Int. J. Appl. Math. Comput. Sci., 2001, Vol.11, No.2, 537–556

TRANSFORMATION OF DYNAMIC ASPECTS OF UML MODELS INTO LOTOS BEHAVIOUR EXPRESSIONS

Bogumiła HNATKOWSKA[*], Zbigniew HUZAR[*]

The lack of formal semantics for the UML creates many ambiguity problems, especially when real-time systems are specified. The paper proposes an approach to a formal definition of UML statecharts. Main features of the UML statecharts are described, and next, a transformation of the UML statecharts into LOTOS is defined.

Keywords: statecharts, UML, LOTOS

1. Introduction

The Unified Modelling Language (UML) is a language for specifying, constructing, visualising, and documenting the artefacts of a software development process (Booch *et al.*, 1998). The UML represents a collection of the best object-oriented engineering practices that have proven successful in the modelling of large and complex systems (Douglass, 1999).

Although the developers of the UML have tried to provide a sufficient semantics and a notation, the formalism of the language still needs more improvements. Official UML documents describe most of the language constructs in a precise natural language. Some formality is necessary to help users and CASE designers to understand the language. It must be precise and approachable; a lack of either dimension damages its usefulness.

A model in the UML should express both static and dynamic aspects of the modelled system. The static aspects are mainly expressed by use-case diagrams as well as class and object diagrams. The dynamic aspects are expressed by interaction diagrams and state diagrams. The interaction diagrams are usually limited to a partial specification of the model behaviour, while state diagrams may be used for a complete specification of the model behaviour.

A state diagram of a given use-case or an object represents a state machine. The state machine specifies the behaviour as a set of sequences of states that the use-case or the object goes through during its lifetime in response to external events.

There are two ways to visualise the state machines: by activity diagrams or by statechart diagrams.

[*] Computer Science Department, Wrocław University of Technology, ul. Wybrzeże Wyspiańskiego 27, 50–370 Wrocław, Poland, `e-mail: {b.hnatkowska, z.huzar}@ci.pwr.wroc.pl`

The UML statecharts constitute an extension of Harel's statecharts (Harel, 1987), which are a generalisation of finite-state machines. Nestings of states and parallel states are two specific statecharts mechanisms, which effectively prevent a state explosion during specification. At the beginning, Harrell's statecharts were defined informally; next, several attempts have been made to formalise them (Harel *et al.*, 1987). It appeared that formalisation of their semantics is not a trivial task. Various approaches were used to formalise statecharts. For example, the paper (Armstrong, 1998) uses Real-Time Logic, and (Pnueli and Shalev, 1991) uses a denotational approach. The statecharts have been effectively used in several packages supporting software development. The STATEMATE is a widely-known example of their application (Harel and Naamad, 1996; Harel and Politi, 1996). The UML statecharts have well defined syntax, but their semantics is still informally defined.

The aim of the paper is to give precise semantics of the UML statecharts. To do this, we decided to use LOTOS (ISO, 1989) as an expressive formal description technique, which has proved to be useful in many applications. The semantics of a given statechart is defined as a LOTOS behaviour expression. We define an algorithm which transforms any statechart into a LOTOS behaviour expression. The UML statecharts possess many mechanisms which are sometimes redundant. Therefore, we concentrated on a subset of statecharts which seems to be the most important and, at the same time, which is representative for presentation of the transformation.

The paper is organised as follows. Section 2 gives a brief description of the UML statecharts and imposes restrictions that define their considered subset. Section 3 is the main section and presents rules of the transformation. The transformation is rather complex and therefore some of its details are moved to Appendix A. An example statechart and its transformation into a LOTOS behaviour expression are presented in Section 4. Further details of the example are presented in Appendix B. Section 5 ends the paper with some final remarks.

2. UML Statecharts

2.1. Statechart Graphs

UML statecharts visualise state machines by emphasising the potential states and transitions among these states. State and transitions are represented graphically as state boxes and transition arrows.

A *state* is a situation during which some (usually implicit) invariant condition holds. A *transition* is a relationship between two states indicating that an object in the first state may leave it and after that enters the second state. Usually, the occurrence of some event triggers a transition between states. An event has a location in time and space, and it is a stimulus that can trigger a state machine. Both in a state and during transition between states some actions may be performed. The actions are interpreted as executable atomic computations that result in a change in the state or a return of a value.

The following state categories are distinguished:

- *Pseudostates*, which consist of *initial* and *final* states, *shallow* and *deep* history, *fork* and *join, junction* and *stub* states.
- *Synchronisation* states, used to synchronise concurrent regions of a state machine.
- Normal states, or just *states*, which may be simple or composite. A *composite* state is a state that contains one or more other states. Any state enclosed within a composite state is called a substate, or a 'child' of that composite state, which in turn is called its 'parent'. A state is called a direct substate when it is not contained in any other state; otherwise it is referred to as a transitively nested substate. If a state contains only one direct substate, then it is called a *sequential* state. Otherwise, if it contains more than one direct substate, then the substates are called *regions* and the state is called a *concurrent* state. A state is *simple* if it does not contain any substate.

A statechart is represented as a graph. A *statechart graph* S is a six-tuple:

$$S = \langle BoxN, childB, typeB, defaultB, ArcN, Arc\rangle, \tag{1}$$

where:

- $BoxN$ is a finite set of names for boxes (graph vertices).
- $childB \subseteq BoxN \times BoxN$ is a hierarchy relation: $\langle b_1, b_2\rangle \in childB$ means that b_2 is a 'child' of a 'parent' b_1. The set $BoxN$ and the hierarchy relation $childB$ define a tree of boxes. The root r of the tree has no parents, and the leaves of the tree have no children. $childB^*$ is the reflexive transitive closure of $childB$.
- $typeB$: $BoxN \rightarrow$ {PRIM, XOR, AND, FIN} is a function that gives the type for each box. The root r is of type XOR, the leaves are of type PRIM or FIN, and other boxes may be either of type XOR (a sequential state box) or AND (a concurrent state box). The leaves of type FIN represent final boxes.
- $defaultB$: $BoxN \rightarrow 2^{BoxN}$ is a function that gives a default for each box. The default for a XOR box is a set with exactly one box of its children, while the default for an AND box is the set of all its children. The default for a PRIM box is the empty set. An extension of the default function is $defaultB : BoxN \rightarrow 2^{BoxN}$ defined as follows: $b \in defaultB(b)$, and for boxes $b' \in BoxN$ such that $\langle b, b'\rangle \in childB^*$ we require $b' \in defaultB(b)$ if and only if $defaultB(b') \subseteq defaultB(b)$.
- $ArcN$ is a finite set of arc names. Note that $BoxN \cap ArcN = \emptyset$.
- $Arc \subseteq BoxN \times ArcN \times BoxN$ is the set of arcs. An arc $\alpha \in Arc$ is a triple $\langle b_1, a, b_2\rangle$ with $source(\alpha) = b_1$, $target(\alpha) = b_2$, and $name(\alpha) = a$. It is assumed that arcs are uniquely identified by arc names.

2.2. Graph Labelling

Both state boxes and transition arrows are labelled. In addition to its name, a state box has:

- *actions*, executed on entering and exiting the state, which are atomic, i.e. uninterrupted and undivided computations,

- *sequences of actions*, executed in the state; actions are still atomic, but a sequence of actions may be interrupted after completion of any of its action,
- *internal transitions*, which are handled without causing a change in the state,
- *deferred events*, which are not handled in the state but are postponed and queued for handling in another state.

Each transition arrow has its source and target states, and moreover, it may be labelled by:
- a *triggering event*, whose reception in the source state makes the transition eligible to fire, provided that its guard condition is satisfied,
- a *guard condition* — a Boolean expression defined on the object's attributes evaluated at the moment of the reception of the triggering event; if evaluated to true, the transition is enabled to fire, if evaluated to false, the transition does not fire, and if there is no other transition that could be triggered by the same event, the event is lost,
- an *effect action*, which may directly act on the object that owns the state machine, and indirectly on other objects that are visible to the object.

The label with a triggering event may be empty, which means that the transition happens immediately after completing activities in its source state. If the label is not empty, then one of the following four kinds of events may occur:
- a *signal* event, which represents reception of an asynchronous signal,
- a *call* event, which represents reception of a request to synchronously invoke a specific operation,
- a *change* event, which occurs when an explicit Boolean expression becomes true as a result of a change in the value of one or more attributes,
- a *time* event, which occurs at the moment of expiration of a specific deadline.

Signal and call events may have parameters to carry data to objects.

A *statechart* is defined as the triple:

$$MS = \langle S, labB, labA \rangle, \tag{2}$$

where:
- S is a statechart graph,
- $labB$ is a *box labelling function* which assigns a box $b \in BoxN$ a quintuple

$$labB(b) = \langle entry(b), do(b), exit(b), deferrable(b), internal(b) \rangle, \tag{3}$$

  where $entry(b)$, $do(b)$, $exit(b)$ are sequences of actions, and $deferrable(b)$, $internal(b)$ are lists of events,
- $labA$ is an *arc labelling function*, which also assigns an arc $a \in ArcN$ a quintuple

$$labA(a) = \langle source(a), target(a), trigger(a), effect(a), guard(a) \rangle, \tag{4}$$

  where
  - $source(a)$ and $target(a)$ are functions determining the source and target boxes for a given arc a,

- $\mathit{trigger}(a)$ is a triggering event; it may be an event of any kind, in particular, it may be a timed event (time-out), and if it is empty, it means that the transition occurs immediately after completion of activities in the source state,
- $\mathit{effect}(a)$ is an action; if empty, it means there is no accompanying action,
- $\mathit{guard}(a)$ is a guard condition; if empty, it means an always-true condition.

For the sake of further use, we will employ an auxiliary function

$$\mathit{typeA} : \mathit{ArcN} \to \{\text{EV-LAB}, \text{TO-LAB}, \text{UN-LAB}\} \tag{5}$$

in order to distinguish between one of three situations: when the triggering event is a signal or a call event (EV-LAB), when it is a time event (TO-LAB), or when there is no triggering event (UN-LAB).

2.3. Statechart Transitions

The semantics of a statechart is described in terms of a sequence of its configurations. At a given moment a statechart is in some configuration. A configuration consists of a subset of active box states. If a statechart is in a composite box state, then this state is active and so are some of its sub-states. So, a configuration can be described as a tree of boxes.

More precisely, a *configuration* of the statechart S is a subset of boxes $B \subseteq \mathit{BoxN}$ such that $r \in B$, and for each box $b \in B$, if $\mathit{typeB}(b) = \text{AND}$ then all its children are in B, and if $\mathit{typeB}(b) = \text{XOR}$, then exactly one of its children is in B. The hierarchy relation childB restricted to this set B forms a subtree $\langle B, \mathit{childB} \cap B \times B\rangle$ of the tree of boxes $\langle \mathit{BoxN}, \mathit{childB}\rangle$. An initial configuration of the statechart S is defined as $B_{\text{init}} = \mathit{defaultB}(r)$.

Let $\mathit{lca}(b_1, b_2)$ stand for the *least common ancestor* of b_1 and b_2 in the box tree $\langle \mathit{BoxN}, \mathit{childB}\rangle$, i.e. $\langle \mathit{lca}(b_1, b_2), b_i\rangle \in \mathit{childB}^*$ $(i = 1, 2)$, and there is no other b such that $\langle b, b_i\rangle \in \mathit{childB}^*$, and $(\mathit{lca}(b_1, b_2), b) \in \mathit{childB}^*$. The boxes $b_1,\ b_2 \in \mathit{BoxN}$ are *orthogonal* if neither is an ancestor of the other and $\mathit{typeB}\,(\mathit{lca}(b_1, b_2)) = \text{AND}$. Similarly, two arcs $\langle b_1, a_1, b_1'\rangle$ and $\langle b_2, a_2, b_2'\rangle$ are orthogonal if boxes $\mathit{lca}(b_1, b_1')$ and $\mathit{lca}(b_2, b_2')$ are orthogonal.

A *transition* T from a configuration B to a configuration B' is defined as the maximal set of names of mutually orthogonal arcs in which a source box of each arc is an element of B.

A configuration is changed as a result of triggering a transition. Transitions depend on a configuration of the statechart and on an external environment of the statechart that generates events which enable passing along arcs. There are two kinds of transitions: low-level transitions between simple states and high-level transitions leaving composite states. High-level transitions have priority over low-level ones.

2.4. Statechart Limitations

In the UML standard document (Unified Modelling Language, 1998), the change of configurations is presented informally. In the following, we define formal semantics for the UML statecharts. We do not take into account evaluation of an object's attributes, and moreover, for simplicity, the following constraints are assumed:

– Only normal states (composite and simple), and initial and final pseudostates are taken into consideration.
– Crossing the border of composite states is not allowed. This means that if there is some arc between two boxes (states) b_1, $b_2 \in BoxN$, i.e. $\langle b_1, a, b_2 \rangle \in Arc$, then each of the boxes is a child of the same XOR type state. This restriction is introduced not only for the purposes of the paper, but it is also a strong recommendation for system modelling.
– The labelling function $labB$ is limited to the first three elements, i.e. there are neither deferred events nor internal transitions.
– The labelling function $labA$ is limited to the first three elements, i.e. there are no accompanying actions, and the guard conditions are always true.
– The unlabelled transitions are not considered at all. The main reason of this limitation is unclear semantics of such transitions in the context of complex states.

3. Transformation Rules

In this section, a function transforming a given statechart into a LOTOS specification is presented. We assume that the reader is familiar with LOTOS. For other readers, we recommend the book (Turner, 1993).

The function *B_Trans*(b) is the main function of the transformation. It defines some LOTOS behaviour expression for a given state b. The behaviour expression represents the functionality of a statechart rooted at the box b. A modelled system is always in its root state, and therefore the root state has no entry and exit actions. *B_Trans*(b) employs two additional functions $procD^*(b)$ and $procD(b)$, generating process definitions, representing AND and XOR types of boxes, respectively. The structure of the function follows the statecharts transformation into the process algebra presented in (Uselton and Smolka, 1994). The *B_Trans* function is used within the *SM_Trans*(sm) function, which yields for a given sm statechart a LOTOS specification.

Appendix A contains a skeleton of the *SM_Trans* function. In this section only the *B_Trans* function is presented.

The transformation function *SM_Trans* generates for a given statechart sm LOTOS specification S, see Fig. 1. The roles of the processes and their gates are described below.

The StateHandler process remembers an active state for each XOR box. The gates $putS$ and $getS$ are used for setting and getting an active state for a given XOR box, respectively.

The *Synchroniser* process locks reception of events by an entrance-state until the state becomes stable, i.e. after completion of all (nested) entry actions. The locking is constrained to the least common ancestor of the source and target states. The $arcN$

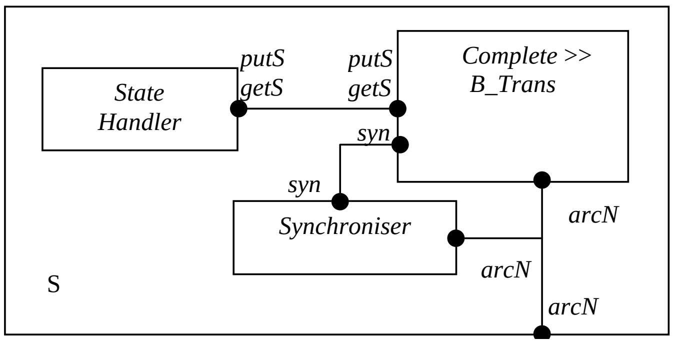

Fig. 1. The structure of the specification S. The behaviour expression of the specification instantiates three parallel processes: *StateHandler*, *Synchroniser* and the process which is the sequential composition of the *Complete* process with a behaviour expression resulting from application of the *B_Trans* function.

and *syn* gates are used respectively to lock and unlock reception of events in a given state.

The *Complete* process is responsible for the execution of the nested entry and exit actions for composite states. The process performs the following operations:

1. for a given arc a, it takes from the *StateHandler* an active box through the *getS* gate, provided that $\mathit{typeB}(\mathit{source}(a)) = \text{XOR}$;
2. it executes the suitable (nested) exit actions in the source state;
3. it executes the suitable (nested) entry actions in the target state;
4. it sets a new active state or states (communication on the *putS* gate);
5. it unlocks reception of events in the least common ancestor of the source and target states (communication on the *syn* gate).

A complete formal definition of the *SM_Trans* transformation function is rather lengthy; for further details we refer to Appendix A. The transformation upholds all the assumptions made in Subsection 2.4. Furthermore, it is assumed that action names and event names are unique, and the set of state names and the set of event names are disjoint. Within the definition the names of transformation functions are written in italics, while fragments of LOTOS specifications are written using a normal font.

The TO-LAB transitions are uniquely numbered, and the *trigger* function returns for a given TO-LAB transition its number. The $\mathit{leaving}(b)$ is a function that results for a given state b in the set of arc A for that $a \in A$ if $\mathit{source}(a) = b$.

We assume that $\text{card}\{a \in \mathit{leaving}(b) \mid \mathit{typeA}(a) = \text{TO-LAB}\} \leq 1$ for a given state b. The parallel and sequential compositions of behaviour expressions P_i, where $i \in I$, belonging to the finite set of behaviour expressions, are described by $\prod_{i \in I} P_i$

and $\sum_{i \in I} P_i$, respectively, i.e.

$$\prod_{i=1,\ldots,n} P_i = P_1[\,]P_2[\,]\cdots[\,]P_n \quad \text{and} \quad \sum_{i=1,\ldots,n} P_i = P_1|||P_2|||\cdots|||P_n$$

$$\mathit{B\_Trans}(b) = \begin{cases} b[\mathit{ExtArcN}] & \text{if } \mathit{typeB}(b) = \text{PRIM} \\ & \text{or } \mathit{typeB}(b) = \text{FIN}, \\ \mathit{B\_Trans}\big(\mathit{defaultB}(b)\big)[\text{¿}b[\mathit{ExtArcN}] & \\ & \text{if } \mathit{typeB}(b) = \text{XOR} \\ & \text{for each } b' \in \mathit{childB}(b) \\ & \text{a new process definition} \\ & \text{is generated by } \mathit{procD}(b'), \\ \left(\displaystyle\prod_{b' \in \mathit{childB}(b)} \mathit{B\_Trans}(b')\right)[\text{¿}b[\mathit{ExtArcN}] & \\ & \text{if } \mathit{typeB}(b) = \text{AND}, \text{ for all } b' \in \mathit{childB}(b) \\ & \text{a new process definition is generated by } \mathit{procD}^*(b') \end{cases} \tag{6}$$

where

$$\mathit{ExtArcN} = \mathit{list}\big(\{\mathit{putS}, \mathit{getS}, \mathit{syn}\} \cup \mathit{ActionNames} \cup \mathit{ArcN}\big), \tag{7}$$

$$\begin{aligned} \mathit{ActionNames} \;=\; \bigcup_{b \in \mathit{BoxN}} \Big(&\mathit{AS\_Names}\big(\mathit{entry}(b)\big) \cup \mathit{AS\_Names}\big(\mathit{exit}(b)\big) \\ &\cup \mathit{AS\_Names}\big(\mathit{do}(b)\big)\Big), \end{aligned} \tag{8}$$

$$\mathit{EventArcN} = \big\{a \in \mathit{ArcN} \bullet \mathit{typeA}(a) = \text{EV-LAB}\big\}. \tag{9}$$

The $\mathit{list}(S)$ is a function that transforms a given set S into a list containing all elements from S. The $\mathit{AS\_Names}(a)$ is a function that, for a given sequence of actions a, yields a set of actions names belonging to the sequence.

A process definition, representing a root state r, is an independent statechart and has the form

$$\textbf{process}\; r[\mathit{ExtArcN}] : \textbf{noexit} \;:=\; \textbf{stop endproc}. \tag{10}$$

A function procD^* for a given state b is defined as follows:

$$\mathit{procD}^*(b) = \textbf{process}\; b[\mathit{ExtArcN}] : \textbf{noexit} \;:=\; \textbf{stop endproc}. \tag{11}$$

A function procD for a given state b is defined as follows:

1. If $\mathit{typeB}(b) = \text{FIN}$ then

$$\mathit{procD}(b) = \textbf{process}\; b[\mathit{ExtArcN}] : \textbf{noexit} \;:=\; \textbf{stop endproc}. \tag{12}$$

2. If $typeB(b) \neq \mathrm{FIN}$ then

$$\begin{aligned}
&ProcD(b) = \textbf{process}\ b[ExtArcN] : \textbf{noexit}\ := \\
&(\ \ Do\_Trans(b)\, Time\_Out\big(leaving(b)\big) \\
&\ \ \ Disabling\_Part(b) \\
&)\ \ After\_Time\_Out\big(leaving(b)\big) \\
&\textbf{endproc}.
\end{aligned} \tag{13}$$

The functions that appear in the *procD* definition have the form:

$$Do\_Trans(b) = \begin{cases} do(b); & \text{if } do(b) \text{ is a non-empty sequence of actions separated with a semicolon,} \\ \text{an empty string} & \text{otherwise,} \end{cases} \tag{14}$$

$$Disabling\_Part(b) = \begin{cases} [> \Big(\sum_{\langle b,a,b'\rangle \in Arc} \big(trigger(a); exit\big) \\ >> \text{Complete } [ExtArcN](a) \\ >> B\_Trans(b')\Big) & \text{if there exists } x \in leaving(b) \text{ that } typeA(x) = \text{EV-LAB,} \\ \text{an empty string} & \text{otherwise,} \end{cases} \tag{15}$$

$$Time\_Out(A) = \begin{cases} \textbf{stop} & \text{if for each } a \in A\ typeA(a) = \text{EV-LAB,} \\ trigger(a); \textbf{exit} & \text{if there exits } a \in A \text{ such that } typeA(a) = \text{TO-LAB,} \end{cases} \tag{16}$$

$$After\_Time\_Out(A) = \begin{cases} >> \text{Complete}[ExtArcN](a) \\ >> B\_Trans\big(target(a)\big) & \text{if there exists } a \in A \text{ such that } typeA(a) = \text{TO-LAB,} \\ \text{an empty string} & \text{otherwise.} \end{cases} \tag{17}$$

4. Testing Examples

In order to verify the correctness of the transformation presented in the previous section, an exhausted testing was carried out. First, according to the transformation defined, testing statecharts were transformed by hand into executable LOTOS specifications. Next, the LOTOS specifications were tested by means of the modelling package LOLA (Pavon *et al.*, 1995).

In Figs. 2 and 3, we present only two simple examples of the tested statecharts. The first example illustrates a statechart with a composite sequential state and with a time-out transition. The second one illustrates a statechart with a composite concurrent state.

LOTOS specifications are rather lengthy, especially when statecharts contain concurrent states. In Appendix B we present the final text of LOTOS specification for the statechart of Fig. 2.

We assume that each simple state has entry and exit actions, represented by appropriate gates, for example the *ia* gate enables an entry action performance for box *bA*, and *ea* – an exit action performance for box *bA*.

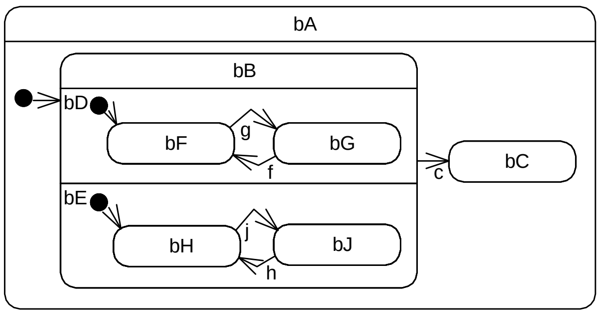

Fig. 2. The statechart with a sequential composite state.

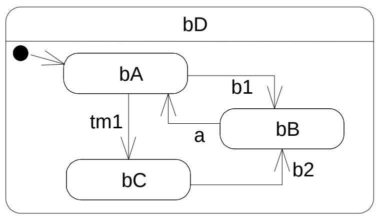

Fig. 3. The statechart with a concurrent composite state.

5. Conclusions

The paper discusses briefly the UML statecharts and presents a formalisation of their semantics using the formal specification technique LOTOS (ISO, 1989). LOTOS specifications are formal, which enables their examination by algebraic means, and they are executable, which enables their validation by testing. Unfortunately, LOTOS does not enable real-time systems to be specified. Its new enhanced version E-LOTOS (ISO, 1997) enables a time representation and the modelling of time constraints. However, up to now, there have been no programming tools supporting the development and validation of E-LOTOS specifications. In LOTOS, we model real-time aspects of UML statecharts using internal actions to represent time events.

We have concentrated on a subset of UML statecharts only; however, some extensions of the subset are possible. For example, omitted flow data can be easily included in an extended transformation.

A more difficult task is to take into account all kinds of states, because this entails the complexity of a final specification. A similar problem is caused by crossing borders of composite states by transition arrows. However, it seems that the maintenance of this restriction should be a firm methodological recommendation.

Internal transitions in a state box as well as effect actions labelling transition arrows can be easily taken into account. Another problem is related to the lack of deferred events. A solution to the problem requires extension of the notion of statechart configurations.

Our transformation of statecharts into LOTOS specifications is a modification and an extension of the algebraic approach presented in (Uselton and Smolka, 1994). In our approach, we took into account time transitions and, which was rather difficult, exit and entry actions.

The correctness of the transformation is the main evaluation problem. Because original semantics of UML statecharts is expressed informally, testing is the only way to validate it. The validation was performed in the following way. First, according to our transformation, exemplary statecharts were transformed into LOTOS specifications. Next, by means of the LOLA package (Pavon *et al.*, 1995), which enables syntax checking and step-by-step execution of specifications, transition sequences were generated. Finally, the obtained transition sequences of LOTOS specifications were compared with respective transition sequences from original statecharts.

It is well-known that a LOTOS specification can be automatically transformed into an (executable) program (Turner, 1993). So, our transformation may be treated as a prototype implementation for UML statecharts.

The UML is still under development. The paper contributed to formalisation of UML semantics, which is one of the most important demands.

References

Armstrong J. (1998): *Industrial integration of graphical and formal specifications.* — J. Syst. Software, Vol.40, pp.211–225.

Booch G., Rumbaugh J. and Jacobson I. (1998): *The Unified Modeling Language User Guide.* — Reading, Massachusetts: Addison-Wesley.

Douglass B.P. (1999): *Doing Hard Time. Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns.* — Reading, Massachusetts: Addison-Wesley

Harel D. (1987): *Statecharts: A visual formalism for complex systems.* — Science of Computer Programming, Vol.8, No.3, pp.231–274.

Harel D., Pnueli A., Schmidt J. and Sherman R. (1987): *On the formal semantics of statecharts.* — Proc. 2nd IEEE Symp. *Logic in Computer Science*, Ithaca, NY, pp.54–64.

Harel D. and Naamad A. (1996): *The statemate semantics of statecharts.* — ACM Trans. Software Engineering Method, Vol.5, No.4.

Harel D. and Politi M. (1996): *Modelling Reactive Systems with Statecharts: The Statemate Approach.* — Patent No.D-1100-43, i-Logix Inc.

International Standard Organisation (1989): *International Standard ISO/IEC 9646 Information Processing Systems—Open Systems Interconnection—LOTOS—A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour.*

International Standard Organisation (1997): *ISO/IEC JTC1/SC21 WG7—Final Committee Draft on Enhacements to LOTOS.*

Pavon S., Larrabeiti D. and Rabay G. (1995): *LOLA—LOTOS Laboratory, User Manual.* — Departamento de Ingenieria Telematica, Universidad Politechnica de Madrid, LOLA/NS/V10.

Pnueli A. and Shalev M. (1991): *What is in a step: On the semantics of statecharts*, In: Theoretical Aspects of Computer Software (T. Ito and A.R. Meyer, Eds.). — Berlin: Springer, pp.244–264.

Turner K.J. (Ed.) (1993): *Using Formal Description Techniques.* — Chichester: Wiley.

Unified Modelling Language (1998): UML Semantics Vol.1.3. — Rational Software Corporation.

Uselton A.C. and Smolka S.A. (1994): *A process algebraic semantics for statecharts via state refinement*, — Proc. IFIP Working Conf. *Programming Concepts, Methods and Calculi (PROCOMET)*, State University of New York at Stony Brook, pp.262–281.

Appendix A

Definition of the *SM_Trans* Function

SM_Trans(*ms*) =

```
specification S[list(ActionNames ∪ EventArcN)]: noexit
  type ArcNames is
  sorts arcN
  opns
    a_i : –> arcN               (* for each a_i ∈ ArcN such that typeA(a_i) = EV-LAB *)
    trigger(tm_i) : –> arcN    (* for each tm_i ∈ ArcN such that typeA(tm_i) = TO-LAB *)
    default : –> arcN          (* points to the default initial state *)
  endtype
  type BoxNames is
  sorts boxN
  opns
    b_i : –> boxN                 (* for each b_i ∈ BoxN *)
  endtype

behaviour
  hide putS, getS, syn, list({trigger(tm_i)}) in
                                   (* for each tm_i ∈ ArcN that typeA(tm_i) = TO-LAB *)
    (
       StateHandler[$putS$, $getS$]
    |[$putS$, $getS$]|
    ( Complete[$ExtArcN$](default) >>
       ($B\_Trans(r)$)                    (* where $r$ is the name of the root state *)
    )
    |[$syn$, $list(ArcN)$]|
       Synchroniser[$syn$, $list(ArcN)$]($TrueList$)
    )
    where

  process Synchroniser[$syn$, $list(ArcN)$]($BoxList$): noexit:=
   $LetExpression(r)$                (* where $r$ is a name of the root state *)
  ( $\sum_{b \in BoxN \bullet typeB(b)=\mathrm{PRIM} \text{ or } typeB(b)=\mathrm{FIN}}$ (
       $syn$ !boxN($b$); Synchroniser[$syn$, $list(ArcN)$]($ChangeBoxList(b)$) )
    [ ] $\sum_{a \in ArcN \bullet b=source(a)}$([$b$]->
       $trigger(a)$; Synchroniser[$syn$, $list(ArcN)$]($ChangeBoxList(target(a))$) )
  )
  endproc      (* Synchroniser *)

process StateHandler[$putS$, $getS$]: noexit :=
    $\prod_{b \in BoxN \bullet typeB(b)=\mathrm{XOR}}$ $b$[$putS$, $getS$]($default(b)$)
  where

process $b_i$[$putS$, $getS$]($s$ : boxN): noexit :=
                                        (* for each $b_i \in BoxN \bullet typeB(b_i) = \mathrm{XOR}$ *)
      $getS$ !$b_i$ !$s$; $b_i$[$putS$, $getS$]($s$)
     [ ] $\sum_{a \in choldB(b_i)}$ $putS$ !$b_i$ !$a$; $b_i$[$putS$, $getS$]($a$)
    endproc (* $b_i$ *)

  endproc (* StateHandler *)

  process Complete[$ExtArcN$]($s$ : arc$N$): exit :=
    ( [s=default] ->
        Entry_r[$ExtArcN$];
        $SetdefaultBox(r)$;        (* where $r$ is a name of the root state *)
    exit )
   [ ] $\sum_{a \in ArcN}$ ((([$s = a$]->         (* reception of an event *)
                                   (* the state becomes unstable *)
       Exit_$source(a)$[$ExtArcN$];      (* execution of appropriate entry actions *)
       Entry_$target(a)$[$ExtArcN$];     (* execution of appropriate exit actions *)
       $SetDefaultBox(target(a))$;    (* setting of a new active state *)
       $SetStableBox(target(a))$;     (* unlocking reception of events *)
                                   (* the state becomes stable *)
    exit ))

endproc (* Complete *)
```

```
(* definitions of processes that execute an entry action in a given $b_i$ state *)

  process Entry_$b_i$[$ExtArcN$] : exit :=     (* for each $b_i \in BoxN \bullet typeB(b_i)$ = PRIM *)
    $entry(b_i)$ exit                          (* the entry function returns a sequence *)
  endproc                                      (* of entry actions, separated and ended *)
                                               (* with a semicolon *)
  process Entry_$b_i$[$ExtArcN$]: exit :=      (* for each $b_i \in BoxN \bullet typeB(b_i)$ = FIN *)
    exit
  endproc

  process Entry_$b_i$[$ExtArcN$] : exit :=     (* for each $b_i \in BoxN \bullet typeB(b_i)$ = XOR *)
    $entry(b_i)$ Entry_$default(b_i)$          (* the entry function returns a sequence *)
  endproc                                      (* of entry actions, separated and ended *)
                                               (* with a semicolon *)
  process Entry_$b_i$[$ExtArcN$] : exit :=     (* for each $b_i \in BoxN \bullet typeB(b_i)$ = AND *)
    $entry(b_i)$                               (* the entry function returns a sequence *)
  [($\prod_{a \in childB(b)}$ Entry_$a$[$ExtArcN$])   (* of entry actions, separated and ended *)
  endproc                                      (* with a semicolon *)

(* definitions of processes that execute an exit action in a given $b_i$ state *)

  process Exit_$b_i$[$ExtArcN$] : exit :=      (* for each $b_i \in BoxN \bullet typeB(b_i)$ = PRIM *)
    $exit(b_i)$ exit                           (* the exit function returns a sequence *)
  endproc                                      (* of exit actions, separated and ended *)
                                               (* with a semicolon *)
  process Exit_$b_i$[$ExtArcN$] : exit :=      (* for each $b_i \in BoxN \bullet typeB(b_i)$ = FIN *)
   exit
  endproc

  process Exit_$b_i$[$ExtArcN$] : exit :=      (* for each $b_i \in BoxN \bullet typeB(b_i)$ = XOR *)
                                               (* $b_i \neq$ root *)
    $getS$ !$b_i$ ?$s$ : boxN;
    ( $\sum_{a \in childB(b_i)}$([$s = a$]−> Exit_$a$[$ExtArcN$])
    ); $exit(b_i)$ exit                        (* the exit function returns a sequence *)
  endproc                                      (* of exit actions, separated and ended *)
                                               (* with a semicolon *)
  process Exit_$b_i$[$ExtArcN$] : exit :=      (* for each $b_i \in BoxN \bullet typeB(b_i)$ = AND *)
    ( $\prod_{a \in choldB(b_i)}$ Exit_$a$[$ExtArcN$]   (* the exit function returns a sequence *)
    ); $exit(b_i)$ exit                        (* of exit actions, separated and ended *)
  endproc                                      (* with a semicolon *)

(* definitions of the processes generated by $procD$, and $procD^*$ functions *)
(* definition of the root state process $r$ *)
endspec
```

The *SM_Trans* function uses auxiliary functions defined as follows:

$$\mathit{SetdefaultBox}(b) = \begin{cases} \mathit{putS}\ !b'\ !b; & \text{if } \mathit{typeB}(b) = \text{PRIM} \\ & \text{or } \mathit{typeB}(b) = \text{FIN} \\ & \text{and } \exists b' \in \mathit{BoxN} \bullet b \in \mathit{childB}(b') \\ & \text{and } \mathit{typeB}(b') = \text{XOR} \\ \mathit{putS}\ !b'\ !b; & \text{if } \mathit{typeB}(b) = \text{XOR and} \\ \mathit{SetdefaultBox}(\mathit{default}(b)) & \exists b' \in \mathit{BoxN} \bullet b \in \mathit{childB}(b') \\ & \text{and } \mathit{typeB}(b') = \text{XOR} \\ \mathit{SetdefaultBox}(\mathit{default}(b)) & \text{if } \mathit{typeB}(b) = \text{XOR and} \\ & \neg\exists b' \in \mathit{BoxN} \bullet b \in \mathit{childB}(b') \\ & \text{and } \mathit{typeB}(b') = \text{XOR} \\ \mathit{putS}\ !b'\ !b; & \text{if } \mathit{typeB}(b) = \text{AND} \\ & \text{and } b_i \in \mathit{childB}(b) \\ \mathit{SetdefaultBox}(b_1)\ldots & \text{and } \exists b' \in \mathit{BoxN} \bullet b \in \mathit{childB}(b') \\ \qquad \mathit{SetdefaultBox}(b_k) & \text{and } \mathit{typeB}(b') = \text{XOR} \\ \mathit{SetdefaultBox}(b_1)\ldots & \text{if } \mathit{typeB}(b) = \text{AND} \\ \qquad \mathit{SetdefaultBox}(b_k) & \text{and } b_i \in \mathit{childB}(b) \\ & \text{and } \neg\exists b' \in \mathit{BoxN} \bullet b \in \mathit{childB}(b') \\ & \text{and } \mathit{typeB}(b') = \text{XOR} \end{cases} \tag{18}$$

$$\mathit{SetdefaultBox}(b) = \begin{cases} \mathit{syn}\ !b; & \text{if } \mathit{typeB}(b) = \text{PRIM} \\ & \text{or } \mathit{typeB}(b) = \text{FIN} \\ \mathit{SetdefaultBox}(\mathit{default}(b)) & \text{if } \mathit{typeB}(b) = \text{XOR} \\ \mathit{SetdefaultBox}(b_1)\ldots & \text{if } \mathit{typeB}(b) = \text{AND} \\ \qquad \mathit{SetdefaultBox}(b_k) & \text{and } b' \in \mathit{childB}(b) \end{cases} \tag{19}$$

$$\mathit{LetExpression}(b) = \begin{cases} \mathit{LetExpression}(b_1)\ldots\mathit{LetExpression}(b_k) & \\ \text{Let } b : \text{bool } = (b_1 \text{ and } \ldots \text{ and } b_k) \text{ in} & \text{if } \mathit{typeB}(b) = \text{XOR} \\ & \text{or } \mathit{typeB}(b) = \text{AND} \\ & \text{and } b_i \in \mathit{childB}(b) \\ \text{an empty string} & \text{otherwise} \end{cases} \tag{20}$$

$$\mathit{TrueList} = \text{true}^1, \ldots, \text{true}^k \qquad k = \text{card}\big(\{b \in \mathit{BoxN} \bullet \mathit{typeB}(b) = \text{PRIM} \text{ or } \mathit{typeB}(b) = \text{FIN}\}\big) \tag{21}$$

$$\mathit{BoxList} = b_1 : \text{ bool}, \ldots, b_k : \text{ bool} \qquad \text{Where } b_1, \ldots, b_k = \mathit{list}\big(\{b \in \mathit{BoxN} \bullet \mathit{typeB}(b) = \text{PRIM or } \mathit{typeB}(b) = \text{FIN}\}\big) \tag{22}$$

$$\mathit{ChangeBoxList}(b) = \mathit{if\text{-}not}(b, b_1), \ldots, \mathit{if\text{-}not}(b, b_k) \qquad \text{Where } b_1, \ldots, b_k = \mathit{list}\big(\{b \in \mathit{BoxN} \bullet \mathit{typeB}(b) = \text{PRIM or } \mathit{typeB}(b) = \text{FIN}\}\big) \tag{23}$$

$$\textit{If-not}(b_1, b_2) = \begin{cases} b_2 & \text{if } b_1 \neq b_2 \text{ and } (b_2 \notin \textit{childB}^*(b_1) \\ & \text{or } \big(b_2 \in \textit{childB}(b_1) \\ & \text{and } \textit{defaultB}(b_1) \neq b_2)\big) \\ \text{not}(b_2) & \text{if } b_1 = b_2 \\ \textit{if-not}\big(\textit{default}(b_1), b_2\big) & \text{if } \textit{typeB}(b_1) = \text{XOR} \\ & \text{and } b_2 \in \textit{childB}^*(b_1) \\ \textit{if-not}(b_3, b_2) & \text{if } \textit{typeB}(b_1) = \text{AND} \\ & \text{and } b_3\ \textit{childB}(b_1) \\ & \text{and } b_2 \in \textit{childB}^*(b_3) \end{cases} \tag{24}$$

The *SetdefaultBox* and *SetStableBox* functions generate fragments of the behaviour expression within the *Complete* process definition.

The other functions are related to the *Synchroniser* process definition. Their formal and actual parameters are described by the *BoxList* and *TrueList* functions, respectively. The number of formal (actual) parameters is the total number of PRIM and FIN states. The variables $b_1, \ldots, b_k$ of the sort *bool*, contained in the *Let* expression (the *LetExpression* function), represent the state of the corresponding composite states. If b_i is *true*, this means that the respective state is stable, otherwise the state is unstable. The variables are evaluated on the basis of actual parameters of the *Synchroniser* process. The composite state is stable only if all of its substates are stable. A simple target state is unstable until actions performed during the transition have been completed.

Appendix B

An Exemplary Specification Resulting from the *SM_Trans* Function

```
Specification S[ia,ea,ib,eb,ic,ec,b1,b2,a]: noexit
library Boolean endlib

type ArcNames is
sorts arcN
opns
  b1,b2,a :-> arcN
  tm1 :-> arcN
  default :-> arcN
endtype

type BoxNames is
sorts boxN
```

```
opns bA, bB, bC, bD :-> boxN
endtype

behaviour

hide putS, getS, syn, tm1 in
(* time-out transitions are hidden *)
(
  StateHandler[putS, getS]
|[putS, getS]|
(
  Complete[putS,getS,syn,ia,ea,ib,eb,ic,
       ec,b1,b2,a,tm1](default) >>
  ( bA[putS,getS,syn,ia,ea,ib,eb,ic,ec,b1,b2,a,tm1]
   [> bD[putS,getS,syn,ia,ea,ib,eb,ic,ec,b1,b2,a,tm1]
  )
)
|[syn,b1,b2,a,tm1]|
  Synchroniser[syn,b1,b2,a,tm1](true, true, true)
)

where

process Synchroniser[syn,b1,b2,a,tm1](A,B,C: bool): noexit :=
let bD: bool = (A and B and C) in
( syn !bA; Synchroniser[syn,b1,b2,a,tm1](not(A),B,C)
[]
  syn !bB; Synchroniser[syn,b1,b2,a,tm1](A,not(B),C)
[]
  syn !bC; Synchroniser[syn,b1,b2,a,tm1](A,B,not(C))
[]
  [A] -> b1; Synchroniser[syn,b1,b2,a,tm1](A,not(B),C)
[]
  [A] -> tm1; Synchroniser[syn,b1,b2,a,tm1](A,B,not(C))
[]
  [B] -> a; Synchroniser[syn,b1,b2,a,tm1](not(A),B,C)
[]
  [C] -> b2; Synchroniser[syn,b1,b2,a,tm1](A,not(B),C)
)
endproc (* Synchroniser *)

process StateHandler[putS, getS]: noexit :=
  bD[putS, getS](bA)
where

  process bD[putS, getS](s: boxN): noexit :=
    getS !bD !s; bD[putS, getS](s)
  []
```

```
    putS !bD !bA; bD[putS, getS](bA)
  []
    putS !bD !bB; bD[putS, getS](bB)
  []
    putS !bD !bC; bD[putS, getS](bC)
  endproc (* bD *)
endproc (* StateHandler *)

process Complete[putS,getS,syn,ia,ea,ib,eb,ic,ec,
                                      b1,b2,a,tm1](s: arcN): exit :=

( [s=default] ->
  EntrybD[putS,getS,syn,ia,ea,ib,eb,ic,ec,b1,b2,a,tm1]
  >> putS !bD !bA; exit
)
[]
( [s=b1] ->
  ExitbA[putS,getS,syn,ia,ea,ib,eb,ic,ec,b1,b2,a,tm1]
  >> EntrybB[putS,getS,syn,ia,ea,ib,eb,ic,ec,b1,b2,a,tm1]
  >> putS !bD !bB; syn !bB; exit
)
[]
( [s=b2] ->
  ExitbC[putS,getS,syn,ia,ea,ib,eb,ic,ec,b1,b2,a,tm1]
  >> EntrybB[putS,getS,syn,ia,ea,ib,eb,ic,ec,b1,b2,a,tm1]
  >> putS !bD !bB; syn !bB; exit
)
[]
( [s=a] ->
  ExitbB[putS,getS,syn,ia,ea,ib,eb,ic,ec,b1,b2,a,tm1]
  >> EntrybA[putS,getS,syn,ia,ea,ib,eb,ic,ec,b1,b2,a,tm1]
  >> putS !bD !bA; syn !bA; exit
)
[]
( [s=tm1] ->
  ExitbA[putS,getS,syn,ia,ea,ib,eb,ic,ec,b1,b2,a,tm1]
  >> EntrybC[putS,getS,syn,ia,ea,ib,eb,ic,ec,b1,b2,a,tm1]
  >> putS !bD !bC; syn !bC; exit )
endproc

(* ---------------- entry and exit ----------------*)

process EntrybA[putS,getS,syn,ia,ea,ib,eb,ic,ec,b1,b2,a,tm1]: exit :=
  ia; exit
endproc
```

```
process ExitbA[putS,getS,syn,ia,ea,ib,eb,ic,ec,b1,b2,a,tm1]: exit :=
  ea; exit
endproc

process EntrybB[putS,getS,syn,ia,ea,ib,eb,ic,ec,b1,b2,a,tm1]: exit :=
  ib; exit
endproc

process ExitbB[putS,getS,syn,ia,ea,ib,eb,ic,ec,b1,b2,a,tm1]: exit :=
  eb; exit
endproc

process EntrybC[putS,getS,syn,ia,ea,ib,eb,ic,ec,b1,b2,a,tm1]: exit :=
  ic; exit
endproc

process ExitbC[putS,getS,syn,ia,ea,ib,eb,ic,ec,b1,b2,a,tm1]: exit :=
  ec; exit
endproc

process EntrybD[putS,getS,syn,ia,ea,ib,eb,ic,ec,b1,b2,a,tm1]: exit :=
  EntrybA[putS,getS,syn,ia,ea,ib,eb,ic,ec,b1,b2,a,tm1]
endproc

(* ------------- processes definitions ------------- *)

process bA[putS,getS,syn,ia,ea,ib,eb,ic,ec,b1,b2,a,tm1]:
noexit :=
(  (* DoTrans is empty *)
  tm1; exit (* Time Out *)
  [> (* Disabling Part *)
  ( (b1; exit)
    >> Complete[putS,getS,syn,ia,ea,ib,eb,ic,ec,b1,b2,a,tm1](b1)
    >> bB[putS,getS,syn,ia,ea,ib,eb,ic,ec,b1,b2,a,tm1]
  )
) (* After Time-Out *)
  >> Complete[putS,getS,syn,ia,ea,ib,eb,ic,ec,b1,b2,a,tm1](tm1)
  >> bC[putS,getS,syn,ia,ea,ib,eb,ic,ec,b1,b2,a,tm1]
endproc

process bB[putS,getS,syn,ia,ea,ib,eb,ic,ec,b1,b2,a,tm1]:
noexit :=
( stop (* Time Out *)
[> (* Disabling Part *)
  (  (a; exit)
    >> Complete[putS,getS,syn,ia,ea,ib,eb,ic,ec,b1,b2,a,tm1](a)
    >> bA[putS,getS,syn,ia,ea,ib,eb,ic,ec,b1,b2,a,tm1]
  )
```

```
) (* After Time Out is empty *)
endproc

process bC[putS,getS,syn,ia,ea,ib,eb,ic,ec,b1,b2,a,tm1]:
noexit :=
( stop (* Time Out *)
   [> (* Disabling Part *)
  ( (b2; exit)
    >> Complete[ putS,getS,syn,ia,ea,ib,eb,ic, ec,b1,b2,a,tm1](b2)
    >> bB[putS,getS,syn,ia,ea,ib,eb,ic,ec,b1,b2,a,tm1]
)
) (* After Time Out is empty *)
endproc

process bD[putS,getS,syn,ia,ea,ib,eb,ic,ec,b1,b2,a,tm1]:
noexit :=
  stop (* root *)
endproc
endspec
```

Received: 17 April 2000
Revised: 12 December 2000