A Formal Semantics of Timed Activity Diagrams
and its PROMELA Translation

Nicolas Guelfi and Amel Mammar
Software Engineering Competence Center
Faculty of Science, Technology and Communication
University of Luxembourg, L-1359 Luxembourg-Kirchberg, Luxembourg
{nicholas.guelfi,amel.mammar}@uni.lu

Abstract

The lack of a precise semantics for UML activity diagrams makes the reasoning on models constructed using such diagrams infeasible. However, such diagrams are widely used in domains that require a certain degree of confidence. Due to economical interests, the business domain is one of these. To enhance confidence level of UML activity diagrams, this paper provides a formal definition of their syntax and semantics. The main interest of our approach is that we chose UML activity diagrams, which are recognized to be more tractable by engineers, and we extend them with timing constraints. We outline the translation of our semantics into the PROMELA input language of the SPIN model checker which can be used to check several properties.

1. Introduction

The work presented in this paper is part of the Efficient project [3] that aims at providing business experts with a tool set for the building of trusted business processes. A business process can be defined as a sequence of actions that is intended to be completed in order to produce a specific result. Such actions executed by actors produce messages (business documents) that can be input to other actions. In order to represent a business process in a way that both business experts and IT analysts can understand, we have chosen UML [20] activity diagrams. In the context of Efficient, UML diagrams are edited under the commercial MagicDraw CASE tool [15]. The modeling process is not addressed in this paper, more details about it can be found in [17].

It is recognized that the lack of a precise semantics for UML diagrams prohibits rigorous analysis and reasoning on the obtained models. To overcome this shortcoming, in this paper, we define a formal syntax and semantics of activity diagrams endowed with timed characteristics. In fact, in the business domain, the emphasis is made on the response time properties that guarantee a quality of the service.

To allow business users to check properties on their transactions, we have defined a set of translation rules of our semantics into the PROMELA formal language. PROMELA is the input format of the SPIN[10] model checker that allows a wide range of propositional requirements to be checked. There are several reasons for using model checkers instead of theorem provers. Model checking is a completely automated technique. Moreover, if a requirement fails, a counterexample is made available which helps understand why it has failed. We have chosen the SPIN tool instead of any other timed model checker for different reasons. PROMELA is rather a simple specification language whose syntax is very similar to that of the C language, which is not too difficult to understand. Furthermore, the SPIN tool is used in a large number of applications in several domains including the aerospace industry [6]. In addition, our previous experience with the Kronos tool [1] has shown that the translation of an activity diagram into timed automata requires a large number of complex rules.

The OMG semantics of UML 1.5 activity diagram is based on their translation into statechart for which several semantics already exist [11], [2], [14], [13]. We have not considered such semantics because although activity and statechart diagrams seem syntactically similar, however as Eshuis has demonstrated in [4], not every activity diagram can be translated into a statechart diagram. Besides, the forthcoming UML 2.0 distinguishes drastically statechart and activity diagrams. Consequently, we have chosen to define a new semantics of activity diagrams that could be adapted easily to the one defined in UML 2.0. In [7], a definition of a semantics for collaboration and activity diagrams based on Place-Transition Petri Nets is proposed. We have not considered this as feasible to achieve since such work are not mature enough and not supported by tools. Moreover, integrating Petri-net analysis tools, like PEP[19] or others, in our development environment (MagicDraw) seem to be very difficult. The semantics presented in this paper is comparable to the one presented in [5] in terms of Clocked Transition System (CTS) and used to support the automatic verification of properties using NuSMV[18]. Our approach has the advantage of dealing with loops which may be very
interesting in the business domain. Moreover, in Eshuis’ approach, no details are given about how timers are defined, it seems that there are as many timers as timed transitions. However, a huge number of timers would reduce the efficiency of model checkers. Finally, Eshuis does not provide information about how he translates his semantics into the SMV language.

The remainder of the paper is structured as follows. Section 0 starts with explanations of the syntax of the timed-activity diagrams we consider. Our formal semantics of these diagrams is described in Section 0. Section 0 presents the PROMELA translation of the proposed semantics. Our approach on how to deal with loops is explained in Section 0. Finally Section 0 concludes and presents some potential future work.

2. Syntax of timed UML activity diagrams

The system that we use for illustration purposes deals with a simplified production scenario described by the activity diagram of Figure 1. Ovals and rectangles represent activities and documents (action and object flow states respectively in UML 2.0 terminology). A bar models an AND-node which represents a fork or a join. Similarly, a diamond denotes an XOR node, which is either a split or a merge. The underlying transaction starts at the black dot, ends at a bull’s eye and has the following informal semantics. Two time periods after a customer’s order has been received, there are two parallel activities that are launched: “CheckStock” and “CheckCustomer”. The former checks whether the quantity in stock is enough to correspond to the quantity requirements of the customer’s order, the latter checks whether the status of the customer entitles him to place orders at all. If the quantity in stock is deficient, a restocking plan is launched. Also, depending on the reliability of the customer, a reject letter or a bill is sent to him. In case of a reliable customer, after 15 time periods from the reception of the order, the transaction ends by shipping the order. For some activities, the desired behavior duration is specified.

In the context of our project, a timed UML activity diagram is a tuple \( TAD = (\text{Nodes}, \text{Duration}, \text{Events}, \text{Trans}) \) where:

- \( \text{Nodes} \) denotes the set of nodes. This set is partitioned into: set \( AN \) of activity (or action) nodes, set \( ON \) of object nodes, set \( FN \) of final nodes, an initial node \( IN \), set \( DN \) of decision nodes, and set \( FN \) (resp. \( JN, MN \)) of fork (resp. join, merge) nodes. A document is an output result of an action and can be input to another action. An action denotes a piece of work that has to be carried out, and for which duration may be specified: \( \text{Duration}: AN \rightarrow \mathbb{N} \cup \{ \bot \} \) where \( \bot \) represents the absence of a value (undefined value). We do not consider dense time because this would complicate our model considerably. In the context of e-business transactions, this is not a big restriction, since applications of this domain rather denote discrete-time models.

- \( \text{Events} \) is the set of event expressions. In this paper, we consider temporal events of the form \( \text{After}(t) \) and \( \text{When}(t) \) where \( t \) denotes a strictly positive integer. Note that external events are not considered.

- \( \text{Trans} \) is the set of transitions connecting the nodes. Each transition is described by its source and target nodes respectively: \( \text{Source} : \text{Trans} \rightarrow \text{Nodes} \) and \( \text{Target} : \text{Trans} \rightarrow \text{Nodes} \). Except transitions sorting from decision nodes and those incoming to join or merge nodes, a transition may be labeled with a temporal event. The function that returns the temporal event associated with a transition is defined by: \( \text{EventTrans}: \text{Trans} \rightarrow \text{Events} \cup \{ \bot \} \). A transition labeled with \( \text{When}(t) \) (resp. \( \text{After}(t) \)) event means that its target node may be executed after time \( t \) has passed from the beginning of the execution of the transaction (a time equal to \( t \) after its source node has finished its execution). In other words, in case of \( \text{When} \) event for instance, the node cannot start execution neither before nor after a time \( t \). In our context, the meaning of the assertion “a node \( A \) has finished (has terminated)
A. If A denotes an action (resp. a document), it signifies that the piece of work the action represents has been accomplished (resp. the document is available). For all other nodes, it means that the execution of the transaction arrives at the corresponding node.

3. Formal semantics of activity diagrams

In this section, we give our formal semantics of UML timed-activity diagrams by mapping them to a clocked transition system (CTS) [12] restricted to integer variables modeling discrete real time aspects. Let \( Var \) be the set of variables, a state \( \sigma \) is defined by the values \( \sigma(v) \) of each variable \( v \) of \( Var \). We denote the set of all the states \( \Sigma(Var) \). Formally, a clocked transition system (CTS) is a tuple \( S=(Var, \sigma_0, \rightarrow) \) where:

1. \( Var = Status \cup ExecTrans \cup FireableTrans \cup Clock \cup Timers \) is a finite set of variables. \( Status \) is a function indicating the state of each node. In our case, we distinguish four different states detailed in Section 0: \( STATUS=\{"Sleep", "WaitTrans", "SetDuWait", "Finish"\} \). Function \( Status \) is then defined by: \( Status: Nodes \rightarrow STATUS \). \( ExecTrans \) is a function that gives, for each transition, the number of times it has been made fireable. It is defined by \( ExecTrans: Trans \rightarrow \mathbb{N} \). The use of integer type is especially required for activity diagrams that contain loops since their transitions can be passed several times. \( FireableTrans: Trans \rightarrow \mathbb{N} \) is a function that indicates the number of times a transition can be fired. Each time a node has finished execution and depending on its kind, it notifies that a new instance of one or all its outgoing transitions becomes fireable. This function is also required for loops where, for example, several instances of the same document are emitted before they are used. It is worth noting the difference between \( ExecTrans \) and \( FireableTrans \). The former counts, from the beginning of the transaction, the number of times a given transition is made fireable. The latter stores the number of times a transition can be fired in future. Indeed contrary to \( ExecTrans \), \( FireableTrans \) is decreased each time the corresponding transition is passed. Finally, to deal with the timed aspects, a master clock, denoted by \( clock \), and a set of additional clocks, called \( Timers \), are defined. \( Timers \) are used to count the elapsed time between two given moments. The master clock is needed to be able, at each moment; to retrieve the elapsed time from the beginning of the transaction. The next section outlines the key ideas for defining timers.

2. \( \rightarrow \subseteq \Sigma(Var) \times \Sigma(Var) \) denotes the transition relation between the set of states of the system.

3. \( \sigma_0 \) denotes the initial state of the system.

Before defining the transition relation, we outline the main principles of the approach we have developed to define the timers.

3.1. Definition of timers

To define timers, at least two solutions are conceivable. The simplest one consists in assigning a distinct timer to each timed transition and/or action with duration. The main drawback of this approach is the great number of timers needed in case of several timed actions and/or transitions. Such a great number of timers increases the size of the state-vector during analysis and thus increases the memory requirements for analysis. To cope with this, we have developed an approach that defines a minimum number of timers. The two following points summarize the central idea of our approach:

- A single timer is used for activities and/or transitions that are serially executed because their executions cannot interleave.
- Distinct timers must be used for activities and/or transitions whose executions may interleave.

In [16], we have described an algorithm that defines, for each given timed-activity diagram, a set of minimum timers satisfying the above two properties. Applying this algorithm to the activity diagram of Figure 1 defines two distinct timers \( timer_1 \) and \( timer_2 \) as shown in Figure 2.

![Figure 2. Defining timers](image-url)
As we can notice, different timers are associated with the two paths launched by the fork node. In contrary, the same timer is assigned to the two paths following a decision node. In fact, as “SendReject” and “ReceivePayment” cannot be active at the same time, the same timer timer is used. In contrary, two distinct timers must be used for “CheckStock” and “CheckCustomer” since they can be active at the same time. In the rest of this paper, function TimerNode returns the timer assigned to a given node: TimerNode: Nodes - {IN, FN} → Timers.

3.2 Transition relation

From a given state represented by the values of different elements of Var, the state of the modeled system evolves throughout the execution of the nodes of its underlying activity diagram. In our semantics, all the nodes are dealt with similarly. Except for the initial node, each node behaves according to the execution cycle depicted in Figure 3:

Figure 3. Execution cycle of a node

Let us give some explanation about each state of Figure 3:

- **Sleep**: the node waits that one or all its incoming transitions become fireable.
- **WaitTrans**: the node states that one fireable instance of all or some of its incoming transitions have been taken into account. In this state, the node waits until the value of its timer becomes null. It represents the passage of time related to its possible incoming timed transitions.
- **SetDuWait**: if the node denotes an action with duration, its associated timer will be set to the related duration, and the node has to wait for a timeout.
- **Finish**: depending on the kind of the node, different actions are done. In case of decision node, only one and one of its outgoing transitions chosen arbitrary is made fireable. In all other cases, timers of the outgoing timed transitions are set to appropriate values and all the outgoing transitions are made fireable. A transition is made fireable by increasing the value of FireableTrans for that transition.

At the beginning of a transaction, as the unique outgoing transition from the initial node is instantaneous and one cannot come back in the same transaction to the initial node, the initial state σ0 of the system satisfies the following assertions:

- all the nodes are in state “Sleep”,
- from the beginning of the transaction, there is exactly one instance of the transition outgoing from the initial node that is made fireable,
- the values of the master clock and all the timers are set to zero.

Formally, the initial state σ0 is defined by:

\[
\forall N \in \text{Nodes} \Rightarrow \sigma_0(\text{Status}(N)) = \text{Sleep} \\
\sigma_0(\text{FireableTrans}(\text{Elem(Source}^{-1}(\text{IN})))) = 1 \\
\forall tr \in \text{Trans} - \text{Source}^{-1}(\text{IN}) \Rightarrow \sigma_0(\text{FireableTrans}(tr)) = 0 \\
\forall N \in \text{Nodes} \Rightarrow \sigma_0(\text{FireableTrans}(tr)) = 0 \\
\sigma_0(\text{clock}) = 0 \land \forall \text{tim} \in \text{Timers} \Rightarrow \sigma_0(\text{tim}) = 0
\]

Where Element(X) denotes the unique element of the singleton set X. Source^{-1}(IN) returns the singleton set constituted of the transition outgoing from the initial node. Hereafter, we give the formal specification of each sub-step of Figure 3.

**Relation →Sleep.WaitTrans**. This relation models a node waits till all or one of its incoming transitions become fireable. In case of a merge node, the node waits until at least one of its incoming transitions becomes fireable. To define this relation, we have to determine the set of incoming transitions to each node N. This set, denoted by IncomTrans, is equal to: IncomTrans(N)=Target^{-1}(IN).

So, the relation →Sleep.WaitTrans is defined by:

\[
\sigma \rightarrow \text{Sleep.WaitTrans}(N) \Leftrightarrow \\
N \in \text{Nodes} \land \sigma(\text{Status}(N)) = \text{Sleep} \land \\
\forall N \in MN \land \exists tr \in \text{IncomTrans}(N) \land \\
\sigma(\text{FireableTrans}(tr)) > 0 \land \\
\sigma'[\text{Status}(N)/\text{WaitTrans}, \\
\begin{array}{c} \text{Dec}(\text{FireableTrans}(tr)) \end{array} | tr \in \text{IncomTrans}(N)] \\
\wedge \\
\forall N \notin MN \land \\
\forall tr \in \text{IncomTrans}(N) \\
\sigma'[\text{Status}(N)/\text{WaitTrans}, \\
\begin{array}{c} \text{Dec}(\text{FireableTrans}(tr)) \end{array} | tr \in \text{IncomTrans}(N)]
\]

Valuation σ'=σ[x // val] assigns to variable x value val, the other variables different from x remain unchanged. Symbol // denotes a bulk update: σ'[a(x_1),...,a(x_n)]≡σ[a(x_1)],...,a(x_n)], where n=#X. Dec(x) will decrement x if its value is strictly positive otherwise
nothing will done. Relation $\rightarrow_{\text{SleepWait}}$ says that a node $N$ becomes in state “WaitTrans” if and only if:

1. $N$ is already in state “Sleep”
2. If $N$ is a merge node, then it exists at least one new instance of its incoming transitions that is fireable.
3. In all other cases, it exists at least one new instance of each incoming transition to $N$ that is fireable.
4. Finally in both cases, the status of $N$ is updated to “WaitTrans”, and one fireable instance of each incoming transition is removed.

Relations $\rightarrow_{\text{WaitTrans_SetDuWait}}$. To define this relation, we have to determine the set of running timers. The running timers represent the timers whose values are positive. In fact, only these timers can be decremented. This set, denoted by $RT$, is defined by: $RT = \{t \mid t \in \text{Timers} \land \sigma(t) > 0\}$. The passage of time is represented by relation $\rightarrow_{\text{time}}$, defined below. The running timers decrease of $1$, while the master clock increases of $1$. We have discretized the time with clock ticks of $1$ time. Such a discretization is acceptable as demonstrated in [9].

$$\sigma \rightarrow_{\text{time}} \sigma' \Leftrightarrow \sigma' = \sigma \left[ \begin{array}{c} \text{clock} / \text{clock} + 1 \\ t / t - 1 \end{array} \right]$$

So, relation $\rightarrow_{\text{WaitTrans_SetDuWait}}$ is then defined by:

$$\begin{align*}
\sigma \rightarrow_{\text{WaitTrans_SetDuWait}} \sigma' & \leftrightarrow \sigma'(\text{Status}(N)) = \text{WaitTrans} \land \\
& \sigma(\text{TimerNode}(N)) = 0 \land \\
& \left( (N \in AN \land \text{Duration}(N) \neq \bot) \Rightarrow \\
& \sigma' = \sigma[\text{TimerNode}(N) / \text{Duration}(N)] \right) \land \\
& \sigma' = \sigma[\text{Status}(N) / \text{SetDuWait}] 
\end{align*}$$

The above definition says that a node $N$ becomes in state “SetDuWait” if only and only if: 1. the current state of $N$ is “WaitTrans”, 2. the value of its timer is null, 3. if $N$ is an action node for which duration is specified, then its related timer will be set to the specified duration, 4. Finally in all cases, the status of $N$ is set to SetDuWait.

Relations $\rightarrow_{\text{SetDuWait_Finish}}$. This relation models a node waiting that its associated timer, representing its possible duration, becomes null. Therefore, it is necessary only for actions that take time. This relation is specified as follows:

$$\begin{align*}
\sigma \rightarrow_{\text{SetDuWait_Finish}} \sigma' & \leftrightarrow \sigma'(\text{Status}(N)) = \text{WaitTrans} \land \\
& \sigma(\text{TimerNode}(N)) = 0 \land \sigma' = \sigma[\text{Status}(N) / \text{Finish}] 
\end{align*}$$

This relation merely states that a Node $N$ whose state is “WaitTrans” will be in state “Finish” as soon as its associated timer has a null value.

Relations $\rightarrow_{\text{Finish_Sleep}}$. To define this relation, we have to determine the set of outgoing transitions from a node $N$, and also among these transitions those that are timed.

These tow sets, denoted by $\text{OutTrans}$ and $\text{OutTimed}$ respectively, are equal to:

$$\begin{align*}
\text{OutTrans}(N) &= \text{Source}^{-1}(\{N\}) \\
\text{OutTimed}(N) &= \text{Source}^{-1}(\{N\}) - \text{EventTrans}^{-1}(\bot)
\end{align*}$$

Finally, the relation $\rightarrow_{\text{Finish_Sleep}}$ is defined by:

$$\begin{align*}
\sigma \rightarrow_{\text{Finish_Sleep}} \sigma' & \leftrightarrow \sigma'(\text{Status}(N)) = \text{Finish} \land \\
& \left[ \begin{array}{c}
\text{Status}(N) / \text{Sleep}, \\
\text{TimerNode}(\text{Target}(tr)) / (\text{Value}(\text{EventTrans}(tr))) \\
(\text{FireableTrans}(\text{OutTrans}(N)) :: \text{cand}_1)
\end{array} \right]
\end{align*}$$

where $\text{TimerNode}(\text{Target}(tr))$ denotes the timer associated with the target node of transition $tr$. Expression $(X[Y]):\text{Pred}$ means that function $X$ is modified, for the elements of $Y$, such that the image of $Y$ by $X$ satisfies predicate $\text{Pred}$. Function $X$ remains unchanged for other elements that are not in $Y$. In our case, predicate $\text{cand}_1$ is defined by:

$$\begin{align*}
\text{cand}_1 &= (N \in DN \Rightarrow \exists tr \bullet (tr \in \text{OutTrans}(N)) \land \\
& \sigma'(\text{FireableTrans}(tr)) = \sigma(\text{FireableTrans}(tr)) + 1 \land \\
& \sigma'(\text{ExecTrans}(tr)) = \sigma(\text{ExecTrans}(tr)) + 1 \land \\
& (N \notin DN \Rightarrow \forall tr \bullet (tr \in \text{OutTrans}(N)) \Rightarrow \\
& \sigma'(\text{FireableTrans}(tr)) = \sigma(\text{FireableTrans}(tr)) + 1 \land \\
& \sigma'(\text{ExecTrans}(tr)) = \sigma(\text{ExecTrans}(tr)) + 1)
\end{align*}$$

Relation $\rightarrow_{\text{Finish_Sleep}}$ says that node $N$ becomes in state “Sleep” if only and only if it is in state “Finish”. In this case, the timers of its outgoing timed transitions are set to appropriate values calculated according to function $\text{Value}$ defined by: $\text{Value} = \left\{ \begin{array}{ll}
\text{Event} \rightarrow Z \\
\text{After}(t) \rightarrow t \\
\text{When}(t) \rightarrow \sigma(\text{clock}) - t
\end{array} \right.$ Let us remark that the value returned by $\text{Value}$ is negative when the value of $\text{clock}$ is greater than the value specified for a $\text{When}$ event. After setting the timers, the node adds new fireable instances to some of its outgoing transitions according to its kind (Predicate $\text{cand}_1$). Roughly speaking, a decision node adds a new instance to only one and one of its outgoing transitions. Other nodes add a new fireable instance to each of its outgoing transitions. Finally for each transition made fireable, we increment its related $\text{ExecTrans}$ value.

4. Translation into PROMELA

In this section, we illustrate the translation of our semantics of timed activity diagrams into the PROMELA language. For the sake of the space, we present only the result of the translation on the case study. More details can be found in [16].
Translation of Var. In PROMELA, we naturally translate timers and the master clock as integer variables. Similarly, \texttt{FireableTrans} and \texttt{ExecTrans} functions are translated into integer variables. Note that all these variables are declared to be global since, as we will see, several PROMELA processes use each of them. For instance, to the transition incoming to the fork node of Figure 1, we assign the following integer variables:

```c
int Fire_Order_Fork; Exec_Order_Fork;
```

Translation of the relations $\rightarrow_{\text{time}}$. In PROMELA, the relation $\rightarrow_{\text{time}}$ is modeled with a process \texttt{Time} defined by:

```c
int clock, timer1, timer2
proctype Time(){
  atomic{
    do::timeout->{
      if ::timer1<=0 && timer2<=0)->break
      ::timer1>0 || timer2>0)->clock++
      ::else->skip
    fi;
    if ::timer1>0->timer1--; ::else->skip; fi;
    if ::timer2>0->timer2--; ::else->skip; fi}
  od }}
```

The use of the atomic keyword means that the enclosed statements are executed in a single step. This permits all the timers as well as the master clock evolving together in a synchronous way. The use of this constructor also allows improving the analysis phase by reducing its state space without affecting its result. The loop specified in the \texttt{Time} process is broken when all the values of all the timers are less than or equal to 0. This means that there is at least one of the processes waiting, either for the enabling of its incoming transitions or for its timer whose value must be equal to zero. This process states that the code enclosed in the "do" construct is repeated until all the timers are negative or null.

Execution cycle. The execution cycle of Figure 3 related to each node $A$ is translated into PROMELA as follows:

```c
active proctype Fork(){
  end:atomic{
    //Sleep
    Fire_Order_Fork >0;
    //WaitTrans
    Fire_Order_Fork--;
    //SetDuWait and finish
    timer1=2;Enable_Fork_CheckStock++;
    Exec_Fork_CheckStock++timer2=2;
    Fire_Fork_CheckCustomer++;
    Exec_Fork_CheckCustomer++;
    goto end}}
```

2. PROMELA translation of an action: the action "CheckStock" is translated into PROMELA as follows:

```c
active proctype CheckStock(){
  end:atomic{
    //Sleep
    Fire_Fork_CheckSTock >0;
    //WaitTrans
    Fire_Fork_CheckSTock--;
    timer1==0;
    //Finish
    timer1=4; timer1==0;
    Fire_CheckStockDecision1++; Exec_CheckStockDecision1++
    goto end}}
```

3. PROMELA translation of a decision: the decision node testing the quantity in stock is translated into PROMELA as follows:

```c
active proctype Decision1(){
  end:atomic{
    //Sleep:
    Fire_CheckStock_Decision1 >0;
    //WaitTrans
    Fire_CheckStock_Decision1--;
    if
      ::Fire_Decision1_MakeProductionPlan++
      Exec_Decision1_MakeProductionPlan++;
      ::Fire_Decision1_Merge++;
      Exec_Decision1_Merge++;
      fi;
    goto end}}
```

Constructor IF expresses that only one and one outgoing transition from the decision node is randomly chosen.

5. Dealing with loops

It is widely accepted that model checking is decidable for bounded models and may be undecidable for unbounded ones. In the context of our project, we are especially
interested in bounded models. In [8], we have proposed and argued that the input and output points of a loop should be a decision and a merge node respectively (See Figure 4.a). However, such an assumption does not prevent a state space explosion. Indeed the activity diagram of Figure 4.a may loop indefinitely. To tackle these problems, this kind of activity diagrams must be ruled out. In general, a loop allows specifying a repetition of some activities while a given condition is not yet satisfied. However, it is not realistic to loop indefinitely: the loop must eventually be left. This is why, for instance, we impose a limit on the number of loops $Nb\_Max$ that can be made (See Figure 4b), otherwise an exception is thrown (else branch of Figure 4.b).

![Figure 4 Case of an activity diagram with infinite state space](image)

As shown on Figure 4.b, the number of allowed loops is specified on the transition that leads to loops. Formally, we consider a function $Nb\_Max: Trans \rightarrow \mathcal{E} \cup \{1\} \cup \{\bot\}$ that specifies the maximal times a given transition can be fireable. By default (value $\bot$), a transition can be fireable at most once. Also, we make the assumption that at each decision node where a loop is possible, there is an outgoing transition labeled by else. Having done these different assumptions, the condition cand$\_i$ of the relation $\rightarrow_{Finish\_Sleep}$ related to the case of a decision node is rewritten into:

$\left[ N \in DN \land \exists (tr_1, tr_2, n) \cdot (\bigwedge_{i=1}^{2} tr_i \in Source^{-1}[\{N\}] \land \right.$

$\left. Max\_Nb(tr_1) = n \land label(tr_2) = else \right] \Rightarrow$

$\left[ \exists tr \cdot (tr \in OutTrans(N) \land tr \neq tr_1 \land \right.$

$\sigma^r(FireableTrans(tr)) = \sigma(FireableTrans(tr)) + 1 \land$

$\sigma^r(ExecTrans(tr)) = \sigma(ExecTrans(tr)) + 1 \land$

$\sigma(ExecTrans(tr)) \geq n \Rightarrow$

$\exists tr \cdot (tr \in OutTrans(N) \land tr \neq tr_2 \land$$\sigma^r(FireableTrans(tr)) =$$\sigma(FireableTrans(tr)) + 1 \land$

$\sigma^r(ExecTrans(tr)) = \sigma(ExecTrans(tr)) + 1 \land$

This last formula says that at the decision node, the possible transition leading to loop would be made fireable only if the maximal number of allowed loops ($Max\_NB$) is not attained yet. In opposite, the transition labeled with "else" would be made active only if $Max\_NB$ is reached.

6. Conclusion

In this paper, we have presented a formal syntax and semantics for UML activity diagrams endowed with times aspects. A systematic way for translating this semantics into the PROMELA language is also provided. Such a translation is not an end in itself, it is a basis for a formal and automatic verification of wide range of constrains including liveness, safety and real time properties that ensure a confidence level for e-business processes.

Currently, we are working on the implementation of this approach. The tool, whose architecture is illustrated by Figure 5, consists of the following steps:
1. Translating an activity diagram into PROMELA language.
2. Selecting a property from a pre-defined list
3. Running the SPIN tool, if the verification fails the path of the activity diagram causing the error is highlighted.

Further work includes the definition of more elaborated translation rules into PROMELA that would reduce the state space of the system in order to improve the performance of SPIN: SPIN has generated 40 states for the running example. The state space reduction can be done, for instance, by using other PROMELA constructors like \texttt{d_step}. In fact, although the PROMELA statements that can be included inside a \texttt{d_step} sequence are rather restricted, however compared to \texttt{atomic}, this sequence is more efficient during verifications.

7. REFERENCES

[18] NuSMV. \texttt{http://nusmv.irst.itc.it/}.