraint Language OCL and the semantics is given in a natural language.

Thus the architects of UML use the meta-modeling level to describe UML using class diagrams and OCL to capture the static relationship between modeling concepts. However this approach is not adequate since class diagrams and OCL are too little precise to describe the language semantics [6]. Furthermore using natural language or OCL makes it hard to discover inconsistencies among various diagrams of an UML model. Moreover the lack of precise semantics of UML can lead to a number of problems relating to readability and interpretation, use of rigorous design process, rigorous semantic analysis and tool support limited only to syntactic concerns [6], [8].

Hence many proposals are issued giving formal meaning to UML Models in order to overcome the above problems and achieve their analysis by means of verification and validation tools. Among them we distinguish two families of approaches:

The first family includes approaches that improve the meta-model of UML to overcome the ambiguities of standard UML semantics. For instance the authors of [7] add Dynamic Meta Modeling rules for the specification of UML consistency constraints and provide concepts for an automated testing environment.

Another work [15] exploits the profile extension mechanism [14] to add definitions of some stereotypes and a set of UML diagrams that enable specification of real-time systems and their properties. To specify properties of such systems, specification-

classes are defined having predefined constraints presented in an extended variant of Timed Computation Tree Logic (TCTL).

Likewise, [3] proposes a formalization of an extension of UML state diagrams for specification of real-time behavior. This enhancement is achieved by the timed statecharts formalism of Kesten and Pnueli. But the next translation step into timed automata is not defined to take advantage of their analysis tools like model-checkers.

The second family defines mappings from UML constructs into rigorous formalisms such as Petri nets and Formal Specification Techniques (FST) to accomplish consistency checking. This way, the approach of [8] exploits a formal notation like Z instead of OCL to formalize UML Components so that the usual facilities for Z become available for type checking and proving properties about components. But the main disadvantage of using Z is that faithfully mapping the UML semantics to Z can result in very verbose and cumbersome specifications.

In a similar work [17] the author proposes a mapping using the incremental two-way translation between UML and SDL concepts. The translation of a subset of UML state diagrams to the SDL ones proceeds by flattening a fragment of nested states. Nevertheless, several suitable concepts for reactive systems are abandoned like concurrent hierarchical states, history states and boundary-crossing transitions of which handling needs complex redefinitions of the translation.

In a same way, the approach of [5] defines a mapping from UML models consisting of use case, class and interaction diagrams to their equivalent in E-LOTOS to form a single formal model in E-LOTOS. However the synchronous communication mode of LOTOS compels adoption of the zero-time semantics and excludes many situations where asynchronous communication is necessary.

There are also works based on graphs as target models, like that of [9] transforming a subset of UML state diagrams into graphs via rewriting rules. Likewise the authors of [4][2][10][11] indicate too dynamic semantics of fragments of UML models based on enhanced Petri nets (respectively high-level Petri nets, object Petri Nets and Generalized Stochastic Petri Nets). Draft heuristics are given in both these papers to transform parts of the UML models into the target formalism.

Another alike paper [12] extends UML with performance annotations (paUML) to deal with the performance indices in mobile agent systems that are modeled in conjunction with design patterns. Then the paUML models are semi-automatically translated to generalized stochastic Petri nets (GSPN).

In a particular way, the author of [18] defines a toolkit of specification techniques for requirements and design engineering (TRADE) where some choices close to the Statemate semantics of StateCharts are adopted to define transition system semantics for UML specifications.

However, the works mentioned above (including GSPN-based ones) deal only with a subset of state diagrams close to particular application domains and the translation rules are not well formalized. Moreover discarded concepts such as concurrent composite states and history vertices are important for modeling reactive and concurrent systems. Also events queuing patterns do not fit for situations where one event may trigger many transitions at once in various orthogonal regions.

Accordingly we define in this paper a new translation method using timed Petri nets as target model. Instead of using time tag over transitions as done in [10], we prefer tag each event token with a time interval because this seems more faithful to

model any dispatching delay. Our method allows also enabling many transitions with respect to the same trigger event. Furthermore, we deal with all kinds of hierarchical states and the most of pseudo-states like history ones. Obviously, introducing these concepts both with boundary-crossing arcs raises many problems we resolve by means of our enhanced model so that UML semantics [13] remain preserved.

Once the stateChart of an UML model is translated into a timed Petri net, we could achieve a consistency validation of the dynamic view with respect to sequence diagrams. The methodology may consist in checking whether event sequences are consistent with those of the reachability graph of the resulting net. Also it may find out whether and how temporal constraints of events paths could be fulfilled with respect to time intervals on the arcs of the reachability graph.

3 UML StateChart Features

The UML State Machines are an object-based variant of HAREL StateCharts that can model discrete behavior of objects through finite state-transition systems [13].

A StateChart (called also state diagram) shows the sequences of states that an object goes through during its lifetime in response to events, together with its response to those events. Note that an event can be a (asynchronous) signal, (synchronous) operation invocation, a time passing or a condition change.

A state is a condition or situation during the life of an object during which it satisfies some condition, performs some activity or waits for some event. Here an event is an occurrence of stimulus that can trigger a state transition. Note that a state may be either simple or composite. Any state enclosed within a composite state is called a substate of that composite state. It is called a direct substate when it is not contained by any other state; otherwise it is referred to as a transitively nested substate. When substates can execute concurrently, they are called orthogonal regions.

So a transition (arc) is a relationship between two states indicating that an object in the first (source) state will perform certain actions and enter the second (target) state when a specified event occurs and specified conditions are satisfied.

In general a state has several parts:

- *"Entry/exit"* actions are executed on entering and exiting the state, respectively.
- *"Do"* activity is executed while being in a state. It stops itself, or the state is exited, whichever comes first.
- *Substates* represent the nested structure of a state, involving sequentially or concurrently active substates connected via *internal* transitions.

When dealing with composite and concurrent states, the simple term "current state" can be quite confusing because more than one state can be active at once. If the control is in a simple state then all the composite states that either directly or transitively contain this simple state are also active.

Furthermore, since some of the composite states in this hierarchy may be concurrent, the current active state is actually represented by a tree of states starting with the single top state at the root down to the individual simple states at the leaves. We refer to such a state tree as a state configuration.

Also any transition originating from the boundary of composite state is called a high-level or group transition. If triggered, it results in exiting of all the substates of that composite state executing their exit actions starting with the innermost (deepest) states in the active state configuration.

3.1 Some Preliminary Expansion Rules

The pseudo-states are not states but only transient vertices used for more convenience when modeling the state machine graph. Thus, in order to simplify our translation method we remove all pseudo-states. But before erasing them, expansion rules below must be done with respect to the well-formedness rules defined in [13]:

- Any composite state containing a *shallow history* pseudo-state, should be labeled with the "history" tag.
- Any state containing a *deep history* pseudo-state, is labeled with the "history" tag and all its direct and transitively nested substates are also labeled with the "history" tag. This rule guarantees the handling of the history property in all transitively nested substates of any composite state owning a deep history pseudo-state.
- Any target state of a transition originating from an *initial* pseudo-state, is labeled with the "initial" tag.
- Target states of a transition originating from a *fork* pseudo-state are labeled with the "initial" tag. Obviously each initial state must belong to one orthogonal region of a concurrent composite state. The transition arc incoming to the fork vertex will be replaced by an incoming transition arc to the edge of the composite state
- All substates (in different orthogonal regions of a composite state) of which transitions merge into a join (terminator) pseudo-state, are labeled with the "final" tag. The transition arc outgoing from a join vertex to one target state, will be a triggerless outgoing arc from the edge of the composite state to the target state.

3.2 Mathematical Structure of a StateChart

Consequently to this previous operation and before defining the translation method of a state machine D into its equivalent Petri net, we use for more convenience the below mathematical structure to model the StateChart D.

A StateChart is a tuple: $D = <S, K, Tag, C, TA, S_0, L, E, G, A>$ where:

- S : set of states with the topmost state S_0.
- K : S→{Simple State, Sequential Composite State, Concurrent Composite State}.
- Tag : S→{Initial, Final, History}.
- C : S → 2^S is a mapping that gives to each composite state its nested states.

Note that $s \not\subset C^*(s)$ where $C^*(s) = \cup_i C^i(s)$ and $C^{i>0}(s) = \cup\{C^{i-1}(s_j)$ for $s_j \in C(s)\}$.
This mapping defines a partial order relation ($\subset$) among states we can depict as a tree.

- TA $\subseteq$SxS: Set of transition arcs.
- L is a labeling mapping: TA→ExGxA where E is the set of events, A is the set of actions and G is the set of guards.

4 Target Model of Translation

The target model of the translation is a kind of interval timed Petri net denoted by ITPN [1]. A marked ITPN is a tuple R = <P, T, Pre, Post, L_1, L_1, Prior, M_I > where:

- P is the places set and T is the transitions set.
- Pre: T→2^P is a backward incidence function giving the inplaces (input places) of transitions and Post: T →$2^{P \times INT}$ is a forward incidence function giving the outplaces (output places) of transitions with the corresponding delay intervals.
- INT = { $[t_1,t_2] \in \Re^{\geq 0} \times \Re^{\geq 0} / t_1 \leq t_2$ } the set of time intervals.
- L_1 and L_2 are labeling functions. L_1: P→EVT (EVT is the set of events) and L_2: T→ACT (ACT is the set of actions).
- Prior ⊆ T x T is a priority relationship between transitions in conflict.
- M_I : P → א is the initial marking of the net at instant 0 from system starting.

A transition "t" is *enabled* within a marking M if the required tokens in inplaces are available: ∀p∈Pre(t) : M(p)=1. Because the arcs are weighted with 1, when a transition "t" fires, a new marking M' is produced as follows: M' = M∪Post(t)–Pre(t).

"t" consumes a token in each of its inplaces and generates one in each of its outplaces. However the timing policy of this ITPN requires any transition to be fired as soon as it is enabled. The produced tokens become available in outplaces only after freeze-up delays that are sampled respectively from the time intervals relating to the arcs joining the transition "t" to its outplaces. If there's no time interval on an arc, we consider the default interval [0,0].

The priorities between transitions aim at setting in order the simultaneous executions of several boundary-crossing arcs originating from transitively nested substates of a same superstate. According to [13] the priority level decreases from innermost substates down to the outermost ones such that when triggered at once, exit actions from the former substates execute before those of the former ones.

In the next sections we denote respectively by *entry* and *exit* places some initial and final places in a Petri net component. These particular places provide the links to connect together subnets relating to substates of any composite state.

In addition, each action will be mapped into an event raise, so that it makes it possible to join any net transition triggering one event to inplaces of another transition triggered by this specific event.

5 Transformation Patterns

Below we present the main patterns about different cases of translation where we use "transition arcs" to mean UML transitions in contrast with Petri net transitions.

5.1 Transition Arc Between Two Simple States

Given one transition arc between two simple states S_1 and S_2 belonging to the same containing state, we distinguish two cases requiring different treatments:

1. The trigger event may be a call event or signal event,
2. The trigger event may be a time event.

Case 1: The trigger event is a call event or signal event. The transition arc is mapped here into a net transition as follows (Fig.1):

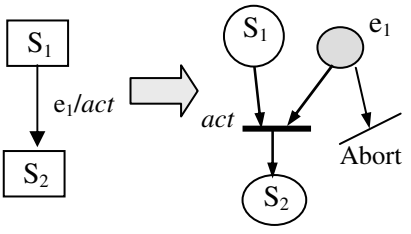

Fig. 1. Arc labeled by a call/signal event

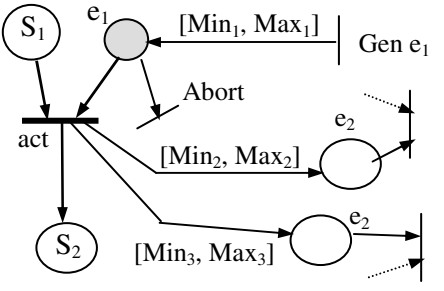

Fig. 2. Connecting trigger event places

The source state S_1 is translated into an inplace of *act* depicting the incoming control flow and the target state S_2 is related to an outplace depicting the outgoing control flow. For the trigger event of the transition *act*, we generate another inplace called event place that depicts the event flow. This special place would be an outplace of any transition raising the relating event e_1.

In the same way, the action *act* labeling the net transition may consist in generating some event e_2 used by other transitions in the StateChart. Hence we should link later our net transition *act* to all (event) places labeled with the trigger event e_2.

Note that the abort transition models the possibility to discard the event e_1 if the state machine in not yet in S_1 with respect to UML semantics. Since any transition must fire whenever it becomes enabled, we must make the priority of the transition *act* superior than that of the abort transition (see Fig.2).

Furthermore, we can label with a time interval any outgoing arc from a net transition to a trigger event place. This interval models well the minimal and maximal time delays between the queuing and dispatching instants of the event. The variable transmission delays depend on the properties of the communication medium and other factors. For instance the possibility of event loss can be depicted by an interval $[x,\infty[$ where x denotes the minimal delay to deliver that event for processing. However, if we would model the synchronous communication we can do it by using the only interval [0,0] so that trigger events become available as soon as they are raised.

Case 2: The trigger event is a time event (when d). The translation of a time event arc into a net transition needs also an event inplace we should join as an outplace to the transition "T" which produces a token in the related place of S_1 (Fig.3).

However the net arc between "T" and the event place would be tagged with a time interval [d,d]. So if "T" fires, it produces a token in each of the two places relating respectively to the state S_1 and the event e. Here the place S_1 is marked immediately, whereas the place e receives its token after d time units. Once this token becomes available and if S_1 is still marked then the transition act is fired. Otherwise, the abort transition is performed consuming the time event token (Fig.3). Note that other transitions outgoing from S_1 may fire so that the transition *act* becomes disabled.

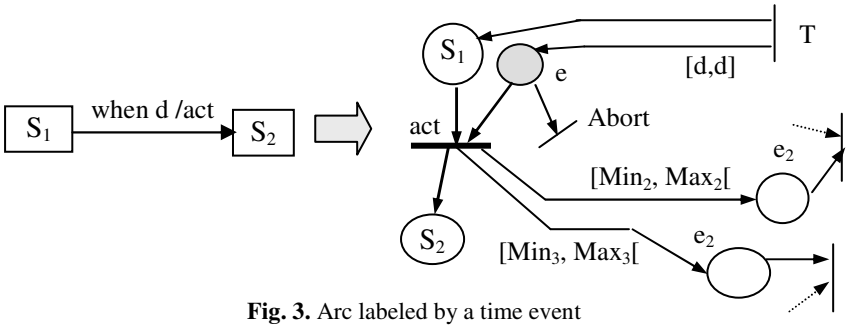

Fig. 3. Arc labeled by a time event

5.2 Transition Arc from a Simple State to a Composite State Without a History Tag

When the trigger event e_1 is dispatched, the system processes the transition action *act* immediately if the state machine is in the suitable state S_1 that has an outgoing arc triggered by e_1. Otherwise, this event e_1 is discarded.

As the arc tagged with e_1 goes to the edge of the composite state S_2, the outplace of the *act* transition is the initial place of the related net to S_2 if this one is sequential (Fig.4). But if S_2 is a concurrent state (Fig.5), all places relating to respectively initial states of concurrent regions of S_2 will be outplaces of the *act* transition. Hence when firing this transition, a token is produced in the initial place of each region of the state S_2, making it possible to activate both all those concurrent regions in the same time.

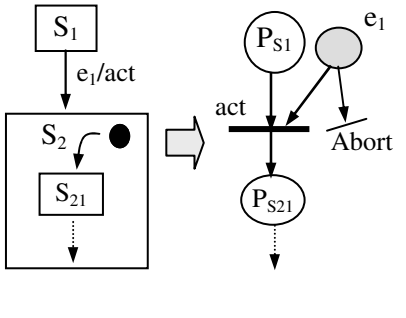

Fig. 4. Sequential composite state

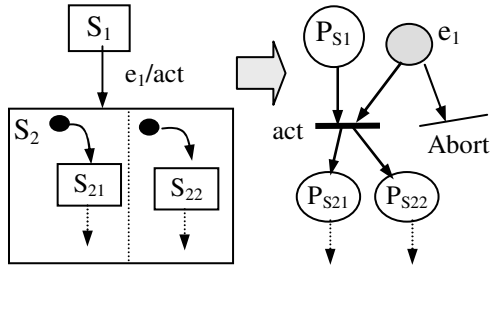

Fig. 5. Concurrent composite state

5.3 Transition Arc from a Composite State Without History Tag

In Fig.6 there are three kinds of transition arcs exiting from the state X.

The simple one is a triggerless transition outgoing to the state G. In this case, we connect the subnet relating to the final substate D of X with the subnet relating to the target state G (the subnets of simple states are single places).

The second kind is a high-level transition outgoing from the edge of X to the state E. Here we have to connect each one of subnets relating to the direct substates of X

with the place of G via a transition p triggered by the event e2. However instead of duplicating three times the transition p in X for each substate of which place becomes its inplace, it is more suitable to use (in Fig.6) all places related to X/B, X/C, X/D as optional inplaces for a single transition p that is triggered by e_2. This feature is depicted by dashed lines joining the inplaces (B, C and D) to the transition p.

The third transition is an exit arc from only one direct substate C to the new target state F. So we have to connect only the subnet of C with the subnet relating to F.

Note that when exiting one region to somewhere outside its concurrent composite state, the other orthogonal regions of X should be also disabled [13].

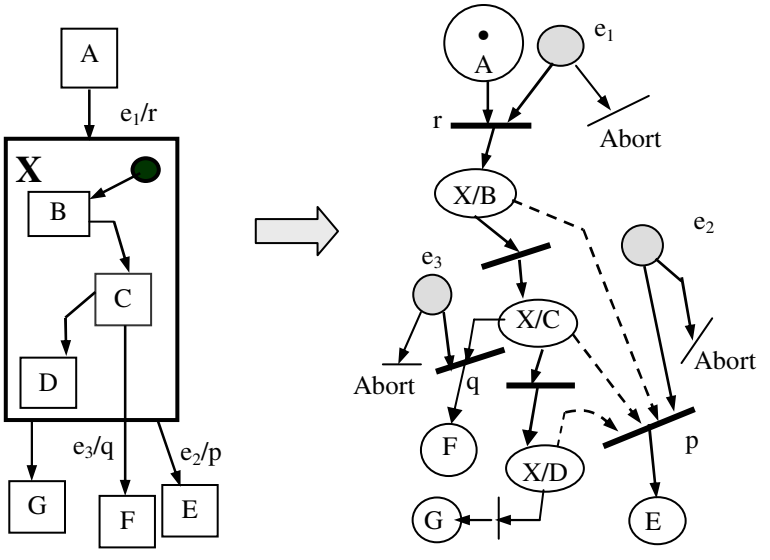

Fig. 6. Source state of an arc is a sequential composite state

5.4 Transition Arc from a Simple State to a Composite State with a History Tag

In Fig.7, the state S_2 is a sequential composite state with the history tag. So when this state is exited and then reentered again, the system should return to its most recent active configuration; that's the state configuration that was active when the composite state was last exited.

Therefore we use a "*thread*" place P_H as entry place to model control passing between substates of S_2.

Remind that an entry place is not marked at start but during net execution it may receive a token from a previous transition so that it activates S_2.

Moreover the place P_{S21} relating to the first substate S_{21} will be initially marked. When firing some internal transitions in S_2, the token of P_{S21} moves to the next places relating to substates of S_2 making it possible to determine every time the current marked place and consequently the related active substate.

Hence if the system leaves S_2 (removing P_H token) while a token is in an intermediate place and if it returns to S_2 (renewing a token in P_H), the recent active

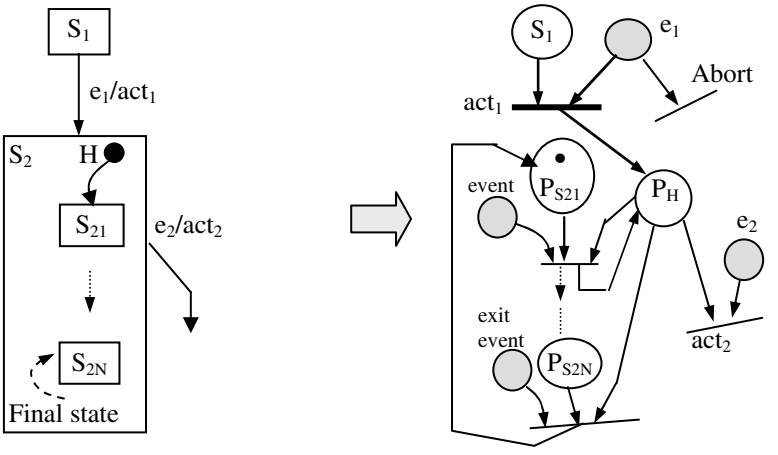

Fig. 7. Sequential composite state with history tag

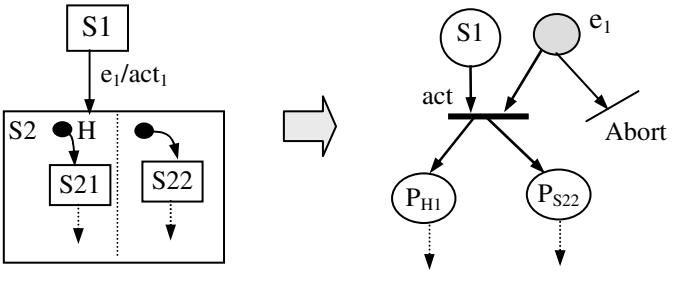

Fig. 8. Concurrent composite state with history tag

configuration is restored because the intermediate place remains marked until the control of S_2 is restored. However if a triggerless exit transition fires, the P_H token is finally consumed whereas a new token is produced in the initial place P_{S21}.

In Fig.8, S_2 is a concurrent composite state with two orthogonal regions of which one is labeled with the history tag. Here we apply to each region the suitable handling method among those we have presented above. It is obvious that the firing of act_1 transition produces a token in the appropriate entry place of each region. Whenever the region is tagged with "History", we consider its thread place P_H as an entry place. Otherwise, the place relating to the first substate of a region is taken as entry place.

5.5 Transition Arc from a Composite State with a History Tag to Other States

To simplify the resulting net in Fig.9, we do not add the transitions that represent event discarding. The dashed lines mean that the transitions p and q should have Y_1, Y_2 and Y_3 as optional inplaces. We recall that exit transitions of orthogonal regions must synchronize. So the exit arc, which is event triggerless, requires joining the final place of each region to that transition q. In addition we join X/P_H to the transition q which firing renews a token only in the initial place X_1 of the history tagged region.

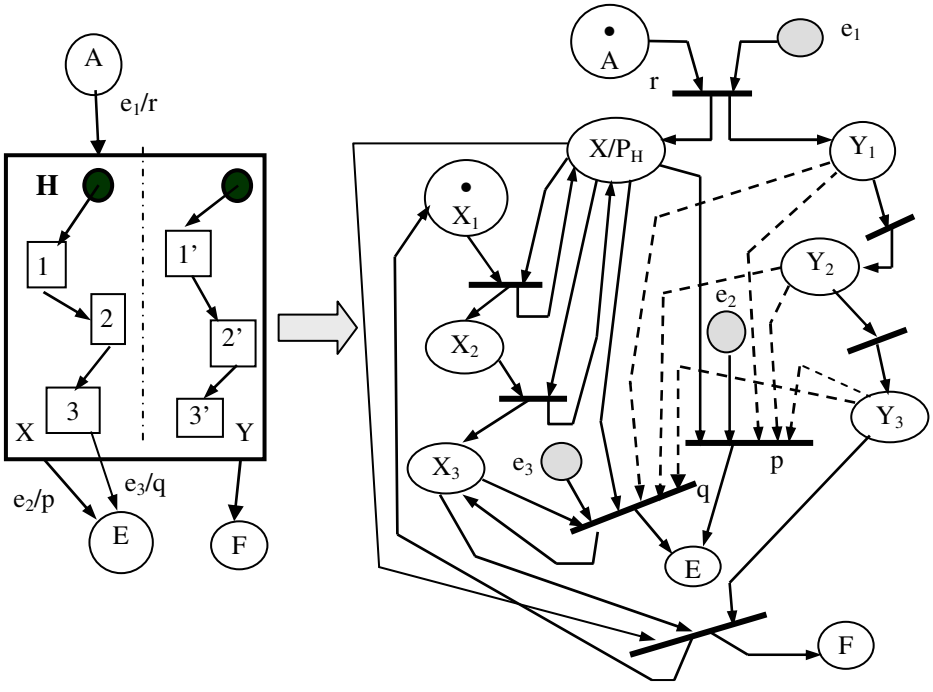

Fig. 9. Concurrent composite state with history tag

If the exit transition is a compound one (i.e. e_2/p) outgoing from the edge of a composite state, then we use as inplace the thread place of the history tagged region and as optional places all the places of the other orthogonal region without history tag.

6 Outlines of the Translation Algorithms

In the first step of the translation, we discard the boundary-crossing arcs that transgress the encapsulation concept of states [16]. Note that high-level transitions are also boundary-crossing transitions which are handled in the same manner.

As handling boundary-crossing arcs together with history vertices is too complex, we prefer go by steps and defer their treatment to the second step of the translation.

Once we obtain all related nets of a whole system, we combine them by means of the parallel operator. Every transition raising an event must have as outplace each one labeled with this event. Afterward, we could tag these added arcs with time intervals modeling dispatching delays of events or time constraints on StateCharts arcs.

6.1 First Step of Translation

Given some state machine D that consists of one topmost level state S_0. We discard all boundary-crossing arcs and then we construct the related net as follows:

Let $S_1,\ldots,S_N$ be the direct substates of S_0 that may be connected together either sequentially or concurrently. First we construct the net $|[S_i]|$ relating to each substate S_i by applying recursively the same *algorithm1*. Then we use one of the two

algorithms *Algo1-Seq* and *Algo1-Par* to combine together the resulting nets $|[S_i]|$ by means of one of two connecting operators; the used operator may be either a parallel composition connector $\|$ (if S_0 is a concurrent state) or a sequential composition $\otimes$ (if S_0 is a sequential state).

If $K(S_0)$ = Concurrent then $|[S_0]| = |[S_1]| \| \ldots \| |[S_N]|$.

If $K(S_0)$ = Sequential then $|[S_0]| = |[S_1]| \otimes \ldots \otimes |[S_N]|$.

The same approach is then employed recurrently to each substate whenever this one is a composite state. The sketch of *algorithm1* is a follows:

```
Algorithm1 (S : IN UML State Machine) → |[S]| : ITPN
Begin
    if S is a simple state then  Algo1-Simple-State (S, N)
    else begin
           let Subnets-List := ∅;        // Subnets relating to
           for each substate SSi of S    // direct substates of S
              do begin   N_i := Algorithm1(SSi);
                         Add(Subnets-List, N_i);
                 end
           if K(S)=Sequential then Algo1-Seq(S, Subnets-List);
           if K(S)=Concurrent then Algo1-Par(S, Subnets-List);
       end
end Algorithm1.
```

Algo1-Simple -State handles only simple states by translating each one "S" of them into one place P_S. If S has not entry/exit actions, P_S becomes both an entry and a final place of that related subnet |[S]|. Otherwise, the simple state with entry/exit actions is translated into three places, P_{Entry}, P_S and P_{Exit} that are linked by means of the entry/exit transitions as depicted in Fig.10.

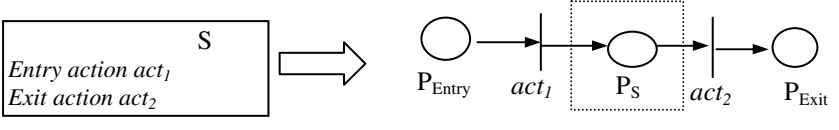

Fig. 10. Entry/exit actions handling in a simple state

The same approach is also applied to a composite state (both in *Algo1-Seq* and *Algo1-Par*) by substituting Ps with the subnet |[S]| related to this state S considered without its entry and exit actions. P_{Entry} is linked by means of the act_1 transition to the initial places of the subnet |[S]|, namely those which would receive the tokens to initiate the activity in that part of the system. Likewise, all final places of the subnet |[S]| have to be linked to P_{Exit} by means of the transition *act2*.

Below we give the core of the subroutines Algo1-Seq and Algo1-Par :

```
Algo1-Seq (S: StateChart; Subnets-List: List of ITPN)→ ITPN
Begin
     for each arc Arc between SS_i and SS_j ∈ Substates(S)
      do if SS_i is a simple state or Arc is triggerless then
          begin   Add a transition T;
                  Join |[SS_i]| & |[SS_j]|∈ Subnets-List via T;
                  Label T with the action name of Arc;
                  Add an inplace of T that is labeled with
                    the trigger event name if it exits;
```

```
        end
    if S is history state  then  add a thread place P_H ;
    Link P_H at the same time as inplace and outplace of the
      each transition of all substates of S;
    Handle the entry and exit actions if they exist.
end Algo1-Seq.
```

Note that if the exit action exists in a history tagged state S then we should add in |[S]| an exit transition which is joined to P_H only as inplace such that when T_{exit} fires it consumes the P_H token. Thus any activity in |[S]| will be disabled. We should also reset the net by joining T_{exit} to the initially marked place of the first substate of S.

```
Algo1-Par (S: StateChart; Subnets-List: List of ITPN)→ ITPN
Begin
    Juxtapose together all regions nets ∈ Subnets-List;
    Link each transition generating an event e to all event
      places labeled with e as outplaces;
    Add a fork place P_entry;
    Join P_entry to the entry places of subnets through an entry
      transition T_entry;
    Label T_entry with the entry action name if it exits;
    Add a join place P_exit;
    Join P_exit to the final places of subnets through an exit
      transition T_exit;
    Label T_exit with the exit action name if it exits;
end Algo1-Par.
```

6.2 Second Step of Translation

In this step we treat boundary-crossing arcs which triggering causes exiting also the concerned superstates starting from the innermost one in the active configuration.

We recall that high-level transitions belong also to this category of arcs. Indeed a high-level transition originates from the boundary of a composite state S and so it can occur wherever the substate the control is in. Therefore we expand any transition of this kind into a group of simple boundary-crossing arcs where each one of them originates from one simple substate of the composite state.

For each boundary-crossing arc *Arc, Algorithm2* generates a transition T with an inplace labeled with the triggering event of *Arc*. Then it simply joins T to the entry place of the subnet related to the target state of *Arc*.

However it is more difficult to cope with the source state (S) of *Arc*. Obviously we join its related place Ps to T as inplace to disable |[S]| when *Arc* occurs. Especially T will have a list of labels, which contains at least the exit action name of S and the name of the action on *Ac*.

The next subroutine of *Algorithm2* depends on whether the superstates of S have a history tag. When the state S is exited, the next procedure assures that any superstate of S of which the superstate is history tagged, should become active when their outermost state is reentered later. The main idea is to regenerate a token through *Arc* in the entry place of any superstate S' that is itself enclosed within a history tagged state. Obviously the entry place depends on whether S' is history tagged (Fig.11).

```
Handle-Boundary-Crossing-Arc (Arc: IN OUT StateChart Arc)
  Begin
    S':= Source state of Arc;     // S' is a simple state
```

```
    S″:= SuperState(S′);
  while S″ does not contain the target state of Arc
     do begin
          Add to Label(T) the exit action name of S″;
          if S″ has a history tag then
                if S′ has not a history tag
                     then join T to the entry place of |[S′]|
                     else join T to the thread place of |[S′]|;
          S′ := S″;   S″ := SuperState(S″);
        end
end Handle-Boundary-Crossing-Arc.
```

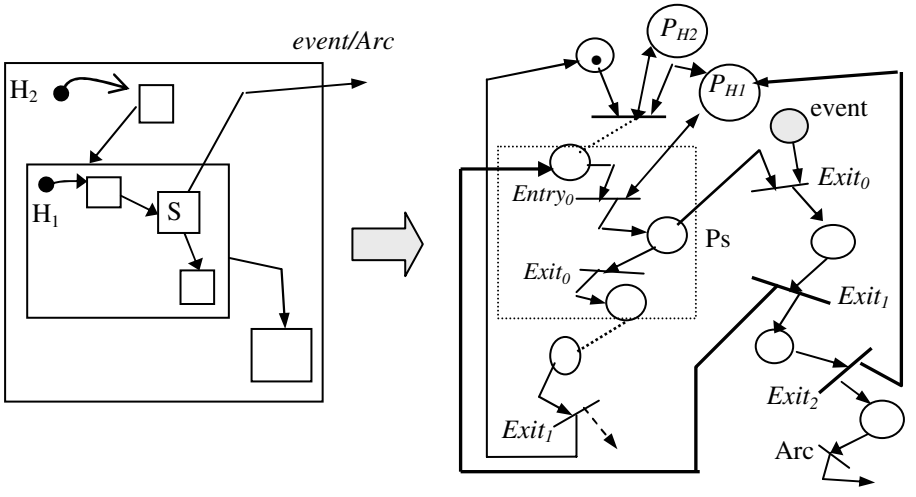

Fig. 11. Boundary-crossing arcs

Remark. Whenever one region of a concurrent state is exited by a boundary-crossing arc, all of orthogonal regions should also be exited. Therefore we add as *optional* inplaces some adequate places of the other orthogonal regions to the exit transition which becomes enabled only when its standard inplaces are marked. But when it fires, tokens in both usual and optional inplaces are consumed wherever the latter ones are available. Hence even if an optional inplace is not marked, the enabled transition fires. Here, thread places of the orthogonal regions are joined as optional inplaces to the exit transition which firing disables the subnets of the other regions.

7 Conclusion

This paper presents a new approach towards the formalization of UML by translating the StateCharts into a rigorous language, namely Interval Time Petri Nets (ITPN). We enhanced the previous formalism with some suitable concepts so that our method deals with both all kinds of composite states and pseudo-states and allows us to model the various dispatching delays of events and timing information on StateChart arcs.

Once an UML State diagram is converted into a Petri net, we can make use of existing tools for Petri net analysis [1] or check the semantically and temporal consistencies with

reference to sequences diagram. In this setting, the intended methodology consists of generating the reachability graph from the related Petri net of any UML model and mapping the sequence diagram into a set of execution paths. Afterward, one can find out whether our graph embeds all these paths and whether timing constraints on events sequences are fulfilled considering the time intervals on graph transitions.

Similarly our approach can also be generalized to the activity diagrams since they are based on UML state machines. However some improvements would be performed to take into account some specific aspects of this kind of UML diagrams.

References

1. W.M.P. van der Aalst. Interval Timed Petri Nets and their analysis. Computing Science Notes 91/09, Eindhoven University of Technology, May 1991.
2. Luciano Baresi. Some Preliminary Hints on Formalizing UML with Object Petri Nets. Integrated Design and Process Technology IDPT-2002, June 2002.
3. Vieri Del Bianco, Luigi Lavazza, Marco Mauri. A Formalization of UML Statecharts for real-time software modeling. Integrated Design and Process Technology IDPT-2002.
4. Luciano Baresi, Mauro Pezzè. Improving UML with Petri Nets. Electronic Notes in Theoretical Computer Science 44 N° 4 (2001).
5. Robert G.Clark, Ana D. Moreira. Use of E-LOTOS in adding Formality to UML. Journal of Universal Computer Science 6(3) 1071-1087, 2000
6. A.Evans, J-M Bruel, R.France, K.Lano. Making UML Precise. In Proc. of the OOPSLA'98 Workshop on Formalizing UML, 1998.
7. Gregor Engels, Jan Hendrik Huasmann, Reiko Heckel, Stefan Sauer. Testing the Consistency of Dynamic UML Diagrams. Integrated Design and Process Technology IDPT-2002.
8. R.France, A.Evans, K.Lano, B.Rumpe. The UML as a formal Notation. UML'98 – Beyond Notation First International Workshop, Mulhouse, France, June 1998.
9. M. Gogolla, F.P. Presicce. State diagrams in UML: A formal semantics using graph transformations. In Proc. of PSMT'98 workshop precise semantics for modeling techniques.
10. G. Huszerl, I. Majzik, A. Pataricza, K. Kosmidis and M. Dal Cin. Quantitative Analysis of UML Statechart Models of Dependable Systems. The Computer Journal, Vol.45, N°3, 2002.
11. Peter King and Rob Pooley. Derivation of Petri Net Performance Models from UML specifications of Communications Software. B.R.Haverkort et al.(Eds.): TOOLS 2000, LNCS 1786, pp.262-276, 2000.
12. J. Merseguer, J. Campos, E. Mena. A pattern approach to model software performance using UML and Petri Nets: Application to Agent-based Systems. Workshop on software and performance, Ottawa, Sept.2000
13. Object Management Group, Inc. OMG Unified Modeling Language Specification. March 2003, Version 1.5, Formal/03-03-01.
14. Object Management Group, Inc. UML Profile for Schedulability, Performance, and Time Specification. September 2003, Version 1.0, Formal/03-09-01.
15. E.E.Roubtsova, J.van Katwijk, W.J.Toetenel, C.Ponk, R.C.M de Rooij. Specification of Real-Time Systems in UML. Electronic Notes in Theoretical Computer Science 39(3), 2000.
16. Anthony J.H. Simons. On the Compositional Properties of UML Statechart Diagrams. Rigorous Object-Oriented Methods, 2000.
17. K. Verschaeve, A.Ek. Three scenarios for combining UML and SDL'96. Eighth SDL Forum, Montréal, Canada, 1999.
18. Roel Wieringa. Formalizing the UML in a Systems Engineering Approach. In Proc. of second ECOOP workshop on precise behavioral semantics, 1998.