



Pre-Order Semantics of UML State-Machines

Yunming Wang, Jean-Pierre Talpin, Albert Benveniste, Paul Le Guernic

► To cite this version:

Yunming Wang, Jean-Pierre Talpin, Albert Benveniste, Paul Le Guernic. Pre-Order Semantics of UML State-Machines. [Research Report] RR-3958, INRIA. 2000. inria-00072690

HAL Id: inria-00072690

<https://inria.hal.science/inria-00072690>

Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Pre-Order Semantics of UML State-Machines

Yunming Wang — Jean-Pierre Talpin — Albert Benveniste — Paul Le Guernic

N° 3958

June 2000

____ THÈME 1 ____

 *apport
de recherche*


Pre-Order Semantics of UML State-Machines

Yunming Wang , Jean-Pierre Talpin , Albert Benveniste , Paul Le Guernic

Thème 1 — Réseaux et systèmes
Projet Ep-Atr

Rapport de recherche n° 3958 — June 2000 — 32 pages

Abstract: The concept of synchronous programming has been proposed and widely accepted in the design of real-time systems, circuits, and embedded systems. Some recent researches have also proposed a mechanism to distribute a synchronous system over asynchronous networks. Meanwhile, UML is also becoming a standard framework of object-oriented methodologies.

Our research aims to take advantage of these rich backgrounds by integrating a synchronous pivot called SPOTS (synchronous pre-order transition system) into UML and to propose a new methodology for the development of real-time distributed systems.

In this paper, we focus on the issue of UML state-machines. We first present a recursive structure of UML state-machines. Compared with earlier studies, this structure supports composite transitions and histories. After a brief introduction to the pivot SPOTS, we will concentrate on the formal semantics of UML state machine by translating it in SPOTS. We will also give some complete examples of the translation according to the prototype we have implemented.

Key-words: Uml, state-machine, statechart, semantics, Spots

La sémantique pré-ordonnée des state-machines de UML

Résumé : Le concept de la programmation synchrone a été proposé et largement accepté dans le développement des systèmes en temps réel, des circuits, et des systèmes inclus. De récentes recherches ont également proposé un mécanisme pour distribuer un système synchrone au-dessus des réseaux asynchrones. En attendant, UML est également devenu un cadre standard des méthodologies orienté objet.

Notre objectif de recherches est de profiter de ces avantages en intégrant un pivot synchrone appelé SPOTS (système synchrone de transition pré-ordonnée) dans UML et de proposer une nouvelle méthodologie pour le développement des systèmes réactifs distribués.

Dans cet article, nous nous intéressons au problème de state-machines de UML. Nous présentons d'abord une structure récursive de state-machines de UML. Comparée à des études précédents, cette structure maintient des transitions composées et des histoires. Après une brève introduction au pivot SPOTS, nous nous concentrons sur la sémantique formelle des state-machines de UML en les traduisant en SPOTS. Nous allons également démontrer des exemples complets de la traduction d'après le prototype que nous avons implémenté dans notre équipe.

Mots-clés : Uml, state-machine, statechart, semantique, Spots

1 Introduction

A methodical approach to the development of distributed software stages from preliminary requirement specifications, design phases, deployment verification down to code generation. Such design principles are present in standard object-oriented notations such as the UML. In such notations, development is supported by an arrangement of informal notations, bringing different and complementary views of a system: class diagrams (the essence of structuring), deployment diagrams (to describe architectures), sequence diagrams (to specify global abstractions) and state diagrams (to locally represent behaviors).

However, a mathematically well-founded method for the development of distributed software can hardly grasp the semantics of a system out of compromised and orthogonal notations. In contrast, it should consist of a simple and minimal set of mathematical concepts used during all the development process. Most formal description techniques designed for the engineering of distributed systems and, in particular, those incorporated into the UML, cannot easily be used in an integrated fashion. They have specific semantic models, suited for a particular step of the development.

The lack of integration between the different views in standard object-oriented formalisms has recently motivated extensions of conventional formalisms, such as the live sequence-charts of [5]. Our aim is to demonstrate the benefit of using a pivot formalism, called BDL [8, 9], for an integrated development of distributed reactive software. This model addresses the key issues raised in the development of distributed reactive systems that none of the existing formalisms yet fully integrate. Namely, the early or partial designs of system specifications; inheritance, refinement and reuse of system descriptions; architecture modeling and deployment.

State-machines form an important and sophisticated component of the UML framework. They play an essential role in the behavioral description of a system. Although, an informal or intuitive semantics for state-machines may be helpful for users to capture key intuitions, it is of interests to build a formal semantics in such a way that verification, code generation, optimization, and deployment can be easily applied to the system. Since the definition of STATECHART [3], numerous contributions have been devoted to providing a semantics for this formalism. Some use automata or hierarchical automata [6], graph expansion [2], labeled transition system [11], process algebras [10].

However, most of these proposals usually study an abstract sub-set of STATECHARTS. For instance, most proposals do not take the notion of *history* into consideration, or do not consider *join*, *fork* connectors. In most cases, available models only consider transitions which have one source state and one target state. Moreover, the source state and the target state are most of the time regarded as sub-states of an or-state (i.e. transitions are only allowed at the same level). Such numerous restrictions do not fit within real-sized application-oriented projects using STATECHART.

BDL [8, 9] has been proposed as a pivot formalism for describing behavioral system specifications and has already proved to be an expressive formalism, by allowing the encoding of message sequence charts, synchronous languages (such as SIGNAL [1]) and other calculi. In BDL, different views of a system can be captured within a simple mathematical model consisting of synchronous transition systems equipped with pre-ordered scheduling

relations (SPOTS). Key properties for a correct deployment of synchronous specifications over asynchronous networks, namely *endochrony* and *isochrony*, have been specified in this structure [9].

Our aim is here to provide a translation and formal semantics of UML state-machines in terms of the synchronous pre-order transition systems of BDL. In section 2, we define the structure of UML state-machines. Section 3 is devoted to a brief introduction of the BDL calculus. The focus of this paper is the definition of a formal semantics of UML state-machines in terms of BDL pre-orders, section 4. Some complete examples of the translation from state-machines to SPOTS are also given in Appendix A, according to the prototype we have implemented.

2 UML state-machines

Recursive structure

We recursively define the structure of UML state-machines as:

- A base-state $\text{base}(b)$ is a state-machine of type **base**.
- The structure $s = \text{and}(\langle s_i \rangle_{i=1}^n)$ is a state-machine of type **and** iff all $s_i, i = 1, ..n$ are state-machines.
- The structure $s = \text{or}(\langle s_i \rangle_{i=1}^n, H, D, T)$ is a state-machine of type **or** iff all $s_i, i = 1, ..n$ are state-machines. s_1 is the initial sub-state-machine, H is the history node, D is the deep-history node, and T is a set of *detailed transitions* at the level of s . This will be defined later in this section.

Substates and history

We write $\text{type}(s)$ for the type of state-machine s , $\mathcal{H}(s)$ its history and $\mathcal{D}(s)$ its deep-history. Given a state-machine s , we define $\mathcal{S}(s)$ to be the set of sub-state-machines of s ; $\mathcal{S}^*(s)$ as the set of descendants including itself; and $\mathcal{S}^+(s)$ as the set of descendants excluding itself. Accordingly, we also define $\mathcal{H}^*(s)$ as the union of $\mathcal{S}^*(s)$ and all of their history nodes and deep-history nodes; $\mathcal{H}^+(s)$ as the union of $\mathcal{S}^+(s)$ and their (deep) history nodes. Their formal definitions are given in Figure 1.

Orthogonality

Two state-machine $s' \neq s''$ are called *orthogonal* if they are in two different sub-state-machines of an and-state-machine. Formally, s_1 and s_2 are *orthogonal* iff there exists an and-state s and $s'_1 \neq s'_2$ in $\mathcal{S}(s)$ s.t. $s_1 \in \mathcal{H}^*(s'_1)$ and $s_2 \in \mathcal{H}^*(s'_2)$.

Transitions

A *transition* (Sr, Tg, l) consists of a *label* l and pair-wise orthogonal sets of *source* states Sr and *target* states Tg (and history nodes). The source nodes of a transition can *only* be states.

$$\mathcal{S}(\text{base}(b)) = \emptyset$$

$$\mathcal{S}(\text{and}(\langle s_i \rangle_{i=1}^n)) = \mathcal{S}(\text{or}(\langle s_i \rangle_{i=1}^n, _)) = \{s_i, i = 1, n\}$$

$$\mathcal{S}^*(s) = \mathcal{S}^+(s) \cup \{s\}$$

$$\mathcal{S}^+(s) = \mathcal{S}(s) \bigcup_{s_i \in \mathcal{S}(s)} \mathcal{S}^+(s_i)$$

$$\mathcal{H}^*(s) = \mathcal{S}^*(s) \bigcup_{\substack{\text{type}(s')=\text{or} \\ s' \in \mathcal{S}^*(s)}} \{\mathcal{H}(s'), \mathcal{D}(s')\}$$

$$\mathcal{H}^+(s) = \mathcal{S}^+(s) \bigcup_{\substack{\text{type}(s')=\text{or} \\ s' \in \mathcal{S}^+(s)}} \{\mathcal{H}(s'), \mathcal{D}(s')\}$$

Figure 1: Substates and history of a state-machine

Target nodes can be either states or history nodes. A label is composed of an *event* E, a *guard* G or an *action* A.

Generally speaking, a *state* denotes the abstraction (the name) of a state-machine. Wherever no confusion may arise, we will use the term *state* in place of *state-machine*.

Detailed transitions

In an or-state-machine, a transition allows to shift from one sub-state to another. The or-state is called the level of the transition and these sub-states its main source and main target.

From another point of view, when a transition takes effect, it exits a set of states, enters another set of states, and stays in yet another. Hence, the level of the transition is the lowest state the system will stay in; the main source is the highest state the transition will exit; and the main target is the highest state the transition will enter.

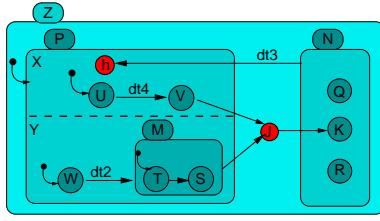
Given a transition $(\text{Sr}, \text{Tg}, l)$, its level s is the lowest common ancestor of $\text{Sr} \cup \text{Tg}$ such that $\text{type}(s) = \text{or}$; its *main source* is the sub-state s' of s satisfying $\text{Sr} \subseteq \mathcal{S}^*(s')$; and its *main target* is the sub-state s'' s.t. $\text{Tg} \subseteq \mathcal{H}^*(s'')$.

A *detailed transition* of a state-machine $s = \text{or}(\langle s_i \rangle_{i=1}^n, H, D, T)$ is a tuple $(s_{\text{out}}, s_{\text{in}}, \text{Sr}, \text{Tg}, l) \in T$ consisting of a transition $(\text{Sr}, \text{Tg}, l)$ at level s , a main source s_{out} and a main target s_{in} .

Example 1 For instance, the following UML state-machine consists of a top or-state-machine Z composed of two sub-state-machines P and N . P is an and-state-machine and U, V are base-state-machine. X has history node h and Y has deep history node d . According to previous definitions, it will be represented as depicted on the right of figure 2.

Here, $\mathcal{S}(Y) = \{W, M\}$, $\mathcal{S}^*(Y) = \{W, M, T, S\}$, and $\mathcal{H}^*(P) = \{X, Y, U, V, h, W, M, T, S\}$. U and W are orthogonal, so are V and M , but K and W are not.

Transition dt_1 makes the state-machine Z exit sub-state P and enter sub-state N , so its level is Z , its main source P and its main target N .



$Z = \text{or}(\langle P, N \rangle, \text{nil}, \text{nil}, \{dt_1, dt_3\})$
 $P = \text{and}(\langle X, Y \rangle)$
 $X = \text{or}(\langle U, V \rangle, h, \text{nil}, \{dt_4\})$
 $Y = \text{or}(\langle W, M \rangle, \text{nil}, d, \{dt_2\})$
 $dt_3 = (N, P, \{N\}, \{h\}, E_3[G_3]/A_3)$
 $dt_1 = (P, N, \{V, M\}, \{K\}, E_1[G_1]/A_1)$
 \dots

Figure 2: A state-machine and its representation

Actions

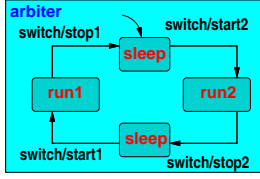
Each transition has an action. A state-machine S may also be equipped with three actions: $A_{\text{stay}}(S)$, $A_{\text{out}}(S)$, $A_{\text{in}}(S)$, which are executed when the system stays, exits, enters the state-machine respectively. Finally, two transitions are conflicting if they may exit some common state(s).

3 An introduction to SPOTS

BDL [8, 9] consists of symbolic transition systems in which transitions relate labeled pre-orders specified via directed graphs (so-called SPOTS or synchronous pre-order transition systems).

Example 2 *Let us for instance consider the arbiter between two tasks run1 and run2 and observe the correspondence between automata and pre-orders. In the arbiter, four state-transitions are specified in reaction to the input message switch. Each of them incurs a state-transition (from a run mode to a sleep mode) and the occurrence of an output message (a start or a stop command to the modulo counters).*

A pre-order transition system represents this structure by decomposing it into atomic reactions γ , and by rendering the behavior of the system as a choice of a reaction γ within a set Γ , depending on the initial state (the π^-) and on the input events. For instance, the transition from sleep to run2 state requires the input switch and produces the output start2. The term τ denotes a silent transition.



$$\Gamma_{\text{arbiter}} = \tau$$

$$\begin{aligned} & \vee (\pi^-(\text{sleep}) \rightarrow \pi^+(\text{run2}) \parallel \text{switch} \rightarrow \text{start2}) : \gamma_1 \\ & \vee (\pi^-(\text{run2}) \rightarrow \pi^+(\text{sleep}) \parallel \text{switch} \rightarrow \text{stop2}) : \gamma_2 \\ & \vee (\pi^-(\text{sleep}) \rightarrow \pi^+(\text{run1}) \parallel \text{switch} \rightarrow \text{start1}) : \gamma_3 \\ & \vee (\pi^-(\text{run1}) \rightarrow \pi^+(\text{sleep}) \parallel \text{switch} \rightarrow \text{stop1}) : \gamma_4 \end{aligned}$$

Figure 3: The arbiter and its translation

Name-spaces

Synchronous pre-order transition systems (SPOTS) are defined over five name-spaces that account for ports $x \in X$, variables $v \in V$, pins $\pi \in \Pi$, constants $c \in C$ and operations $f \in F \supset \Sigma_{n \geq 1}(C^n \rightarrow C)$. An expression e , of value in C , can be a name u (a constant c or a variable v) or an operation $f(e_1, \dots, e_n)$. Messages $x(e)$ are represented as associations of expressions e to ports x . Pins $\pi \in \Pi$ are used to represent initial states $\pi^-(e)$ and final states $\pi^+(e)$ of the transition system.

Graphs

A graph γ consists of messages $x(e)$ and of reflexive-transitive causality relations $\gamma \rightarrow \gamma'$ of minima in $\Pi^- \times C$ and maxima in $\Pi^+ \times C$. The composition $\gamma \parallel \gamma'$ of graphs is defined by the union of the graphs γ and γ' iff, for all $x(e) \in \gamma$ and $x(e') \in \gamma'$, $e = e'$. The sequence $\gamma \rightarrow \gamma'$ is defined by the composition of γ , of γ' and of causality relations $x(e) \rightarrow x'(e')$ for all $x(e) \in \gamma$ and $x'(e') \in \gamma'$.

$\gamma ::= \tau$	(silence)
$x(e)$	(message)
$\pi^-(e)$	(pre-condition)
$\pi^+(e)$	(post-condition)
$\gamma \rightarrow \gamma'$	(sequence)
$\gamma \parallel \gamma'$	(composition)

Figure 4: Graphs γ

Family of graphs

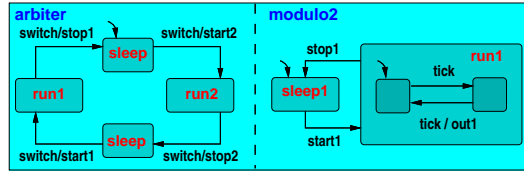
A family Γ is a set of graphs γ assembled by the choice operator \vee . We write $\gamma \in \Gamma$ for the membership of γ to Γ iff $\Gamma = \gamma \vee \Gamma'$. The composition $\Gamma \parallel \Gamma'$ of two families is defined in terms of the choice operator by considering all pairs of graphs (γ, γ') from (Γ, Γ') in which

ports are either simultaneously absent from γ and γ' or simultaneously present in γ and γ' with the same value. We write $\Gamma \wedge \Gamma' = (\Gamma \setminus \tau) \parallel (\Gamma' \setminus \tau)$ for the synchronization of Γ and Γ' .

$\Gamma ::= \gamma$	(reaction)
$\Gamma \vee \Gamma'$	(choice)
$\Gamma \parallel \Gamma'$	(composition)
$\Gamma \wedge \Gamma'$	(synchronization)

Figure 5: Families Γ

Example 3 *Let us now consider the synchronous composition of the arbiter (example 2) with a modulo counter. The specification of the modulo counter is rendered by a family consisting of three reactions $\gamma_{a,b,c}$ plus τ . The family Γ_{run1} denotes the behavior of the sub-state run1.*



$$\begin{aligned} \Gamma_{\text{modulo2}} = & \tau \vee (\pi^-(\text{sleep1}) \rightarrow \pi^+(\text{run1}) \parallel \text{start1}) : \gamma_a \\ & \vee (\pi^-(\text{run1}) \rightarrow \pi^+(\text{sleep1}) \parallel \text{stop1}) : \gamma_b \\ & \vee (\pi^-(\text{run1}) \rightarrow \pi^+(\text{run1}) \parallel \Gamma_{\text{run1}}) : \gamma_c \end{aligned}$$

Figure 6: A counter modulo 2 and its translation

*In the composed family $\Gamma_{\text{arbiter}} \parallel \Gamma_{\text{modulo2}}$, γ_1 , γ_2 and γ_c are independent, whereas γ_3 shares the event **start1** with γ_a and γ_4 the event **stop1** with γ_b .*

$$\Gamma_{\text{arbiter}} \parallel \Gamma_{\text{modulo2}} = \tau \vee (\gamma_3 \parallel \gamma_a) \vee (\gamma_4 \parallel \gamma_b) \vee (\gamma_1 \vee \gamma_2 \vee \gamma_c \vee (\gamma_1 \parallel \gamma_c) \vee (\gamma_2 \parallel \gamma_c))$$

Synchronous composition

Let us write $x \in X(\gamma)$ and $\pi \in \Pi(\gamma)$ for the finite sets of *ports* and *pins* of a graph γ . Two graphs $\gamma \in \Gamma$ and $\gamma' \in \Gamma'$ are *composable*, denoted $\gamma \bowtie \gamma'$, iff any vertex $x(c)$ (resp. $\pi^-(c)$, $\pi^+(c)$) which satisfies $x \in X(\Gamma) \cap X(\Gamma')$ (resp. $\pi \in \Pi(\Gamma) \cap \Pi(\Gamma')$) occurs in γ iff it occurs in γ' . Then,

$$\Gamma \parallel \Gamma' = \bigvee_{\gamma \in \Gamma, \gamma' \in \Gamma', \gamma \bowtie \gamma'} (\gamma \parallel \gamma')$$

Quantifiers and guards

In a smooth move from a mathematical model towards a full-fledged specification language, some useful derived operators can be defined, such as the enumeration $\forall v.\Gamma$ or the restriction $\exists x.\Gamma$ (of the scope of the port x to the family Γ). Let us write D_x and D_v for the domain of a port x and of a variable v . The universal quantification $\forall v.\Gamma$ defines a variable v which generically models the set of families $\Gamma[c/v]$ corresponding to the reactions to all possible values c of the variable v . Hence, $\forall v.\Gamma = \tau \vee_{c \in D_v} \Gamma[c/v]$. Guards are modeled by assertion ports α whose domain are restricted to $D_\alpha = \{\# \}$. A message $\alpha(e)$ defines an assertion as it carries a boolean expression e which can only be present if the value of e is true.

Example 4 *In the SPOTS, enumeration and guard allow to generically define the set of reactions of infinite state systems, e.g. a modulo4 counter, by defining a default behavior, consisting of the transition from $\pi^-(m)$ to $\pi^+(n)$ under the condition $n = (m+1) \bmod 4$ and upon receipt of a tick event, and letting it inherit, by synchronous composition, the behavior of emitting out1 when the count n is reset to 0.*

$$\Gamma_{\text{run1}} = \tau \vee \forall b, b'. \left(\left(\begin{array}{c} \text{tick} \parallel \pi^-(b) \rightarrow \pi^+(b') \\ \parallel \alpha(b' = \neg b) \end{array} \right) \parallel \left(\begin{array}{c} (\alpha(b' = \#) \parallel \text{tick} \rightarrow \text{out1}) \\ \vee (\alpha(b' = \#) \parallel \text{tick}) \end{array} \right) \right)$$

Concatenation and semantics

The construction of the synchronous trace of a family Γ consists of the concatenation of appropriate pre-orders chosen in that family. Each concatenated pre-order hence defines an elementary pattern that models an atomic reaction. Two well-formed graphs γ, γ' are *concatenable* (denoted $\gamma \odot \gamma'$) iff, $\forall \pi \in \Pi(\gamma) \cup \Pi(\gamma'), \pi^+(c) \in \max(\gamma) \wedge \pi^-(c') \in \min(\gamma') \Rightarrow c = c'$.

The synchronous concatenation $\gamma \bullet \gamma'$ of two concatenable graphs γ and γ' is obtained by first identifying $\pi^+(c) \in \max(\gamma)$ and $\pi^-(c) \in \min(\gamma')$ as $\pi(c)$ and then by taking the disjoint union $\gamma \uplus \gamma'$. Thus $\Gamma \bullet \Gamma' = \{\gamma \bullet \gamma' \mid \gamma \in \Gamma, \gamma' \in \Gamma', \gamma \odot \gamma'\}$, and the co-iterative application of the concatenation operator \bullet defines the synchronous traces $\Gamma_0 \bullet (\Gamma)^\omega$ of a process whose initial state (resp. behavior) is described by Γ_0 (resp. Γ).

4 Pre-order semantics of state-machines

4.1 Preliminary observation

In order to define a semantics of UML state-machines using SPOTS, we first define the *reduction* and the *frame* of an or-state-machine, and then study their semantics relation.

A *clean* state-machine $\mathcal{C}(s)$ is a state-machine obtained from s by removing all of its transitions. The *reduction* $\mathcal{R}(s)$ of an or-state-machine $s = \text{or}(\langle S_i \rangle_{i=1}^n, H, D, T)$ consists of its top level transitions only, as opposed to its *frame* $\mathcal{F}(s)$, consisting of lower level transitions only.

Figure 8 gives an or-state-machine, its frame and its reduction. In the frame, we can see there is no transfer of control between sub-states. But transitions inside the active

$$\mathcal{C}(s) = \begin{cases} \text{or}(\langle \mathcal{C}(s_i) \rangle_{i=1}^n, H, D, \emptyset) & \text{if } s = \text{or}(\langle s_i \rangle_{i=1}^n, H, D, T) \\ \text{and}(\langle \mathcal{C}(s_i) \rangle_{i=1}^n) & \text{if } s = \text{and}(\langle s_i \rangle_{i=1}^n) \\ s & \text{otherwise} \end{cases}$$

$$\begin{aligned} \mathcal{R}(s) &= \text{or}(\langle \mathcal{C}(s_i) \rangle_{i=1}^n, H, D, T) \\ \mathcal{F}(s) &= \text{or}(\langle s_i \rangle_{i=1}^n, H, D, \emptyset) \end{aligned}$$

Figure 7: Frame and reduction of an or-state-machine

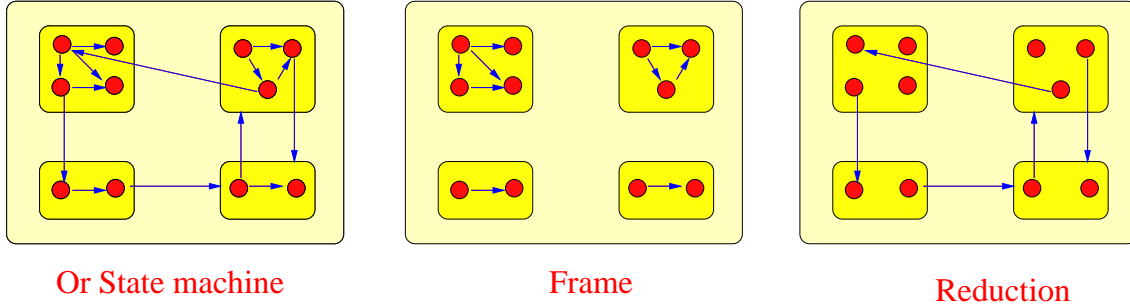


Figure 8: An or-state-machine, its frame and reduction

sub-state can still be invoked when necessary. So, the frame describes the behavior of an or-state-machine when there is no control transfer between its (direct) sub-states.

As a complement, in the reduction, each transition will lead to a control transfer between its sub-states. So, it describes the behavior of an or-state-machine when it has such control transfer. Note that, due to conflicting transitions, there can be at most one transition between sub-state-machines invoked in one step. Furthermore, whenever there is a transition between sub-state-machines, no transition inside its sub-state-machine can be invoked. As a result, the behavior of an or-state-machine is a choice between its frame and its reduction.

4.2 Basic translation

We now give a formal semantics of UML state-machines by translating it into BDL's synchronous pre-order transition systems. We will first provide a basic translation, without any consideration of actions attached to states, nor priorities of transitions. In section 4.3 and 4.4, we will discuss these issues.

Pins and initial graph

As we have already stated, for an or-state, if it is active, one and only one of its sub-states is active. So, we will create a pair of pins h_S^\pm (a start pin and an exit pin) in SPOTS for each or-state s to record the active sub-state. $h_S^-(s_i)$ means the active sub-state of s was s_i at

last time, and $h_S^+(s_i)$ means it will be in s_i after this step. We will also use a pin generator $\mathcal{P}^\pm(s)$ as follows, where $f(s)$ denotes the father of s . (s is a sub-state of $f(s)$.)

$$\mathcal{P}^\pm(s) = \begin{cases} \{h_{f(s)}^\pm(s)\} & \text{if } \text{type}(f(s)) = \text{or} \\ \emptyset & \text{if } \text{type}(f(s)) = \text{and} \end{cases}$$

Since the initial state of $S = \text{or}(\langle s_i \rangle_{i=1}^n, H, D, T)$ is s_1 , we have $\mathcal{P}^+(s_1)$ in the initial graph of SPOTS, which is actually $\{h_S^+(s_1)\}$.

Structural translation

We now define a structural translation from state-machines to pre-orders transition systems by means of the map \mathcal{T} .

- The semantics of a base state-machine $\text{base}(b)$ is silent. i.e.

$$\mathcal{T}(\text{base}(b)) = \tau$$

- The semantics of a state-machine $S = \text{and}(\langle s_i \rangle_{i=1}^n)$ is the synchronous composition of its sub-state-machines, i.e.

$$\mathcal{T}(S) = \parallel_{i=1}^n \mathcal{T}(s_i)$$

- The semantics of an or-state-machine S is a choice between its frame and its reduction:

$$\mathcal{T}(S) = \mathcal{T}(\mathcal{R}(S)) \bigvee \mathcal{T}(\mathcal{F}(S))$$

- The semantics of the reduction of an or-state-machine $S = \text{or}(\langle s_i \rangle_{i=1}^n, H, D, T)$ is a choice among all of its transitions:

$$\mathcal{T}(\mathcal{R}(S)) = \bigvee_{dt \in T} \mathcal{T}(dt)$$

- The frame of an or-state-machine $S = \text{or}(\langle s_i \rangle_{i=1}^n, H, D, T)$ is translated into the sub-state-machine that was active:

$$\mathcal{T}(\mathcal{F}(S)) = h_S^-(v) \triangleright \left(\bigvee_{i=1}^n (\alpha(v = s_i) \wedge \mathcal{T}(s_i) \rightarrow h_S^+(s_i)) \right)$$

Where we use the notion “ \triangleright ” for hierarchy as stated in [9]. Here is the explanation of this formula:

- $h_S^-(v)$: suppose S was previously in sub-state v
- $\alpha(v = s_i)$: if v is s_i
- $\mathcal{T}(s_i)$: then its behavior is $\mathcal{T}(s_i)$
- $h_S^+(s_i)$: as a result, S will still stay in the s_i

Figure 9 gives a summary of these formulas.

$$\begin{aligned}
\mathcal{T}(\text{base}(b)) &= \tau \\
\mathcal{T}(S = \text{and}(\langle S_i \rangle_{i=1}^n)) &= \parallel_{i=1}^n \mathcal{T}(S_i) \\
\mathcal{T}(S = \text{or}(\langle S_i \rangle_{i=1}^n, H, D, T)) &= \mathcal{T}(\mathcal{R}(S)) \bigvee \mathcal{T}(\mathcal{F}(S)) \\
\mathcal{T}(\mathcal{F}(S)) &= h_S^-(v) \triangleright \left(\bigvee_{i=1}^n (\alpha(v = S_i) \wedge \mathcal{T}(S_i) \rightarrow h_S^+(S_i)) \right) \\
\mathcal{T}(\mathcal{R}(S)) &= \bigvee_{dt \in T} \mathcal{T}(dt)
\end{aligned}$$

Figure 9: Structural translation

Translation of detailed transitions

Consider a detailed transition in general: $dt = (S_{\text{out}}, S_{\text{in}}, Sr, Tg, E[G]/A)$. We decompose its translation into three parts:

$$\mathcal{T}(dt) = \left(\begin{array}{c} (\mathcal{T}^-(Sr, S_{\text{out}}) \parallel \mathcal{T}(E) \parallel \mathcal{T}(G)) \\ \downarrow \\ \mathcal{T}(A) \\ \downarrow \\ \mathcal{T}^+(Tg, S_{\text{in}}) \end{array} \right) \begin{array}{l} : \text{When is it enabled?} \\ : \text{What to do if it is activated?} \\ : \text{What's the result configuration?} \end{array}$$

Figure 10: Translation of detailed transitions

Translation of start pins

Given a set of pairwise orthogonal states Sr , $Sr \subseteq \mathcal{S}^*(S_{\text{out}})$, $\mathcal{T}^-(Sr, S_{\text{out}})$ indicates a set of start pins ensuring the system in states Sr .

Let's consider this way: The transition will be enabled only if the system is in S_{out} . Additionally, we may have some “further information” that the system must also be in states Sr if it is enabled. We will define $\mathcal{T}^-(Sr, S_{\text{out}})$ recursively by case analysis on this “further information” and the type of S_{out} . This is illustrated in Table 1.

Translation of exit pins

The exit pins represent the result configuration of the state-machine after a step. When a transition $dt = (S_{\text{out}}, S_{\text{in}}, Sr, Tg, E[G]/A)$ takes effect, the system are supposed to enter its main target S_{in} , and similarly, we have “further indication” that the system must enter Tg as

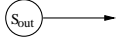
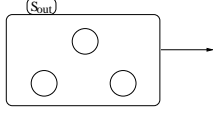
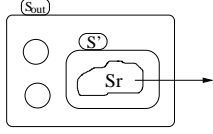
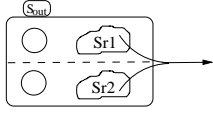
<i>transition</i>	$\mathcal{T}^-(\text{Sr}, s_{\text{out}})$
	if $\text{type}(s_{\text{out}}) = \text{base}$ $\mathcal{T}^-(\text{Sr}, s_{\text{out}}) = \mathcal{P}^-(s_{\text{out}})$
	else if $\text{type}(s_{\text{out}}) = \text{or}$, $\text{Sr} = \{s_{\text{out}}\}$ or $\text{Sr} = \emptyset$ /* no further information in Sr */ $\mathcal{T}^-(\text{Sr}, s_{\text{out}}) = \mathcal{P}^-(s_{\text{out}})$
	else if $\text{type}(s_{\text{out}}) = \text{or}$, $\exists s' \in \mathcal{S}(s_{\text{out}})$, $\text{Sr} \subseteq \mathcal{S}^*(s')$ /* Sr is in sub-state s' */ $\mathcal{T}^-(\text{Sr}, s_{\text{out}}) = \mathcal{P}^-(s_{\text{out}}) \cup \mathcal{T}^-(\text{Sr}, s')$
	else /* $\text{type}(s_{\text{out}}) = \text{and}$ */ $\mathcal{T}^-(\text{Sr}, s_{\text{out}}) = \mathcal{P}^-(s_{\text{out}}) \bigcup_{s' \in \mathcal{S}(s_{\text{out}})} \mathcal{T}^-(\text{Sr} \cap \mathcal{S}^*(s'), s')$

Table 1: Translation of start pins

well. The translation $\mathcal{T}^+(\text{Tg}, s_{\text{in}})$ is based on a case analysis upon these “further indications” and the type of s_{in} . Its definition is given in Table 2.

Example 5 Consider the detailed transition $\text{dt}_1 = (P, N, \{V, M\}, \{K\}, e1[g1]/a1)$ of example 1. It takes effect only if the system is in state V and M , so its start pins will be $\{h_Z^-(P), h_X^-(V), h_Y^-(M)\}$. After this transition, the system will be in state K , so the exit pins will be $\{h_Z^+(N), h_N^+(K)\}$.

Translation of events, guards, and actions

At this stage, we have presented the structural translation. We will now consider the translation of basic ingredients: events E , guards G and actions A . Let us first recall these concepts as defined in UML.

- Event

There are four kinds of events E in UML: signal events, call events, time events, and change events. An event E consists of an event name id and a list (may be empty) of parameters $e_{1..n}$. These parameters may be used in actions.

- Guard

A guard G is a boolean expression $f_b(e_{i=1..n})$ attached to a transitions as a fine-grained control over its firing. It should be a pure expression without side-effects. It is evaluated when an event is dispatched by the state machine. If the guard is true at that time, the transition is enabled, otherwise, it is disabled.

<i>transition</i>	$\mathcal{T}^+(\text{Tg}, s_{\text{in}})$
	if $\text{type}(s_{\text{in}}) = \text{base}$ $\mathcal{T}^+(\text{Tg}, s_{\text{in}}) = \mathcal{P}^+(s_{\text{in}})$
	else if $\text{type}(s_{\text{in}}) = \text{or}$, $\text{Tg} = \emptyset$ or $\text{Tg} = \{s_{\text{in}}\}$ /* no further information in Tg */ $\mathcal{T}^+(\text{Tg}, s_{\text{in}}) = \mathcal{P}^+(s_{\text{in}}) \cup \mathcal{T}^+(\emptyset, s_1)$
	else if $\text{type}(s_{\text{in}}) = \text{or}$, $\text{Tg} = \{\mathcal{H}(s_{\text{in}})\}$ /* Enter the history */ $\mathcal{T}^+(\text{Tg}, s_{\text{in}}) = \mathcal{P}^+(s_{\text{in}}) \cup h_{s_{\text{in}}}^-(v) \triangleright \left(\bigvee_{s' \in \mathcal{S}(s_{\text{in}})} (\alpha(v = s') \rightarrow \mathcal{T}^+(\emptyset, s')) \right)$
	else if $\text{type}(s_{\text{in}}) = \text{or}$, $\text{Tg} = \{\mathcal{D}(s_{\text{in}})\}$ /* Enter the deep-history */ $\mathcal{T}^+(\text{Tg}, s_{\text{in}}) = \mathcal{P}^+(s_{\text{in}}) \cup h_{s_{\text{in}}}^-(v) \triangleright \left(\bigvee_{s' \in \mathcal{S}(s_{\text{in}})} (\alpha(v = s') \rightarrow \mathcal{T}^+(\mathcal{D}(s'), s')) \right)$
	else if $\text{type}(s_{\text{in}}) = \text{or}$, $\exists s' \in \mathcal{S}(s_{\text{in}}), \text{Tg} \subseteq \mathcal{H}^*(s')$ /* Tg is in s' */ $\mathcal{T}^+(\text{Tg}, s_{\text{in}}) = \mathcal{P}^+(s_{\text{in}}) \cup \mathcal{T}^+(\text{Tg}, s')$
	else /* $\text{type}(s_{\text{in}}) = \text{and}$ */ $\mathcal{T}^+(\text{Tg}, s_{\text{in}}) = \mathcal{P}^+(s_{\text{in}}) \cup_{s' \in \mathcal{S}(s_{\text{in}})} \mathcal{T}^+(\text{Tg} \cap \mathcal{H}^*(s'), s')$

Table 2: Translation of exit pins

$$\begin{aligned}
v &::= id|id\$ \\
e &::= v|c|f(e_{i=1..n}) \\
E &::= id|id()|id(e_{i=1..n}) \\
G &::= f_b(e_{i=1..n}) \\
A &::= (id := e)|(emit E)
\end{aligned}$$

Figure 11: Grammar of events, guards and actions

- Action

An action A is the specification of an executable statement. It may be written in terms of operations, attributes, links of the owning object, the parameters of the triggering event, and any other features visible in its scope. In the definition of UML [7] the script of action is specified in *ActionExpression*, which may be a sequence comprising a number of distinct actions, including actions that explicitly generate events. However, the details of *ActionExpression* depend on the action language chosen for the model. We consider two kinds of actions below: assignment and emission of event.

We propose their grammar as shown in the Figure 11, where notion $id\$$ is used for the previous value of id , and f_b is a boolean function without side-effect.

We have to distinguish the name spaces of state-machine and that of SPOTS before the presentation of their translations. A careful study will illustrate that a variable in state-machine is quite different from that in SPOTS. As an example, the action $x := y\$ + z$ will be translated into $(y^-(u), z^+(v)) \rightarrow x^+(u + v)$ in two steps:

1. Associate the value of $y\$$ to variable u (in SPOTS) by start pin $y^-(u)$; and that of z to v by exit pin $z^+(v)$. This is the translation \mathcal{T}_u in the following table, and
2. Associate the value of x to $u + v$ by exit pin $x^+(u + v)$. This is translation \mathcal{T}_a .

Similarly, when these notions in state-machines are translated into that of SPOTS, we have two parts: $\mathcal{T}(\cdot) = \mathcal{T}_u(\cdot) \rightarrow \mathcal{T}_a(\cdot)$. Figure 12 gives these translations according to their grammar articles.

<i>grammar article</i>	\mathcal{T}_u	\mathcal{T}_a
$v ::= id$	$id^+(u_new_id)$	u_new_id
$v ::= id\$$	$id^-(u_old_id)$	u_old_id
$e ::= v$	$\mathcal{T}_u(v)$	$\mathcal{T}_a(v)$
$e ::= c$		c
$e ::= f(e_{i=1..n})$	$\bigcup_{i=1..n} \mathcal{T}_u(e_i)$	$f(\mathcal{T}_a(e_{i=1..n}))$
$E ::= id$		$id()$
$E ::= id()$		$id()$
$E ::= id(e_{i=1..n})$	$\bigcup_{i=1..n} \mathcal{T}_u(e_i)$	$id(\mathcal{T}_a(e_{i=1..n}))$
$G ::= f(e_{i=1..n})$	$\bigcup_{i=1..n} \mathcal{T}_u(e_i)$	$f(\mathcal{T}_a(e_{i=1..n}))$
$A ::= id := e$	$\mathcal{T}_u(e)$	$id^+(\mathcal{T}_a(e))$
$A ::= emit E$	$\mathcal{T}_u(E)$	$\mathcal{T}_a(E)$

Figure 12: Translation of events, guards and actions

4.3 Actions attached to states

As stated in section 2, a state-machine S may also be equipped with three actions: $A_{\text{stay}}(S)$, $A_{\text{out}}(S)$, $A_{\text{in}}(S)$. They are executed when the system stays, exits, enters the state-machine respectively. When these actions are considered, the translation is represented as follows.

Structural translation

For each state S , we will add its stay action $A_{\text{stay}}(S)$ to its behavior (cf. Figure 13).

$$\begin{aligned}
\mathcal{T}(\text{base}(b)) &= \mathcal{T}(A_{\text{stay}}(b)) \\
\mathcal{T}(S = \text{and}(\langle S_i \rangle_{i=1}^n)) &= \mathcal{T}(A_{\text{stay}}(S)) \parallel \bigwedge_{i=1}^n \mathcal{T}(S_i) \\
\mathcal{T}(S = \text{or}(\langle S_i \rangle_{i=1}^n, H, D, T)) &= \mathcal{T}(A_{\text{stay}}(S)) \parallel (\mathcal{T}(\mathcal{R}(S)) \bigvee \mathcal{T}(\mathcal{F}(S))) \\
\mathcal{T}(\mathcal{F}(S)) &= h_S^-(v) \triangleright \left(\bigvee_{i=1}^n (\alpha(v = S_i) \wedge \mathcal{T}(S_i) \rightarrow h_S^+(S_i)) \right) \\
\mathcal{T}(\mathcal{R}(S)) &= \bigvee_{dt \in T} \mathcal{T}(dt)
\end{aligned}$$

Figure 13: Structural translation

Detailed transitions

Consider a detailed transition $dt = (S_{\text{out}}, S_{\text{in}}, Sr, Tg, E[G]/A)$ in general. Before the action of A , we have to perform the action of leaving states $\mathcal{T}_{\text{out}}(Sr, S_{\text{out}})$. After the action A , we have to do the action of entering states $\mathcal{T}_{\text{in}}(Tg, S_{\text{in}})$. So, its translation will look like Figure 14.

$$\mathcal{T}(dt) = \left(\begin{array}{c} (\mathcal{T}^-(Sr, S_{\text{out}}) \parallel \mathcal{T}(E) \parallel \mathcal{T}(G)) \rightarrow \\ (\mathcal{T}_{\text{out}}(Sr, S_{\text{out}}) \rightarrow \mathcal{T}(A) \rightarrow \mathcal{T}_{\text{in}}(Tg, S_{\text{in}})) \rightarrow \\ \mathcal{T}^+(Tg, S_{\text{in}}) \end{array} \right)$$

Figure 14: Translation of detailed transitions

Translation of exiting/entering actions

The UML state machine specifies generic rules for sequencing the exiting actions and entering actions. That is, inner states are exited earlier than outer states, and outer states are entered earlier than inner states. Such scheduling constraints are also translated in SPOTS.

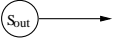
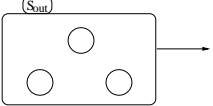
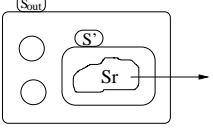
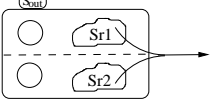
<i>transition</i>	$\mathcal{T}_{out}(Sr, s_{out})$
	if $\text{type}(s_{out}) = \text{base}$ $\mathcal{T}_{out}(Sr, s_{out}) = \mathcal{T}(A_{out}(s_{out}))$
	else if $\text{type}(s_{out}) = \text{or}$, $Sr = \{s_{out}\}$ or $Sr = \emptyset$ /* no further information in Sr */ $\mathcal{T}_{out}(Sr, s_{out}) = h_{Sout}^-(v) \triangleright \left(\bigvee_{s' \in \mathcal{S}(s_{out})} (\alpha(v = s') \rightarrow \mathcal{T}_{out}(\emptyset, s')) \right) \rightarrow \mathcal{T}(A_{out}(s_{out}))$
	else if $\text{type}(s_{out}) = \text{or}$, $\exists s' \in \mathcal{S}(s_{out})$, $Sr \subseteq \mathcal{S}^*(s')$ /* Sr is in sub-state s' */ $\mathcal{T}_{out}(Sr, s_{out}) = \mathcal{T}_{out}(Sr, s') \rightarrow \mathcal{T}(A_{out}(s_{out}))$
	else /* $\text{type}(s_{out}) = \text{and}$ */ $\mathcal{T}_{out}(Sr, s_{out}) = \parallel s' \in \mathcal{S}(s_{out}) \mathcal{T}_{exit}(Sr \cap \mathcal{S}^*(s'), s') \rightarrow \mathcal{T}(A_{out}(s_{out}))$

Table 3: Translation of exiting actions

Similar to the translations of start/exit pins, $\mathcal{T}_{out}(Sr, s_{out})$ and $\mathcal{T}^+(Tg, s_{in})$ are defined in Table 3 and 4.

Example 6 Consider again the detailed transition $dt_1 = (P, N, \{V, M\}, \{K\}, e1[g1]/a1)$. It will enter N and K in order. So its entering action is $\mathcal{T}(A_{in}(N)) \rightarrow \mathcal{T}(A_{in}(K))$. This transition exits the states V and $X, T/S$ (depending on which one is active in the system), M, Y , then it will also exit P . Hence, the translation of exiting actions is $((\Gamma_a \rightarrow \Gamma_b) \parallel \Gamma_c) \rightarrow \Gamma_d$ where $\Gamma_a, \Gamma_b, \Gamma_c, \Gamma_d$ are given in figure 15.

$$\begin{aligned}
\Gamma_a &= h_M^-(v) \triangleright ((\alpha(v = S) \parallel \mathcal{T}(A_{out}(S))) \vee (\alpha(v = T) \parallel \mathcal{T}(A_{out}(T)))) \\
\Gamma_b &= \mathcal{T}(A_{out}(M)) \rightarrow \mathcal{T}(A_{out}(Y)) \\
\Gamma_c &= \mathcal{T}(A_{out}(V)) \rightarrow \mathcal{T}(A_{out}(X)) \\
\Gamma_d &= \mathcal{T}(A_{out}(P))
\end{aligned}$$

Figure 15: Exiting action of dt_1

The rest of translation is the same as in section 4.2.

4.4 Priority

In UML, priority of transitions is introduced to solve partly the problem of conflicting transitions. When two conflicting transitions are both enabled, the one with higher priority will be fired.

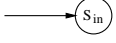
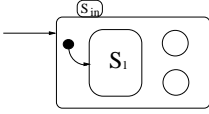
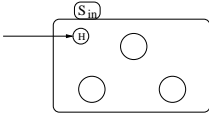
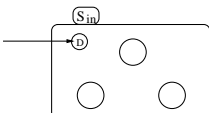
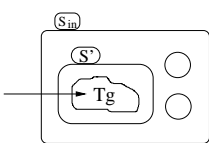
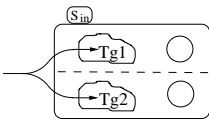
<i>transition</i>	$\mathcal{T}^+(\text{Tg}, s_{\text{in}})$
	<p>if $\text{type}(s_{\text{in}}) = \text{base}$</p> <p>$\mathcal{T}_{\text{in}}(\text{Tg}, s_{\text{in}}) = \mathcal{T}(\text{A}_{\text{in}}(s_{\text{in}}))$</p>
	<p>else if $\text{type}(s_{\text{in}}) = \text{or}$, $\text{Tg} = \emptyset$ or $\text{Tg} = \{s_{\text{in}}\}$</p> <p><i>/* no further information in Tg */</i></p> <p>$\mathcal{T}_{\text{in}}(\text{Tg}, s_{\text{in}}) = \mathcal{T}(\text{A}_{\text{in}}(s_{\text{in}})) \rightarrow \mathcal{T}_{\text{in}}(\emptyset, s_1)$</p>
	<p>else if $\text{type}(s_{\text{in}}) = \text{or}$, $\text{Tg} = \{\mathcal{H}(s_{\text{in}})\}$</p> <p><i>/* Enter the history */</i></p> <p>$\mathcal{T}_{\text{in}}(\text{Tg}, s_{\text{in}}) = \mathcal{T}(\text{A}_{\text{in}}(s_{\text{in}})) \rightarrow h_{\mathcal{S}_{\text{in}}}^-(v) \triangleright \left(\bigvee_{s' \in \mathcal{S}(s_{\text{in}})} (\alpha(v = s') \rightarrow \mathcal{T}_{\text{in}}(\emptyset, s')) \right)$</p>
	<p>else if $\text{type}(s_{\text{in}}) = \text{or}$, $\text{Tg} = \{\mathcal{D}(s_{\text{in}})\}$</p> <p><i>/* Enter the deep-history */</i></p> <p>$\mathcal{T}_{\text{in}}(\text{Tg}, s_{\text{in}}) = \mathcal{T}(\text{A}_{\text{in}}(s_{\text{in}})) \rightarrow h_{\mathcal{S}_{\text{in}}}^-(v) \triangleright \left(\bigvee_{s' \in \mathcal{S}(s_{\text{in}})} (\alpha(v = s') \rightarrow \mathcal{T}_{\text{in}}(\mathcal{D}(s'), s')) \right)$</p>
	<p>else if $\text{type}(s_{\text{in}}) = \text{or}$, $\exists s' \in \mathcal{S}(s_{\text{in}}), \text{Tg} \subseteq \mathcal{H}^*(s')$</p> <p><i>/* Tg is in s' */</i></p> <p>$\mathcal{T}_{\text{in}}(\text{Tg}, s_{\text{in}}) = \mathcal{T}(\text{A}_{\text{in}}(s_{\text{in}})) \rightarrow \mathcal{T}_{\text{in}}(\text{Tg}, s')$</p>
	<p>else <i>/* type(s_in) = and */</i></p> <p>$\mathcal{T}_{\text{in}}(\text{Tg}, s_{\text{in}}) = \mathcal{T}(\text{A}_{\text{in}}(s_{\text{in}})) \rightarrow \left(\parallel_{s' \in \mathcal{S}(s_{\text{in}})} \mathcal{T}_{\text{in}}(\text{Tg} \cap \mathcal{H}^*(s'), s') \right)$</p>

Table 4: Translation of entering actions

A detailed transitions dt_1 has higher priority than dt_2 , written $dt_1 > dt_2$, iff its source nodes are all substates of that of dt_2 . Formally, let $p(dt) = \mathcal{T}^-(\text{Sr}, \text{S}_{\text{out}})$ the set of start pins, $dt_1 > dt_2$ iff $p(dt_2) \subset p(dt_1)$.

Since only enabled transitions participate in the competition of activation, we can dynamically define the priority of a transition as \emptyset (the lowest priority) if it is not enabled. Formally:

$$\tilde{P}(dt) = \begin{cases} p(dt) & \text{if } dt \text{ is enabled} \\ \emptyset & \text{if } dt \text{ is not enabled} \end{cases}$$

Figure 16: Dynamic priority

With this in mind, we can define a “choice with priority” as:

$$\begin{aligned} \bigvee_{i=1..n}^p \mathcal{T}(dt_i) &= \bigvee_{i=1..n} \left(\mathcal{T}(dt_i) \wedge \alpha \left(\bigwedge_{j \neq i} \tilde{P}(dt_i) \not\subset \tilde{P}(dt_j) \right) \right) \\ \Gamma_1 \bigvee^p \Gamma_2 &= \bigvee_{i=1..2} \left(\Gamma_i \wedge \alpha \left(\mathcal{T}(dt) \in \Gamma_i \bigwedge_{\mathcal{T}(dt') \in \Gamma_{(3-i)}} \tilde{P}(dt) \not\subset \tilde{P}(dt') \right) \right) \end{aligned}$$

Figure 17: Choice with priority

Given a detailed transition $dt = (\text{S}_{\text{out}}, \text{S}_{\text{in}}, \text{Sr}, \text{Tg}, \text{E[G]}/\text{A})$ in general, these formulas are rewritten in terms of SPOTS as Figure 18.

$$\begin{aligned} & \left((\mathcal{T}^-(\text{Sr}, \text{S}_{\text{out}}), \mathcal{T}(E), \mathcal{T}(G)) \rightarrow \tilde{P}_{dt}(p(dt)) \right) \vee \tilde{P}_{dt}(\emptyset) \\ \bigvee_{i=1..n}^p \mathcal{T}(dt_i) &= \bigvee_{i=1..n} \left(\mathcal{T}(dt_i) \wedge \alpha \left(\bigwedge_{j \neq i} u_i \not\subset u_j \right) \right) \parallel_{i=1..n} \tilde{P}_{dt_i}(u_i) \\ \Gamma_1 \bigvee^p \Gamma_2 &= \bigvee_{i=1..2} \left(\Gamma_i \wedge \alpha \left(\mathcal{T}(dt) \in \Gamma_i \bigwedge_{\mathcal{T}(dt') \in \Gamma_{(3-i)}} (\tilde{P}_{dt}(u_dt), \tilde{P}_{dt'}(u_dt'), u_dt \not\subset u_dt') \right) \right) \end{aligned}$$

Figure 18: Implementation of priority in SPOTS

Structural translation

We can rewrite easily the structural translation by using this “choice with priority”, as shown in Figure 19.

$$\begin{aligned}
\mathcal{T}(\text{base}(b)) &= \mathcal{T}(\text{A}_{\text{stay}}(b)) \\
\mathcal{T}(S = \text{and}(\langle S_i \rangle_{i=1}^n)) &= \mathcal{T}(\text{A}_{\text{stay}}(S)) \parallel \bigwedge_{i=1}^n \mathcal{T}(S_i) \\
\mathcal{T}(S = \text{or}(\langle S_i \rangle_{i=1}^n, H, D, T)) &= \mathcal{T}(\text{A}_{\text{stay}}(S)) \parallel (\mathcal{T}(\mathcal{R}(S)) \overset{p}{\bigvee} \mathcal{T}(\mathcal{F}(S))) \\
\mathcal{T}(\mathcal{F}(S)) &= h_S^-(v) \triangleright (\bigvee_{i=1}^n (\alpha(v = S_i) \wedge \mathcal{T}(S_i) \rightarrow h_S^+(S_i))) \\
\mathcal{T}(\mathcal{R}(S)) &= \bigvee_{dt \in T}^p \mathcal{T}(dt)
\end{aligned}$$

Figure 19: Structural translation

Translation of detailed transitions

Based on this, we can now use the dynamical priority to activate a detailed transition, as shown in Figure 20.

$$\mathcal{T}(dt) = \left(\left(\tilde{P}_{dt}(u), \alpha(u \neq \emptyset) \right) \rightarrow (\mathcal{T}_{\text{out}}(S_r, S_{\text{out}}) \rightarrow \mathcal{T}(A) \rightarrow \mathcal{T}_{\text{in}}(Tg, S_{\text{in}})) \rightarrow \mathcal{T}^+(Tg, S_{\text{in}}) \right)$$

Figure 20: Translation of detailed transitions

The rest of translation is the same as section 4.3.

4.5 Adequacy of the translation

We have provided a formal specification of the UML semantics of state-machines, as defined in [7], by giving a detailed translation of its definition into a calculus consisting of pre-ordered transition systems. This pivot formalism provides a data structure which allows to address the key issues encountered in the deployment of reactive system designs on asynchronous networks. We assess the adequacy of the translation \mathcal{T} by considering its invariants.

Only enabled actions are activated The decomposition of the translation for detailed transitions (i.e. $\mathcal{T}(dt)$) formally specifies the transfer of control denoted by the corresponding UML state-machine structure. The set of start pins, the translation of events and guards ensures that only enabled actions will be activated.

Activated transitions are mutually non-conflicting This property is ensured by exclusive choice between 1/ the frame $\mathcal{F}(s)$ and the reduction $\mathcal{R}(s)$ of an or-state-machine, 2/ the transitions in the same level.

Activated transitions are mutually orthogonal This property is guaranteed by the pre-conditions to the translations of start pins $\mathcal{T}^-(s_r, s_{out})$ and composition among the set of sub-state-machines of an and-state-machine.

The ordering of activations actions is correct Activated transitions are maximal. Inner states are exited earlier than outer states. Outer states are entered earlier than inner states. The ordering of the transfers of control is rendered by causality relations \rightarrow between the translations of the successive activation and exit/enter actions. This provides a proper and methodical scheduling or sequencing of control.

Every step will result in a proper configuration The case analysis in the definition of \mathcal{T}_{out} and \mathcal{T}_{in} results in constructing the proper configuration for preparing the next step of the execution.

Actions in orthogonal area are executed concurrently This potential parallelism is rendered by utilizing synchronous composition.

5 Conclusion

Taking advantage of the rich background of the so-called synchronous approach in the design of reactive and distributed systems, we gave a faithful the translation of UML state-machines into a pivot synchronous calculus, based on mathematical notions of pre-orders. This translation enables an integrated development cycle for the reliable deployment of synchronous system specifications over asynchronous networks. In addition to earlier studies on the matter of providing a formal semantics for UML state-machines, the structure of our transition into SPOTS supports composite transition and history. To some extend, still, we focused on a kernel of UML state-machines relevant to the design of reactive systems. In particular, we did not formally discuss some more sophisticated constructs of UML state-machine, such as deferred events (which can however be modeled using states encoded by pins), *sync-state* (which might be encoded as synchronous composition of events), completion transitions (which could be encoded in a way similar to history), dynamical branches, etc. A prototype of the translator from state-machines to pre-orders has been implemented.

Acknowledgments

This work is partially supported by the project REUTEL-2000 funded by ALCATEL.

A Prototype and examples

A prototype of this translation has already been implemented in our group. For the time being, it reads a state-machine from a text file describing its root, the set of states and the set of transitions. Then the prototype will check if the state-machine is well defined. For example, if all sub-state-machines are defined, if the relation of “sub-state-machine” forms a tree, if the events, guards, and actions are well written according to our grammar, etc.. If the check passes, the prototype will calculate the level, main source, and main target of each transition and build the recursive structure as presented in section 2. Then it will generate the initial graph of the state-machine and the family of graphs of its behavior. Here we will give two translation examples according to the prototype.

A.1 Account example

There are two states for an account: credit (when balance is positive) and debit (when balance is 0 or negative). Two operation can be applied to an account when it is in credit: *deposit(amount)* and *withdraw(amount)*. However, only *deposit(amount)* can be applied when it is in debit. Figure 21 shows its state-machine.

In order to demonstrate the translation of staying action, we suppose the balance have an interest of 10% when it stays in credit, and a compensate of 20% when it stays in debit. These staying actions are not shown in Figure 21 for clarity.

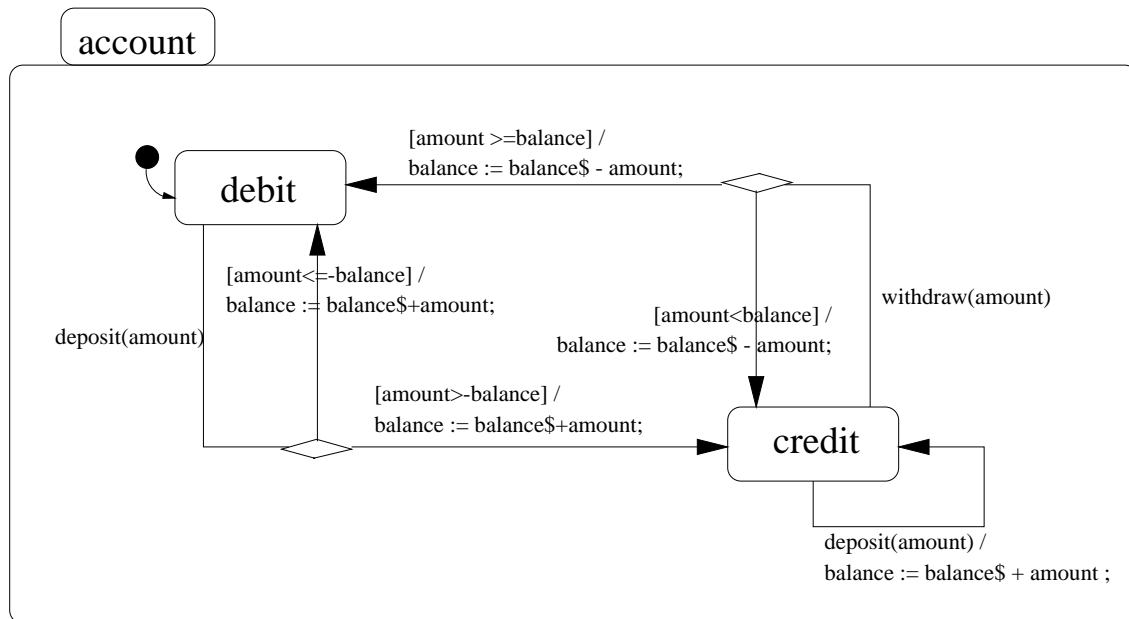


Figure 21: The state-machine of account

A.1.1 Input file

```
root = account
```

```
state = {  
  name = debit  
  type = base  
  stayaction = balance := balance$ * 1.2;  
}
```

```
state = {  
  name = credit  
  type = base  
  stayaction = balance:=balance$*1.1;  
}
```

```
state = {  
  name = account  
  type = or  
  substates = { debit, credit}  
}
```

```
transition =  
{  
  name = deposit2  
  source = {debit}  
  target = {debit}  
  label = deposit(amount) [amount<=-balance$]/balance:=balance$+amount;  
}
```

```
transition =  
{  
  name = withdraw1  
  source = {credit}  
  target = {debit}  
  label = withdraw(amount) [amount>=balance$]/balance:=balance$-amount;  
}
```

```
transition =  
{  
  name = withdraw2  
  source = {credit}  
  target = {credit}  
  label = withdraw(amount) [amount<balance$]/balance:=balance$-amount;  
}
```

```
transition =  
{  
  name = deposit3
```

```

    source = {credit}
    target = {credit}
    label = deposit(amount)/balance:=balance$+amount;
}

transition =
{
    name = deposit1
    source = {debit}
    target = {credit}
    label = deposit(amount) [amount>-balance$]/balance:=balance$+amount;
}

```

A.1.2 Recursive representation

```

account=0r(<debit,credit>,,{deposit1,deposit3,withdraw2,withdraw1,deposit2})
deposit1=(debit,credit,{debit},{credit},
    deposit(amount) [amount>-balance$]/balance:=balance$+amount;)
deposit3=(credit,credit,{credit},{credit},
    deposit(amount) []/balance:=balance$+amount;)
withdraw2=(credit,credit,{credit},{credit},
    withdraw(amount) [amount<balance$]/balance:=balance$-amount;)
withdraw1=(credit,debit,{credit},{debit},
    withdraw(amount) [amount>=balance$]/balance:=balance$-amount;)
deposit2=(debit,debit,{debit},{debit},
    deposit(amount) [amount<=-balance$]/balance:=balance$+amount;)
debit=*base*
credit=*base*

```

A.1.3 Translation

Initial graph:

```
{H_account+(debit)}
```

Behavior graph:

```
BDL_account=(BDL_rd_account \ / BDL_fr_account) || ()
```

```
BDL_fr_account=H_account-(v_account) |>
```

```

(    cx(v_account=debit)-->BDL_debit-->H_account+(debit)
  \ / cx(v_account=credit)-->BDL_credit-->H_account+(credit)
)

```

```
BDL_debit=balance-(u_old_balance),
```

```

balance+(u_old_balance*1.2)

BDL_credit=balance-(u_old_balance),
balance+(u_old_balance*1.1)

BDL_rd_account=(BDL_tr_deposit1\BDL_tr_deposit3\BDL_tr_withdraw2\
                BDL_tr_withdraw1\BDL_tr_deposit2)

BDL_tr_deposit1=(H_account-(debit),balance-(u_old_balance),amount+(u_new_amount),
i_e_deposit(u_new_amount),
guard(u_new_amount>-u_old_balance)
)-->()-->(
balance+(u_old_balance+u_new_amount)
)-->()-->H_account+(credit))

BDL_tr_deposit3=(H_account-(credit),amount+(u_new_amount),
i_e_deposit(u_new_amount)
)-->()-->(balance-(u_old_balance),
balance+(u_old_balance+u_new_amount)
)-->()-->H_account+(credit))

BDL_tr_withdraw2=(H_account-(credit),balance-(u_old_balance),amount+(u_new_amount),
i_e_withdraw(u_new_amount),
guard(u_new_amount<u_old_balance)
)-->()-->(
balance+(u_old_balance-u_new_amount)
)-->()-->H_account+(credit))

BDL_tr_withdraw1=(H_account-(credit),balance-(u_old_balance),amount+(u_new_amount),
i_e_withdraw(u_new_amount),
guard(u_new_amount>=u_old_balance)
)-->()-->(
balance+(u_old_balance-u_new_amount)
)-->()-->H_account+(debit))

BDL_tr_deposit2=(H_account-(debit),balance-(u_old_balance),amount+(u_new_amount),
i_e_deposit(u_new_amount),
guard(u_new_amount<=-u_old_balance)
)-->()-->(
balance+(u_old_balance+u_new_amount)
)-->()-->H_account+(debit))

```

A.2 A virtual example

In this virtual example, we aim to show the translation of and-state-machines, composite transitions across different levels, some different cases of generating start/exit pins, manipulation of history etc.. We suppose there is no staying/entering/exiting action in this state-machine for clarity, so, in the translation, we can see a lot of “()”s.

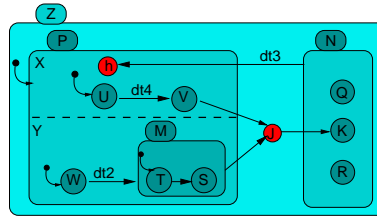


Figure 22: A virtual example

A.2.1 Input

```

root = Z

state = {
    name = Z
    type = or
    substates = {P,N}
}

state = {
    name = P
    type = and
    substates = {X,Y}
}

state = {
    name = X
    type = or
    substates = {U,V}
    history = h
}

state = {
    name = U
    type = base
}

state = {
    name = V
    type = base
}

state = {
    name = Y
    type = or

```

```

        substates = {W,M}
    }

    state = {
        name = W
        type = base
    }

    state = {
        name = M
        type = or
        substates = {T,S}
    }

    state = {
        name = T
        type = base
    }

    state = {
        name = S
        type = base
    }

    state = {
        name = N
        type = or
        substates = {Q,K,R}
    }

    state = {
        name = Q
        type = base
    }

    state = {
        name = K
        type = base
    }

    state = {
        name = R
        type = base
    }

    transition = {
        name = dt3
        source = {N}
        target = {h}
        label = e3[g3==0]/a3:=0;
    }

    transition = {
        name = dt1
        source = {V,M}
        target = {K}
        label = e1[g1==0]/a1:=0;
    }

```

```

    }

transition = {  name = dt4
                source = {U}
                target = {V}
                label = e4[g4==0]/a4:=0;
                }

transition = {  name = dt2
                source = {W}
                target = {M}
                label = e2[g2==0]/a2:=0;
                }

transition = {  name = dt5
                source = {T}
                target = {S}
                label = e5[g5==0]/a5:=0;
                }

```

A.2.2 Recursive representation

The prototype calculate the level, main source and main target for each transition, and put it in the right state-machine in order to build the recursive structure. For example, transition dt_4 is at level X , so it is put in X . On the contrary, transitions dt_3 and dt_1 are at level Z , so they are put in Z .

```

Z=Or(<P,N>,,,{dt1,dt3})
dt1=(P,N,{V,M},{K},e1[g1==0]/a1:=0;)
dt3=(N,P,{N},{h},e3[g3==0]/a3:=0;)
P=And(<X,Y>)
X=Or(<U,V>,h,,,{dt4})
dt4=(U,V,{U},{V},e4[g4==0]/a4:=0;)
U=*base*
V=*base*
Y=Or(<W,M>,,,{dt2})
dt2=(W,M,{W},{M},e2[g2==0]/a2:=0;)
W=*base*
M=Or(<T,S>,,,{dt5})
dt5=(T,S,{T},{S},e5[g5==0]/a5:=0;)
T=*base*
S=*base*
N=Or(<Q,K,R>,,,{})
Q=*base*
K=*base*

```

R=*base*

A.2.3 Translation

Initial graph:

{H_Z+(P), H_X+(U), H_Y+(W), H_M+(T), H_N+(Q)}

Bahavior graph:

BDL_Z=(BDL_rd_Z \ / BDL_fr_Z) || ()

BDL_fr_Z=H_Z-(v_Z) |>
(cx(v_Z=P)-->BDL_P-->H_Z+(P)
 \ / cx(v_Z=N)-->BDL_N-->H_Z+(N)
)

BDL_P=BDL_X || BDL_Y || ()

BDL_X=(BDL_rd_X \ / BDL_fr_X) || ()

BDL_fr_X=H_X-(v_X) |>
(cx(v_X=U)-->BDL_U-->H_X+(U)
 \ / cx(v_X=V)-->BDL_V-->H_X+(V)
)

BDL_U=()

BDL_V=()

BDL_rd_X=(BDL_tr_dt4)

BDL_tr_dt4=(H_X-(U), g4+(u_new_g4),
i_e_e4(),
guard(u_new_g4==0)
)-->()-->(a4+(0)
)-->()-->H_X+(V))

BDL_Y=(BDL_rd_Y \ / BDL_fr_Y) || ()

BDL_fr_Y=H_Y-(v_Y) |>
(cx(v_Y=W)-->BDL_W-->H_Y+(W)
 \ / cx(v_Y=M)-->BDL_M-->H_Y+(M)
)

BDL_W=()


```

BDL_M=(BDL_rd_M \ / BDL_fr_M) || ()

BDL_fr_M=H_M-(v_M) |>
  (   cx(v_M=T) -->BDL_T-->H_M+(T)
    \ / cx(v_M=S) -->BDL_S-->H_M+(S)
  )

BDL_T=()

BDL_S=()

BDL_rd_M=(BDL_tr_dt5)

BDL_tr_dt5=(H_M-(T),g5+(u_new_g5),
i_e_e5(),
guard(u_new_g5==0)
)-->()-->(
a5+(0)
)-->()-->H_M+(S))

BDL_rd_Y=(BDL_tr_dt2)

BDL_tr_dt2=(H_Y-(W),g2+(u_new_g2),
i_e_e2(),
guard(u_new_g2==0)
)-->()-->(
a2+(0)
)-->()-->()-->H_M+(T)-->H_Y+(M))

BDL_N=(BDL_rd_N \ / BDL_fr_N) || ()

BDL_fr_N=H_N-(v_N) |>
  (   cx(v_N=Q) -->BDL_Q-->H_N+(Q)
    \ / cx(v_N=K) -->BDL_K-->H_N+(K)
    \ / cx(v_N=R) -->BDL_R-->H_N+(R)
  )

BDL_Q=()

BDL_K=()

BDL_R=()

BDL_rd_N=SILENT

BDL_rd_Z=(BDL_tr_dt1\ /BDL_tr_dt3)

```

```

BDL_tr_dt1=(H_Z-(P),H_X-(V),H_Y-(M),g1+(u_new_g1),
i_e_e1(),
guard(u_new_g1==0)
)-->((())-->(),BDL_exit_M-->())-->())-->(
a1+(0)
)-->((())-->())-->H_N+(K)-->H_Z+(N))

```

```

BDL_tr_dt3=(H_Z-(N),g3+(u_new_g3),
i_e_e3(),
guard(u_new_g3==0)
)-->(BDL_exit_N)-->(
a3+(0)
)-->((())-->
(
()-->H_X(v_X)|>
(
cx(v_X=U)-->())-->H_X+(U)
\ / cx(v_X=V)-->())-->H_X+(V)
)
,
()-->())-->H_Y+(W)
)-->H_Z+(P))

```

```

BDL_exit_M=H_M-(v) |>
(
(cx(v=T)-->BDL_exit_T)
\ / (cx(v=S)-->BDL_exit_S)
)-->()

```

```

BDL_exit_T=()

```

```

BDL_exit_S=()

```

```

BDL_exit_N=H_N-(v) |>
(
(cx(v=Q)-->BDL_exit_Q)
\ / (cx(v=K)-->BDL_exit_K)
\ / (cx(v=R)-->BDL_exit_R)
)-->()

```

```

BDL_exit_Q=()

```

```

BDL_exit_K=()

```

```

BDL_exit_R=()

```

References

- [1] A. Benveniste, P. Le Guernic and C. Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. In *Science of Computer Programming*, v. 16, 1991.
- [2] M. Gogolla, F. Parisi Presicce. State diagrams in UML: a formal semantics using graph transformations. In *Proceedings of the Workshop on Precise Semantics of Modeling Techniques*. IEEE Press, 1998.
- [3] D. Harel. STATECHARTS: a visual formalism for complex systems. In *Science of Computer Programming*, v. 8. 1987.
- [4] D. Harel, A. Naamad. The STATEMATE semantics of STATECHARTS. In *ACM transactions on software engineering methods*, v. 5, n. 4. ACM press, October 1996.
- [5] D. Harel and H. Kugler. Synthesizing object systems from LCS specifications. *Draft*. August 1999.
- [6] D. Lateela, I. Majzik, M. Massink. Towards a formal operational semantics of UML state diagrams. In *Proceedings of the 3rd. IFIP International Conference on Formal Methods for Open Object-based Distributed Systems*. IEEE press, February 1999.
- [7] J. Rumbaugh, I. Jacobson, G. Booch. *The unified modeling language reference manual*. Addison-Wesley object technology series, 1999.
- [8] J.-P. Talpin, A. Benveniste, B. Caillaud, C. Jard, Z. Bouziane and H. Canon. BDL, a language of distributed reactive objects. *International Symposium on Object-Oriented Real-Time Distributed Computing*. IEEE Press, April 1998.
- [9] J.-P. Talpin, A. Benveniste, P. Le Guernic. Asynchronous deployment of synchronous pre-order transition systems. *Technical report* n. 1269. IRISA, October 1999.
- [10] A. C. Uselton, S. A. Smolka. State refinement in process algebra. In *proceedings of the North American Process Algebra Workshop*. New York, August 1993.
- [11] A. C. Uselton, S. A. Smolka. A compositional semantics for STATECHARTS using labeled transition systems. *Technical report 94/1*. State University of New York at Stony Brook, 1994.



Unité de recherche INRIA Rennes

IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)

Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)

Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur

INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)

<http://www.inria.fr>

ISSN 0249-6399