STUDIA UNIV. "BABEŞ–BOLYAI", INFORMATICA, Volume **XLIV**, Number 2, 1999

EXTENDING STATECHARTS FOR CONCURRENT OBJECTS MODELING

DAN MIRCEA SUCIU

Abstract. Object-oriented concurrent programming is a methodology that seems to satisfy nowadays requirements for complex applications development. The fundamental abstractions used in this methodology are concurrent (or active) objects and protocols for passing messages between them. Statecharts seem to be one of the most appropriate ways of modeling the behavior of concurrent objects. Based on statecharts we defined an executable formalism, called *scalable statechart*, for effective modeling of object-oriented concurrent applications with respecting of *homogeneous object model*.

Key words: object-oriented concurrent programming, reactive systems, statecharts.

1. Introduction

Object-oriented concurrent programming is based on object-oriented programming methodology, which is known at this moment as a top methodology for developing reusable applications. This methodology is conceptually simple and wide applicable and is based on two fundamental concepts:*objects* (that identify knowledge) and *message passing* (that is an unified protocol for communicating between objects).

The idea of building programming languages that can integrate the object- oriented programming mechanisms with concurrency mechanisms is very attractive. To achieve an optimal integration of these mechanisms is very useful to identify objects as activity units and to associate synchronizing code at the message passing level. These objects are often called *concurrent* objects, *active* objects or*actors*. The result of this unification is the integration of all the object-oriented programming and concurrency concepts and it allow the programmer not to be explicitly involved in establishing the synchronization discipline.

In section 2 we will describe and detail the key concepts of the object-oriented concurrent programming. Section 3 contains the structural description of a general concurrent (active) object that belongs to *homogeneous* object model, based on studies realized on over 100 object-oriented concurrent languages [SUC98].

Statecharts represent a visual formalism for describing states and transitions in a modular fashion, enabling nesting, orthogonality and refinement [HAR87],

[OMG99]. Statecharts are used for specifying objects behavior in designing complex systems. In section 4 we propose an extension of statecharts for modeling the behavior of concurrent (active) objects. The formal description of our statechart, called *scalable* statechart, proves its consistency and executability.

2. The context of object-oriented concurrent programming

In object-oriented concurrent programming a system is viewed as a physical simulation model of real or conceptual world behavior. This physical model is defined with a particular programming language and is materialized through an application.

Objects are key elements of object-oriented concurrent programming and they represent real or abstract entities with clear defined role into a system. An object has *identity, state* and *behavior*. All that an object knows *(state)* and can do *(behavior)* is expressed by sets of *properties* (or*attributes)* and *operations* (or *methods).* The state of an object is given by the values of its properties. The operations implement the object's behavior and they are procedures or functions that eventually modify the value of properties.

The object-oriented concurrent applications are composed by a set of objects that interact and communicate between them through *messages*. A *message* is a request for execution of an object's operation and it is composed of three elements: the identity of the receiver object, the name of the requested operation and a list of parameters. The mechanism of message passing allows objects to comunicate between them even if they are in different processes, contexts and/or computers. Because the entire activity of an object is revealed by its operations, the mechanism of message passing can express all the possible interactions between objects.

The process of identification of sets of objects with common properties and behaviors is called *classification.* The *class* is another key concept of object-oriented concurrent programming and represents the abstraction of the common elements (properties or operations) shared by a set of objects and describes their implementation.

The objects are concrete representations of classes and the process of building a particular object based on its class definition is called instantiation. The *concurrent* feature of a programming language represents the capacity of that language to express a potential parallelism. The object-oriented concurrent languages allow building applications where two or more operations are parallely executed, in distinct threads.

Based on the nature of relation between objects and threads, the concurrent object models can be classified in three categories: *orthogonal, homogeneous* and *heterogeneous* [PAP89].

In *orthogonal* approach the objects and threads are viewed as independent concepts. The objects are not implicitly protected by concurrent operation calls. Thus

the protection of the internal state is explicitly realized through low-level synchronization mechanisms, like semaphores or conditional critical regions [PHI95].

The *homogeneous* approach introduces the concept of *concurrent* (or *active) object*. An active object is an object that *controls* and *schedules* the execution of its operations. Active objects own threads, which in most homogeneous object models are implicitly created when a message is received. These objects may or may not be implicitly protected by external concurrent calls and contain specific mechanism for explicit protection of their internal state *(method guards, behavior abstractions, enable sets* etc [PHI95]). The logical conditions used in synchronizing the concurrent operations of active objects are called *synchronization constraints.*

In *heterogeneous* approach there are two kinds of objects: *active* and *passive*. Passive objects not own threads, are not protected (implicitly or explicitly) by external concurrent calls and their operations are executed in threads owned by caller objects.

3. The homogeneous concurrent object model

As we stated in section 2, the active objects that belong to homogeneous concurrent object models can control and schedule the receiving messages to protect their internal state by concurrent operation calls. The protection is implicit, when mechanisms with external control are used *(monitor*-like mechanisms) or explicit, in case of mechanisms with mixed or reflective control *(method guards, enable sets* etc) [SUC98].

In figure 1 is presented the structure of a general active object. The *interface manager* is a special entity located at each active object level. This entity controls and schedules the received messages and is materialized into a particular programming language by a distinct thread, a locking mechanism or a special object encapsulated in the kernel of active objects.

The interface manager controls the messages handling through asynchronous executions of associated operations based on object's state, synchronization constraints values and/or current executed operations. The messages scheduling is achieved through a special structure called *messages queue*, which retains all received and not handled messages. The interface manager, the messages queue, the properties and the synchronization constraints are not externally visible. Furthermore, just a subset of operations is visible and this subset represents the interface of the active object.

4. Scalable statecharts

In this section we define an extension of statecharts formalism for modeling the behavior of active objects corresponding to homogeneous object model. We define the scalable statecharts in an incremental way, starting from finite state machines (that we will call *level 0 scalable statecharts)* and adding elements to handle

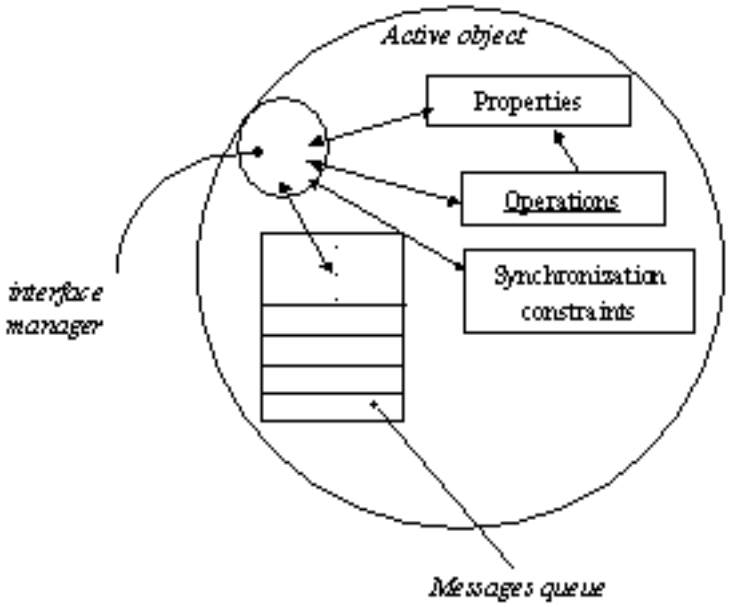


Figure 1. The structure of a general active object

the depth, orthogonality etc. For each intermediary statechart, the *configuration* and *execution* notions will be defined. The approach of definitions is a compositional one, where the execution of a statechart can be expressed by the executions of its components. The scalable statecharts represent an original way of modeling active objects behavior. They are closely related with particular concurrent object-oriented concepts, like *messages, operations, properties* or*synchronization constraints.* Therefore, the scalable statecharts are specialized versions of statecharts used in reactive systems modeling and they extend UML statecharts with specific elements of homogeneous object model presented in section 3.

4.1. **Level** 0 **scalable statecharts** (SS^0)**. Definition 1.** A *level 0 scalable statechart* of a class K is a tuple: $SS^0_K = (M, S, s_0, S_F, T; s_aC)$, where:

- M - is a finite set of messages. We will consider, without affecting the generality, that the messages signature do not contain parameters. The generalization of statecharts considering messages parameters is immediately and does not affect the semantics of statecharts execution. We will use $\perp$ to symbolize the empty message.
- S - is a finite, non-empty set of states,
- $s_0 \in S$ - is the initial state,

- *SF is a finite set of final states.* When S_F is empty, the modeled objects can not destroy themselves.
- $T \subseteq S \times M \times (S \cup S_F)$ is a finite set of transitions. A transition $(s', m, s'') \in T$ means that if an object is in state s' and receives the message m then, after handling of message m (in fact, after terminating of execution of the attached operation), the object will be in state s''.
- $s_a \in S$ - the active state of the statechart in a given moment,
- $C \in M^*$ is a finite sequence of messages, and models the messages queue of an active object. We will figure with $C = m_0 \widehat{} C_r$, where m_0 is the first message of the sequence and C_r represents the rest of sequence (the symbol $\widehat{}$ denotes the operation of concatenation).

Figure 2 contains an example of a SS^0 statechart and its visual representation. The structure of the modeled class *(Bottle)* is defined in the same figure using UML notation.

The first five elements of SS^0 form the static component and they describe the *structure* of the statechart. These five elements are shared by all objects of class K for which SS^0_K is defined and they are not modified by objects execution. The dynamic component consists of active state s_a and messages queue C and not describes the objects behavior. We will use the dynamic component for the *execution* of statechart.

Definition 2. The *configuration* of a SS^0 statechart is a tuple: $(s, _a\, m \widehat{} _0 C_r) \in S \times M^*$, where s_a is the active state and m_0 is the first message from queue C at a given moment. The *initial configuration* of a SS^0 statechart is given by tuple $(s_0, \perp)$.

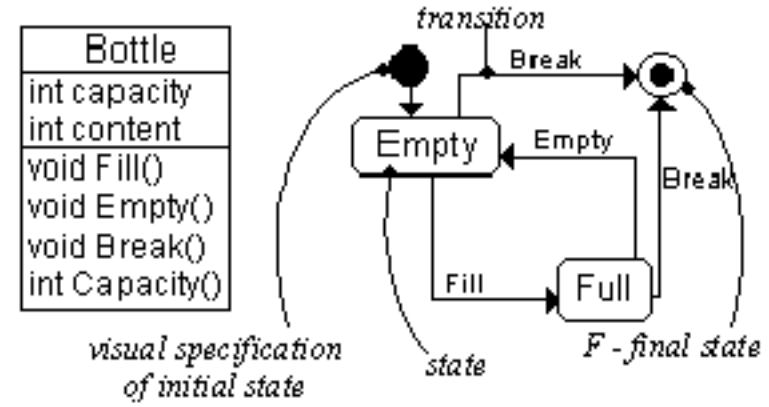
The *execution* of an active object modeled with a SS^0 statechart lay in sequential *interpretation* of messages from the queue C. The interpretation of a message implies the modifying of statechart configuration or the returning of that message in the queue C.

Definition 3. The *interpretation* of a SS^0 statechart configuration is a function: $\delta^0 : S \times M^* \to (S \cup S_F) \times M^*$,

$$\delta^0(s_a, m_0 \widehat{} C_r) = \begin{cases} (s', C'_r), & \text{if } \exists (s_a, m_0, s') \in T \\ (s_a, C'_r), & \text{if there is no } (s_1, s_2 \in S : (s_1, m_0, s_2) \in T \\ (s_a, C'_r \widehat{} m_0), & \text{otherwise} \end{cases}$$

In parallel with the interpretation of a configuration, the queue C may suffer some modifications. In the above definition of function δ^0 it is possible that $C_r \neq C'_r$ (in general, $C'_r = C^R_r$, where $R \in M^*$ represents the sequence of messages received by an object in the time of interpretation).

The interpretation of a SS^0 configuration models the functionality of the interface manager of active objects. A message will be accepted and its attached operation is executed if the message labels a transition that leaves the active state or if it does not appear in any transition label. Otherwise, the message is returned



$SS^0_{Bottle} = (M, S, T, s_o, S_F, s_a, C)$
$M = \{Fill, Empty, Break, Capacity\}$
$S = \{Empty, Full\}$
$T = \{(Empty, Fill, Full),$
$(Empty, Break, F),$
$(Full, Empty, Empty),$
$(Full, Break, F)\}$
$s_o = Empty$
$S_F = \{ F \}$

Figure 2. Graphical representation of SS^0statecharts

in the message queue. When it exists more than one transition labeled with the same message which leaves the active state, the selection of the interpreted transition is non- deterministic. To avoid the non-determinism from statecharts is possible to attach priorities to transitions.

Definition 4. The *execution* of a SS^0 statechart is a finite or infinite sequence of configuration interpretations, starting from the initial configuration, and it is denoted by:

$$(s_o, \perp) \xrightarrow{\delta^0} (s_1, m_1 \widehat{} C_{r1}) \xrightarrow{\delta^0} ... \xrightarrow{\delta^0} (s_k, m_k \widehat{} C_{rk}) \xrightarrow{\delta^0} ...,$$

where $s_0, s_1, , s_k, .. \in S$, $m_1, ..., m_k, ... \in M$ and $C_{r1}, ..., C_{rk}, ... \in M^*$. The execution is finite if a final state becomes an active state.

A possible execution of an object of class *Bottle* defined in figure 2 is:

$(Empty, \perp) \xrightarrow{\delta^0} (Empty, < Fill >< Fill >< Capacity >) \xrightarrow{\delta^0}$
$(Full, < Fill >< Capacity >< Break >) \xrightarrow{\delta^0}$
$(Full, < Capacity >< Break >< Fill >) \xrightarrow{\delta^0}$
$(Full, < Break >< Fill >) \xrightarrow{\delta^0} (F, < Fill >)$.

The message queue C described in definition 1 models a particular mechanism for choosing the next handled message. This mechanism was selected to simplify the description of *interpretation* and *execution* notions. In general, the synchronization mechanisms used in object-oriented concurrent languages are more complex, and allow attaching priorities to messages or have particular policies of selecting of the right message. These mechanisms can be modeled by replacing the queue C with the pair (C', pol), where $C' \in M^*$ and *pol* is a function $pol : M^* \rightarrow M$ that describes the choosing policy of a message from the set of received messages, modeled by C'. In this case, in all previous expressions the message m_0 will be replaced with $pol(C')$.

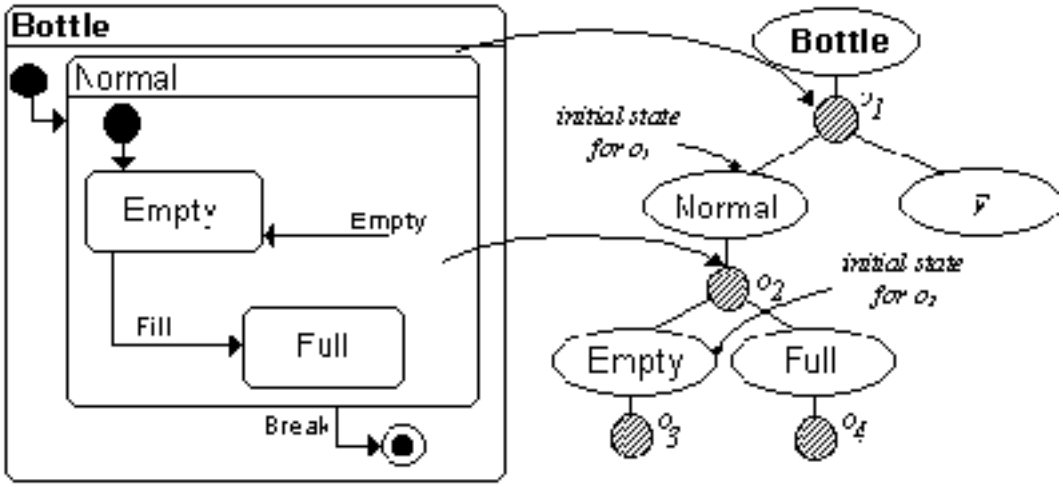
4.2. **Level 1 scalable statecharts** (SS^1)**.** We will attach to SS^0 statecharts the notions of *depth* and *orthogonality* introduced in [HAR87]. Because these extensions allow objects to be in more than one state, in distinct orthogonal components, we will extent the definitions of configuration, interpretation and execution of statecharts.

The notions of depth and orthogonality are modeled in SS^1 through a heterogeneous tree. The root of the tree and the intermediary nodes from even levels are states and the nodes from odd levels are orthogonal components (figure 3). We consider that all states have at least one orthogonal component and each orthogonal component can have zero or more states. The three types of states introduced in [HAR87] can be unitary modeled in this way:

- the *simple states* are states that have only one empty orthogonal component,
- the *composed states (OR-states)* are states that have only one non- empty orthogonal component,
- the *orthogonal states (AND-states)* are states that have more than one non-empty orthogonal components.

Definition 5. *A level 1 scalable statechart* of a class K is a tuple: $SS^1_K = (M, S, O, s_R, S_F, (stSucc, stInit, ortSucc), T; S_a, C)$, where:

- M - is a finite set of messages,
- S - is a finite, non-empty set of states,
- O - is a finite, non-empty set of orthogonal components,
- $s_R \in S$ - is the *root* of the states hierarchy,
- S_F is a finite set of final states. To preserve the consistency of our model we will presume that all the final states will be successors of



$$SS^{Bottle} = (M, S, O, s_R, S_F, (stSucc, stInit, ortSucc), T, C)$$

$M = \{Fill, Empty, Break, Capacity\}$
$S = \{Bottle, Normal, Empty, Full\}$
$O = \{o_1, o_2, o_3, o_4\}$
$s_R = Bottle$
$S_F = \{F\}$

$stSucc : \{o_1, o_2, o_3, o_4\} \rightarrow \wp(\{Bottle, Normal, Empty, Full, F\})$,
$stSucc(o_1) = \{Normal, F\}$, $stSucc(o_2) = \{Empty, Full\}$,
$stSucc(o_3) = \varnothing$, $stSucc(o_4) = \varnothing$,
$stInit : \{o_1, o_2\} \rightarrow \{Bottle, Normal, Empty, Full\}$,
$stInit(o_1) = Normal$, $stInit(o_2) = Empty$,
$ortSucc : \{Bottle, Normal, Empty, Full\} \rightarrow \wp(\{o_1, o_2, o_3, o_4\}) \setminus \{\varnothing\}$,
$ortSucc(Bottle) = \{o_1\}$, $ortSucc(Normal) = \{o_2\}$,
$ortSucc(Empty) = \{o_3\}$, $ortSucc(Full) = \{o_4\}$,

$T = \{(\{Empty\}, Fill, \{Full\}), (\{Full\}, Empty, \{Empty\}),$
$(\{Normal\}, Break, \{F\})\}$

Figure 3. Graphical representation of SS^1 statechart

orthogonal components from the root state s_R. Thus we will eliminate the termination transitions proposed in UML [OMG99] without affecting the modeling power of the statecharts.

- functions that defines the states hierarchy:
  - $stSucc : O \rightarrow \wp(S \cup S_F)$, where $stSucc(o) = s_1, s_2, ..., s_n$ is the set of sub-states of the orthogonal component o, with the restriction that $\forall o_1, o_2 \in O$ if we have $stSucc(o_1) \cap stSucc(o_2) = \phi$;
  - $stInit : O \backslash \{o : stSucc(o) = \phi\} \rightarrow S, stInit(o) = s_0 \in stSucc(o)$, the initial sub-state of the orthogonal component o ($stSucc$ is defined only for non-empty orthogonal components);
  - $ortSucc : S \rightarrow \wp(O) \backslash \{\phi\}$, where $ortSucc(s) = o_1, o_2, ..., o_m$ is the set of the orthogonal components owned by the state s, with the restriction that $\forall s_1, s_2 \in S$ we have $ortSucc(s_1) \cap ortSucc(s_2) = \phi$ (a state has at least one orthogonal component);
- $T \subseteq \wp(S \backslash \{s_R\}) \times M \times \wp(S \backslash \{s_R\})$ is a finite set of transitions. A transition $(s'_1, ..., s'_i), m, s''_1, ..., s''_j) \in T$ means that if an object is in *source* states $s'_1, ..., s'_i \in S \backslash \{s_R\}$ (each source state is located in distinct orthogonal components of a state from S) and receives a message m then, after executing the operation associated with m, the object will enter in *destination* states $s''_1, ..., s''_j \in S \backslash \{s_R\}$. The root state can't be source nor destination for a transition and the sets of source states and destination states do not contain states that include each other.
- $S_a \subseteq S \cup S_F$ - is the set of *active states* of the statechart in a given moment with the restriction that $\forall s_a \in S_a, ortSucc(s_a) = \phi$,
- $C \in M^*$ is a finite sequence of messages, and models the messages queue of an active object.

If an instance of class *Bottle* (modeled in figure 3) is in state *Full* then, corresponding to the states hierarchy, the instance it is in states *Normal* and *Bottle* too. To define in a unique way the *configuration* of a SS^1 statechart we extended the notion of *active* state. In definition 5 an active state of a SS^1 statechart has the property that is a simple state (has only one empty orthogonal component). Corresponding to definition 5, for the example from figure 3 the only states that can become active are *Empty, Full* and F.

Definition 6. A *pseudo-active* state is a composed state that contains an active sub-state. We denote with $S_{pa} \subseteq S$ the finite set of pseudo-active states of a SS^1 statechart in a given moment.

A SS^1 statechart can have more than one active state, each located in distinct orthogonal components of a pseudo-active state. It is obviously that the root state s_R is always pseudo-active.

If a composed state is a destination state of a transition and the transition is triggered, then its initial sub-states will be activated. We will define a recursive

function that will be used to determine the active states when a statechart enters in a composed state.

Definition 7. The *activation function* is a function that associates to each state $s \in S$ its simple or final sub-states that will be implicitly activated when an SS^1 statechart enters in state s. We will denote this function: $activ : S \cup S_F \rightarrow \wp(S \cup S_F)$,

$$activ(s) = \begin{cases} s, \text{ if } (s \in S, ortSucc(s) = 0, stSucc(0) = \Phi) 0rs \in S_F \\ \bigcup_{0 \in ortSucc(s)} activ(stInit(0)), \text{ otherwise} \end{cases}$$

The *global activation function*, denoted by:

$$Activ : \wp(S \bigcup S_F) \rightarrow \wp(S \bigcup S_F), Activ(S') = \bigcup_{s \in S} activ(s),$$

associates to a set of states their implicitly activated sub-states.

Definition 8. A *configuration* of a SS^1 statechart is a tuple $(S_a, m_0 \widehat{} C_r)$, where $S_a \subseteq S$ is the finite set of active states and $m_0 \widehat{} C_r \in M^*$ represents the content of the messages queue C in a given moment. The *initial configuration* of a SS^1 statechart if given by $(active(s_R), \perp)$.

Definition 9. The interpretation of a SS^1 statechart configuration is a function: $\delta^1 : \wp(S) \times M^* \; rightarrow \wp(S \bigcup S_F) \times M^*$,

$$\delta^1(S_a, m_0 \widehat{} C_r) = \begin{cases} (Activ(S"), C'_r), if (S', m_0, S") \in T si S' \subseteq S_a \cup S_{pa} \\ (S_a, C'_r), if \exists S_1, S_2 \subseteq S : (S_1, m_0, S_2) \in T \\ (S_a, C'_r \widehat{} m_0), otherwise \end{cases}$$

Definition 10. The execution of a SS^1 statechart is a finite or infinite sequence of configuration interpretations, starting from the initial configuration, and is denoted:

$$(active(s_R), \perp) \overset{\delta^1}{\rightarrow} (S_1, m_1 \widehat{} C_{r1}) \overset{\delta^1}{\rightarrow} ...(S_k, m_k \widehat{} C_{rk}) \overset{\delta^1}{\rightarrow} ...,$$

where $S_1, ..., S_k, .. \subseteq S, m_1, ..., m_k, ... \in M$ and $C_{r1}, ..., C_{rk}, ... \in M^*$. The execution is *finite* if the set of activated states contains at least a final state.

A possible execution of the statechart for an object of *Bottle* class (figure 3) is:

$(Empty, \perp) \overset{\delta^1}{\rightarrow} (Empty, < Empty >< Fill >< Capacity >) \overset{\delta^1}{\rightarrow}$

$(Empty, < Fill >< Capacity >< Empty >) \overset{\delta^1}{\rightarrow}$

$(Full, < Capacity >< Empty >< Break >) \overset{\delta^1}{\rightarrow}$

$(Full, < Empty >< Break >) \overset{\delta^1}{\rightarrow} (Empty, < Break >) \overset{\delta^1}{\rightarrow} (F, \perp).$

5. Conclusions

In the third section we ascertained a general structure of active objects. The implementation of this model is retrieved in most concurrent object-oriented languages that use synchronization mechanisms that belong to homogeneous concurrent object models. In section four we propose a formalism for modeling of active

objects behavior, called *scalable statechart*, that is based on statecharts visual formalism described by Harel in [HAR87]. The executability of scalable statecharts is a fundamental feature for automatization of active objects implementation. Moreover, the executability allows testing, simulating and debugging of active objects at the same level of abstraction as their behavioral model. In this way the conceptual gap between the formal models of active objects behavior and debugging at their source code level is avoided. Because the scalable statechart was defined having in mind a general model of active objects, it can be used for code generation in any concurrent object-oriented programming language that contains communication and synchronization mechanism belonging to homogeneous approach. This property gives more flexibility in translation of behavioral models in source code.

References

[BAR98] F. Barbier, H. Briand, B. Dano, S. Rideau, *The Executability of Object-Oriented Finite State Machines*, Journal of Object-Oriented Programming, SIGS Publications, 4(11), pp. 16-24, jul/aug 1998.

[BEE94] Michael von der Beeck, *A Comparison of Statecharts Variants*, Formal Techniques in Real-Time and Fault-Tolerant Systems,, L. de Roever & J. Vytopil (eds.), Lecture Notes in Computer Science, vol. 863, pp. 128-148, Springer-Verlag, New York, 1994.

[COO94] S. Cook, J. Daniels, *Designing Object Systems - Object-Oriented Modelling with Syntropy*, Prentice Hall, Englewood Cliffs, NJ, 1994.

[DOR96] Dov Dori, *Unifying System Structure and Behavior Through Object-Process Analysis*, Journal of Object-Oriented Programming, SIGS Publications, 4(9), pp. 66-73, jul/aug 1996.

[DOU99] Bruce Powel Douglas, *UML Statecharts*, Embedded Systems Programming, jan. 1999, available at http://www.ilogix.com/fs_prod.htm.

[GAN93] D. Gangopadhyay, S. Mitra, *ObjChart: Tangible Specification of Reactive Object Behavior*, Proceedings of ECOOP'93, Oscar M. Nierstrasz (ed.), Lecture Notes in Computer Science, vol 707, pp. 432-457, Springer-Verlag, 1993.

[HAR87] David Harel, *Statecharts: A Visual Formalism for Complex Systems*, Science of Computer Programming, vol.8, no. 3, pp. 231-274, June 1987.

[HAR96] D. Harel, A. Naamad, *The STATEMATE Semantics of Statecharts*, ACM Transactions on Software Engineering and Methodology, 5(4), pp. 293-333, 1996.

[HAR97] D. Harel, E. Gery, *Executable Object Modeling with Statecharts*, IEEE Computer, 30(7):31-42, Jul. 1997.

[MAN74] Z. Manna, *Mathematical Theory of Computation*, McGraw-Hill, 1974.

[OMG99] Object Management Group, *OMG Unified Modeling Language Specification*, ver. 1.3, June 1999 available on Internet at http://www.rational.com/.

[PAP89] Michael Papathomas, *Concurrency Issues in Object-Oriented Programming Languages*, in D. Tsichritzis, editor, Object Oriented Development, pg. 207-245, University of Geneva, Switzerland, 1989.

[PHI95] Michael Phillipsen, *Imperative Concurrent Object-Oriented Languages*, Technical Report TR-95- 049, International Computer Science Institute, Berkeley, aug. 1995.

[SCU97] Marian Scuturici, Dan Mircea Suciu, Mihaela Scuturici, Iulian Ober, *Specification of active objects behavior using statecharts*, Studia Universitatis "Babes Bolyai", Informatica, Vol. XLII, no. 1, pp.19-30, 1997.

[SUC98] Dan Mircea Suciu, *Reuse Anomaly in Object-Oriented Concurrent Programming*, Studia Universitatis "Babes-Bolyai", Informatica, Vol. XLII, no. 2, pp. 74-89, 1997.

"Babes-Bolyai" University, Faculty of Mathematics and Computer Science, RO-3400 Cluj-Napoca, Romania

E-mail address: tzutzu@cs.ubbcluj.ro