STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **XLV**, Number 2, 2000

USING SCALABLE STATECHARTS FOR ACTIVE OBJECTS INTERNAL CONCURRENCY MODELING

DAN MIRCEA SUCIU

Abstract. In the last two decades, the design of object models having concurrent features has represented a constant concern for many researchers. 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 will define an executable formalism, called *level 2 scalable statechart (SS^2)*, for modeling of intra-concurrency in object-oriented concurrent applications.

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

1. Introduction

In the last two decades, the design of object models having concurrent features has represented a constant concern for many researchers. This was happening for mainly two reasons. On the one hand, as an effect of the obtained technological progress, many object-oriented programming languages having concurrent features have been designed during this time (over 100 such languages have been discussed and systemized in [10]).

On the other hand, the fact is known that object-oriented programming has been developed having as a model our environment (seen as a set of objects among which several relationships exist and which communicate between them by message transmission). However, in the real world these objects are naturally concurrent, which leads to the normal trend of transposing this thing into programming.

It is interesting how two distinct criteria, the first one objective (determined by the rise of performances and complexities of the calculus systems), and the second one subjective (actually determined by "decency", which urges us to solve different abstract problems looking for similitude with the real world), have finally led to

2000 *Mathematics Subject Classification.* 68N30.

1998 *CR Categories and Descriptors.* D.2.3 [**Software**] : Software Engineering – *Coding Tools and Techniques* D.2.7 [**Software**] : Software Engineering – *Distirbution, Maintenance and Enhancements* .

the development of some concepts, some programming techniques and implicitly of some efficient analysis and design methods for developing applications.

The concurrent programming has occurred before the object-oriented programming. It has been applied for the first time within the framework of procedural languages. Here the main problems studied have been concerned to the synchronization of the parallel execution of some instruction sequences and to the information transmission among many other concurrent activities.

Once with the appearance of object-oriented programming software development has met a qualitative and meaningful leap. In this way, the development of these programs (or applications) does not involve the decomposition of problems into algorithmic procedures, but independent objects that interacts among them. An evaluation of the coordinating primitives of these interactions will be achieved in a concurrent system.

In the same time, a great interest was accorded to object oriented technology, especially to the analysis and design methods. The analysis and design methods may be defined as coherent approaches used to describe a system. Due to the complexity of the systems, different models are built, each of them containing another view of the system. Any model emphasize an aspect and neglect all the others. For instance, the entity- relation model describes the dates involved in the system and indicates nothing about their processing. In order to cover all the aspects connected with the design, every method uses more than one model.

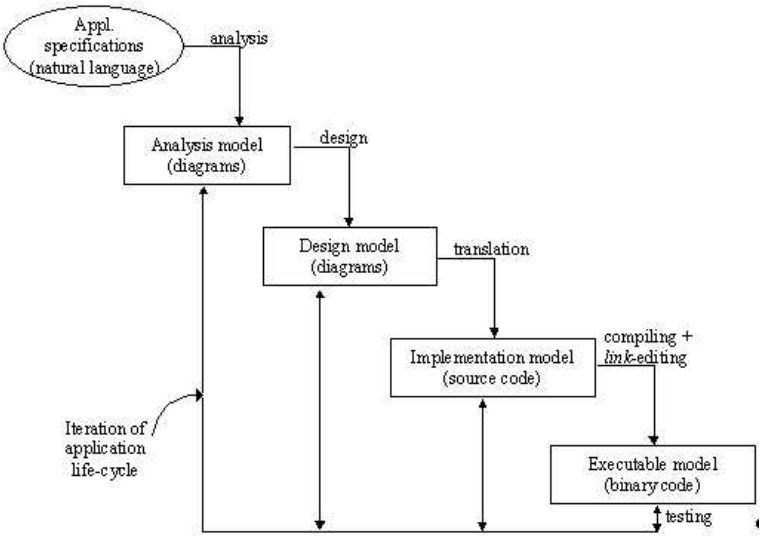

FIGURE 1. Iterative model of applications development using an object-oriented analysis/design method

The life cycle of an application, represents the stages that are go through in the process of developing that application. The most important stages are:

Analysis: where are identified the main characteristics of all possible correct solutions,

Design: that add to analysis models new elements that define a particular solution, based on some criteria optimizations,

Implementation: where an executable design is built for the particular solution modeled in design phase,

Testing: where is verified the equivalence of the implementation with the designed model and validates the fact that the implementation respects the correctness criteria identified in the analysis phase.

The object oriented analysis and design methods allow an iterative approach of the phases from applications life cycle (Figure 1).

CASE (*Computer Aided Software Engineering*) tools are software products able to support medium or large application development. This support is realised by automating some of the activities made in an analysis and design method. If we agree that one of the main goals of an analysis and design method is code generation and that we should obtain automatically a high rate of application code, it is obvious that an efficient use of a method cannot be made without an associated CASE tool.

Typically, the translation of a complex analysis/design model into a programming language takes a long period. A model is called *executable* if this translation can be made automatically. The automatization of the translation process allows running a prototype of an application immediately after building its model.

This paper captures aspects regarding concurrent object oriented application modeling. We analyzed the main object models developed in literature, insisting on concurrency aspects. In the center of this analysis is UML (*Unified Modeling Language*) version 1.3 [8].

The obtained results and the similarities between active object and reactive systems drive us to the idea of modeling their behavior through statecharts formalism. We extended the *scalable statecharts* formalism, introduced in [13], which allow developing executable models and offers support for automatic source code generation and for simulation of active objects behavior.

The executability is an important feature of *scalable statecharts* [13], allowing the automatization of active objects implementation based on their behavioral models. Furthermore, the executability offers support for simulation, testing and debugging of active object execution at the same level of abstraction like the built model.

2. Level 2 scalable statecharts (SS^2)

SS^1 statecharts defined in [13] do not allow parallel triggering of transitions. Thus SS^1 statecharts cannot be used to model intra-object concurrency. Furthermore, SS^1 statecharts do not provide mechanisms for modeling conditional synchronization and synchronization constraints.

We will extent SS^1 with new elements that allow us to specify state invariants, conditions for transition triggering and to handle more than one message from queue.

Definition 1. *A level 2 scalable statechart of a class K is a tuple:*

$$SS^2_K = (M, S, O, P, E, s_R, S_F, (stSucc, stInit, ortSucc,), inv, T; eval, par, S_a, C)$$

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,*
- *P is a finite set of properties,*
- *$s_R \in S$ 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 orthogonal components from the root state s_R. Thus we will eliminate the termination transitions proposed in UML* [8] *without affect the modeling power of the statecharts.*
- *functions that defines the states hierarchy:*
  - *$stSucc : O \to \mathcal{P}(S \cup S_F)$, where $stSucc(o) = \{s_1, s_2, \ldots, s_n\}$ is the set of sub-states of the orthogonal component o, with the restriction that $\forall o_1, o_2 \in O$ we have $stSucc(o_1) \cap stSucc(o_2) = \emptyset$;*
  - *$stInit : O \backslash \{o : stSucc(o) = \emptyset\} \to S, stInit(o) = s_0 \in stSucc(o)$, the initial sub-state of the orthogonal component o ($stSucc$ is defined only for non-empty orthogonal components);*
  - *$ortSucc : S \to \mathcal{P}(O) \backslash \{\emptyset\}$, where $ortSucc(s) = \{o_1, o_2, \ldots, o_m\}$ is the set of the orthogonal components owned by state s, with the restriction that $\forall s_1, s_2 \in S$ we have $ortSucc(s_1) \cap ortSucc(s_2) = \emptyset$ (a state has at least one orthogonal component);*
- *$T \subseteq \mathcal{P}(S \backslash \{s_R\}) \times M \times \mathcal{P}\ (S \backslash \{s_R\})$ is a finite set of transitions. A transition $(\{s'_1, \ldots, s'_i\}, m, \{s''_1, \ldots, s''_j\}) \in T$ means that if an object is in* source *states $s'_1, \ldots, s'_i \in S \backslash \{s_R\}$ (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 to m, the object will enter in* destination *states $s''_1, \ldots, s''_j \in S \backslash s_R\}$. The root state can not be source nor destination for a transition and the sets of source states and destination states not contain states that includes 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* $\forall s_a \in S_a, ortSucc(s_a) = \emptyset$,
- $C \in M^\star$ *is a finite sequence of messages, and models the messages queue of an active object.*

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.

Based on $stSucc$ and $ortSucc$ functions we will define another two functions that return the parent of a state or orthogonal component.

Definition 2. *The function* $stPred : O \to S$, *where* $stPred(o) = s \in S$ *if* $o \in ortSucc(s)$, *determines the parent state of an orthogonal component* $o \in O$. *The function* $ortPred : S \cup S_F \backslash \{s_R\} \to O$, $ortPred(s) = o \in O$ *if* $s \in stSucc(o)$ *determines the orthogonal component that is parent of a state* $s \in S \cup S_F \backslash \{s_R\}$.

The restrictions stated in definition 1:

$$\begin{aligned} \forall o_1, o_2 \in O, stSucc(o_1) \cap stSucc(o_2) &= \emptyset \text{ and} \\ \forall s_1, s_2 \in S, ortSucc(s_1) \cap stSucc(s_2) &= \emptyset, \end{aligned}$$

ensure that $stPred$ and $ortPred$ are well defined.

To complete the formal definition of SS^1 statecharts we will give a formal specification for *valid* transitions. For this reason, we will define first the *nesting relation* between states and/or orthogonal components.

Definition 3. *Two elements* $so_1, so_2 \in S \cup O$ *are in* nesting *relation, denoted by* $so_1 \prec so_2$, *iff one of the above affirmations is true:*

a) $so_1 = so_2$,

b) $so_1 \in S \wedge so_2 \in S \Rightarrow \exists n \in \mathbb{N}^+ : so_2 = \underbrace{stPred(ortPred(\cdots}_{n \text{ times}} so_1 \cdots))$,

c) $so_1 \in O \wedge so_2 \in O \Rightarrow \exists n \in \mathbb{N}^+ : so_2 = \underbrace{ortPred(stPred(\cdots}_{n \text{ times}} so_1 \cdots))$,

d) $so_1 \in S \wedge so_2 \in O \Rightarrow \exists n \in \mathbb{N}^+ : so_2 = \underbrace{ortPred(stPred(\cdots}_{n \text{ times}} ortPred(so_1) \cdots))$

e) $so_1 \in O \wedge so_2 \in S \Rightarrow \exists n \in \mathbb{N}^+ : so_2 = \underbrace{stPred(ortPred(\cdots}_{n \text{ times}} stPred(so_1) \cdots))$.

Proposition 1. *The nesting relation is partial order over* $S \cup O$.

Proof. The reflexivity is assured by the affirmation a) from nesting relation definition.

Let $so_1, so_2, so_3 \in S$ be three states such that $so1 \prec so_2$ and $so_2 \prec so_3$. From definition 7 we have that $\exists n \in \mathbb{N}^+ : so_2 = \underbrace{stPred(ortPred(\cdots}_{n \text{ times}} so_1 \cdots))$ and $\exists m \in \mathbb{N}^+ : so_3 = \underbrace{stPred(ortPred(\cdots}_{m \text{ times}} so_2 \cdots))$. This implies that $\exists r = n + m \in \mathbb{N}^+ : so_3 = \underbrace{stPred(ortPred(\cdots}_{r=n+m \text{ times}} so_1 \cdots))$, so $so_1 \prec so_3$. This means that the nesting relation is transitive over S. Analogous it can be proved that the nesting relation is transitive over $S \cup O$ for so_1, so_2, so_3 belonging to S and/or O.

We will prove that the nesting relation is anti-symmetrical over S.

Let $so_1, so_2 \in S$ be two states for which $so_1 \prec so_2$ and $so_2 \prec so_1$. This implies that:

$$so_1 = so_2,$$

or

$$\exists n, m \in \mathbb{N}^+ : so_2 = \underbrace{stPred(ortPred(\cdots}_{n \text{ times}} so_1 \cdots))$$

and

$$so_1 = \underbrace{stPred(ortPred(\cdots}_{m \text{ times}} so_2 \cdots)).$$

Let us suppose that $so_1 \neq so_2$. Then

$$\exists r = n + m \in \mathbb{N}^+ : so_1 = \underbrace{stPred(ortPred(\cdots}_{r=n+m \text{ times}} so_1 \cdots)).$$

From definition 1 we have that the above statement is true only for $r = 0$. This is obviously impossible because $r \in \mathbb{N}^+$. We deduce that $so_1 = so_2$. The other three cases ($so_1, so_2 \in O$, $so_1 \in O$ and $so_2 \in S$, $so_1 \in S$ and $so_2 \in O$) are analogous.

Thus, $\forall so_1, so_2 \in S \cup O$, $so_1 \prec so_2 \wedge so_2 \prec so_1 \Rightarrow so_1 = so_2$, i.e. the nesting relation is anti-symmetrical over $S \cup O$.

Because the relation $(S \cup O, \prec)$ is reflexive, transitive and anti-symmetrical we deduce that the nesting relation is partial order over $S \cup O$. □

Definition 4. *For a state or orthogonal component $so \in S \cup O$, $\{so' : so' \in S \cup O, so \prec so'\}$, denoted by $PRED_{so}$, is the set of all its predecessors.*

Proposition 2. *For all $so \in S \cup O$, $(PRED_{so}, \prec)$ is total order.*

Proof. Corresponding to proposition 1, the relation $(PRED_{so},)$ is partial order. Let $so', so'' \in PRED_{so} \cap S$ be two predecessor states of so. According to definition 8 we have:

$$\exists n' \in \mathbb{N}^+ : so' = \underbrace{stPred(ortPred(\cdots}_{n' \text{ times}} so \cdots))$$

and

$$\exists n'' \in \mathbb{N}^+ : so'' = \underbrace{stPred(ortPred(\cdots}_{n'' \text{ times}} so \cdots)).$$

We suppose that $n' > n''$. We have:

$$\exists n'' \in \mathbb{N}^+ : so'' = \underbrace{stPred(ortPred(\cdots}_{n'-n'' \text{ times}} so' \cdots))$$

that implies $so' \prec so''$. The other three cases ($so', so'' \in PRED_{so} \cap O$, $so' \in PRED_{so} capO$ and $so'' \in PRED_{so} \cap S$, $so' \in PRED_{so} \cap S$ and $so'' \cap PRED_{so} \cap O$) are analogous.

Thus, $\forall so', so'' \in PRED_{so}$, $so' \prec so''$ or $so'' \prec so'$, which implies $(PRED_{so}, \prec)$ is total order. □

Definition 5. *Let $(X, \prec)$ be a partially ordered set and let Y be a subset of X. An element $x \in X$ is a* lower bound *for Y iff $x \prec y$ for all $y \in Y$. A lower bound x for Y is the* greatest lower bound *for Y iff, for every lower bound x' for Y, $x' \prec y$. Whet it exists, we denote the greatest lower bound for Y by $\sqcap Y$.*

In the paper we use the following three well known results [9]:

- if x is a lower bound for Y and $x \in Y$ then $\sqcap Y = x$;
- if $\sqcap Y$ exists then it is unique;
- if $(Y, \prec)$ is total order and Y is finite then $\sqcap Y$ exists and $\sqcap Y \in Y$.

Because $(PRED_{so}, \prec)$ is total order and $PRED_{so}$ is a finite set, we deduce that the greatest lower bound for $PRED_{so}$ does exist, and $\sqcap PRED_{so} \in PRED_{so}$. We will prove that $\sqcap PRED_{so}$ is the parent of so.

Proposition 3. *Let $so \in S \cup O$ be a state or orthogonal component. One of the following affirmations is true:*

1) $so \in S \Rightarrow ortPred(so) = \sqcap PRED_{so}$,
2) $so \in O \Rightarrow stPred(so) = \sqcap PRED_{so}$.

Proof. a) Let $so \in S$ be a state. It is obvious that $so \prec ortPred(so)$, and based on the definition of set $PRED_{so}$ we have that $ortPred(so) \in PRED_{so}$.

Let $so' \in PRED_{so}$ be an arbitrary predecessor of the state so. From definition 8 we have that $so \prec so'$. If so' is an orthogonal component ($so' \in O$) then:

$$\exists n \in \mathbb{N} : so' = \underbrace{ortPred(stPred(\cdots}_{n \text{ times}} ortPred(so) \cdots)),$$

which implies that $ortPred(so) \prec so'$. The case when so' is a state ($so' \in S$) is analogous. Because so' was arbitrary selected from PREDso we will have:

$$\forall so' \in PRED_{so}, ortPred(so) \prec so'$$

. that implies $ortPred(so) = \sqcap PRED_{so}$.

The proof for statement b) is analogous. □

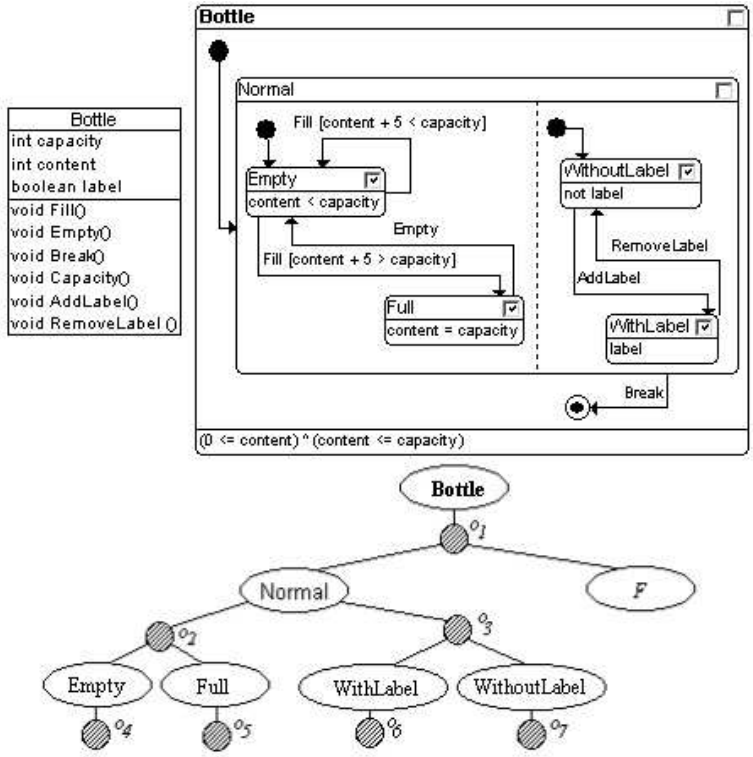

FIGURE 2. Graphical representation of SS^2 statechart

Definition 6. *Two states or orthogonal components* $so', so'' \in S$ *are* orthogonal *iff* $so' \not\prec so''$, $so'' \not\prec so'$ *and* $\sqcap(PRED_{so'} \cap PRED_{so''}) \in S$.

In other words, two states or orthogonal components are *orthogonal* if they are not in nesting relation and the closest common ancestor is a state.

Definition 7. *Let* $t = (\{s'_i \in S : i = 1, \ldots, n\}, m, \{s''_j \in S : j = 1, \ldots, m\}) \in T$ *be a transition. We say that* t *is a valid transition if all the following affirmations are true:*

a) $Ps' = \sqcap \bigcap_n^{i=1} PRED_{s'_i} \in S$ *(the source states are orthogonal),*
b) $Ps'' = \sqcap \cap_m^{j=1} PRED_{s''_j} \in S$ *(the destination states are orthogonal),*
c) $Ps' \not\prec Ps''$, $Ps'' \not\prec Ps'$ *and* $\sqcap(PRED_{Ps'} \cap PRED_{Ps''}) \in O$ *(source and destination states are not orthogonal).*

We will call $dom_t = \sqcap(PRED_{Ps'} \cap PRED'_{Ps'}) \in O$ *the domain of transition* t.

The domain of a transition represents the "smallest" orthogonal component that contains all transition's source and destination states.

In definition 1 function *par* characterizes the algorithm of choosing a set of messages from message queue. The specification of *par* function is not important in this phase of formalization and is imposed by particular mechanisms implemented in various concurrent object oriented languages. We consider that this function will return the maximal set of messages that can be handled concurrently.

Definition 8. *Two transitions* $t', t'' \in T$ *are textslindependent iff their domains are orthogonal, i.e.,* $\sqcap(PRED_{domt'} \cap PRED_{domt''}) \in S$.

Definition 9. *A* configuration *of a* SS^2 *statechart is a tuple* $(S_a, par(C), C_r)$, *where* $S_a \subseteq S$ *is the finite set of active states,* $par(C)$ *is the set of messages from queue which will be processed in parallel and* $C_r \in M^\star$ *the rest of messages queue* C *after removing messages from* $par(C)$. *The* initial configuration *of a* SS^2 *statechart if given by* $(active(s_R), \perp)$.

Definition 10. *The* interpretation *of a* SS^2 *statechart configuration is a function:*

$$\delta^2 : \mathcal{P}(S) \times \mathcal{P}(M) \times M^\star \to \mathcal{P}(S \cup S_F) \times M^\star,$$

$$\delta^2(S_a, \{m_1, \ldots, m_n\}, C_r) =$$

$$= \begin{cases} (Activ(\bigcup_n^{i=1} S_i''), C_r'), & \text{if } \forall i \in \{1, \ldots, n\} \exists (S_i' \subseteq S_a \cup S_{pa} \text{ and } eval(e_i) = true \\ (S_a, C_r'), & \text{if } \forall i \in \{1, \ldots, n\} \nexists S_1, S_2, S_2 \subseteq S_1, e \in E : (S_1, m_i, e, S_2) \in T \\ (S_a, C_r' \wedge m_1 \wedge \cdots \wedge m_n), & \text{else} \end{cases}$$

Definition 11. *The* execution *of a* SS^2 *statechart is a sequence finite or infinite of configuration interpretations, starting from the initial configuration, and is denoted:*

$$(active(s_R), \emptyset, \perp) \xrightarrow{\delta^2} (S_1, par(C), C_{r1}) \xrightarrow{\delta^2} \cdots \xrightarrow{\delta^2} (S_k, par(C), C_{rk}) \xrightarrow{\delta^2} \cdots$$

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

3. Conclusions

We extended the statecharts formalism [7] with new semantically and graphical elements, in order to allow the specification of active objects behavior with respect of a general concurrent object model. The extensions are: allowing scalability, executability and the definition of a precise semantic.

The formalism that is proposed in section two of this paper is called *level two scalable statechart*. The scalability of states minimizes the effort of modeling objects with a complex behavior. In this way, the active objects behavior models can be analyzed at different levels of detail.

Because the semantic of scalable statecharts was defined regarding a general concurrent object model, they allow source code generation in various concurrent

object-oriented languages that use various modalities and mechanisms for specification of concurrency and interaction between concurrent activities. This thing confers a better flexibility in translation of behavioral models in source code.

References

[1] 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

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

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

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

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

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

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

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

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

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

[11] 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

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

[13] Dan Mircea Suciu, Extending Statecharts for Concurrent Objects Modeling, Studia Universitatis "Babes-Bolyai", Informatica, Vol. XLIV, No. 1, pp. 37–44, 1999

Department of Computer Science, "Babeş-Bolyai" University, 1 M. Kogălniceanu St., RO-3400 Cluj-Napoca, Romania

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