URL: http://www.elsevier.nl/locate/entcs/volume82.html 12 pages

A Dual Language Approach Extension to UML for the Development of Time-Critical Component-Based Systems

Luigi Lavazza [1,2]

CEFRIEL and Dipartimento di Elettronica e Informazione
Politecnico di Milano
Milan, Italy

Sandro Morasca [3]

Dipartimento di Scienze Chimiche, Fisiche e Matematiche
Università degli Studi dell'Insubria
Como, Italy

Angelo Morzenti [4]

Dipartimento di Elettronica e Informazione
Politecnico di Milano
Milano, Italy

Abstract

A "dual language" component-based approach to the development of real-time critical applications is proposed. UML provides the constructs for modeling the structure of the system and the behavior of the system's components. A new descriptive language based on temporal logic, called OTL (Object Temporal Logic) is defined, in order to let the developer assert properties of the system at an abstract specification level. A development process consistent with the proposed notation is also briefly described.

[1] This work was partly supported by QUACK (A Platform for the Quality of New Generation Integrated Embedded Systems) cofinanced project MIUR 2001-2002.
[2] Email: lavazza@elet.polimi.it
[3] Email: sandro.morasca@uninsubria.it
[4] Email: morzenti@elet.polimi.it

1 Introduction

In the last few years, UML [1] has achieved a great popularity, essentially thanks to its graphical, easy-to-use notation and extensive tool support. Interestingly, UML is also being increasingly used for the development of real-time software. However, UML was not conceived for modelling real-time software. Its application to the real-time domain is limited by its lack of constructs to express time-related constraints and properties, as well as by its lack of formal semantics. To solve some of these problems, UML for Real-Time (alias UML-RT) has been defined on the basis of ROOM [9] and has been rapidly adopted by many developers. However, the application of UML-RT to the real-time domain is still suffering from several problems:

- UML-RT is not formally well defined. This is a relevant limitation, since very often real-time applications are also safety-critical ones. Thus they call for activities (e.g., property verification, system simulation, test case generation) that are very hard –if at all possible– to carry out when the specifications are written in semi-formal notations.
- UML-RT is an effective notation for the design and implementation of systems, but not for representing requirements or specifications. For instance, when modeling the operational environment of a real-time system, it is often necessary to represent non-deterministic behaviors or simultaneous events. These phenomena are not supported by UML-RT.
- Finally, time-related information (i.e., the representation of time and time constraints) are not treated at a native level, but only through ad-hoc components (like timers).

The problems described above are particularly relevant since UML-RT is probably going to be included in the forthcoming UML 2.0 [11]. The development of real-time critical applications calls for a rigorous process, which should be based on notations that are expressive and close enough to those currently used in industry, to keep their application cost low. At the same time, the notations should be formal, to allow the application of formal methods for property verification, test case generation, etc. In particular, we need:

- A simple, expressive notation for modeling the structure of real-time systems. For this purpose, a UML-based notation is probably the best choice.
- An easy-to-use, though precise notation for specifying the behavior of the system, including time information in a quantitiative way.
- A notation to express the required properties of the system.

UML can deal with all of the above issues. Class and object diagrams describe the system structure, state diagrams illustrate the dynamics of the system, and OCL specifies both static and dynamic properties and constraints. Nevertheless, real-time critical systems cannot be modeled satisfactorily with the standard UML or UML-RT. In this paper, we propose a set of notations and

a process to support the development of this kind of systems.

2 The proposed approach

Our proposal for extending/specializing UML for time critical systems is *not* only a notation, unrelated to the development process. This specialized domain requires systematic and rigorous development, centered on explicit, possibly formal requirements specification, and requirement validation and verification are also of crucial importance. Our proposal combines a set of carefully thought and balanced notations which can support the most suitable development methods and can be used by practitioners in industrial environments.

The hardware and software architecture of critical embedded systems is often quite static. As for hardware, there are obvious reasons of stability and continuity of functioning. From the software viewpoint, a static architecture is needed because these systems are carefully designed to achieve high performance and especially to exhibit a predictable behavior. This is often obtained by a static allocation of objects and, in general, by a management of the computational resources that is as invariant in time as possible.

The notation we propose is centered on architectural diagrams that correspond to UML-RT collaboration diagrams. System components are modeled, along with the relations of inclusion and communication, via a small set of fundamental constructs: capsules correspond to components; ports and protocols model abstract interfaces (i.e., they describe only the alphabet, not the behavior); and connectors correspond to communication relations. The partitioning of a complex system into a set of components (i.e., parts) that conceptually evolve in parallel and communicate via connectors can be iterated to an arbitrary level of depth. This results in a tree-shaped hierarchy of parts and sub-parts, where the root corresponds to the overall system being modeled, and the leaves to the components that are not further structured, which are modeled in an operational style with a state-transition machine. In our proposal, however, leaf-level components may not be associated with a general statechart, i.e., with and/or states decomposition and signal broadcast. We believe that, although they may enrich the notation and make it more concise, these features are often detrimental to understandability and semantic terseness. We deliberately keep the architectural structuring mechanisms (i.e., the partitioning of a system into a set of parts/components) completely separate from those for describing the behavior (i.e., "flat" statecharts).

In addition to the statecharts associated with leaf-level capsules, we also propose a descriptive formalism to specify the behavior of a system and its components, whose style is thus complementary to that of statecharts. Specifically, this description consists of a formula of a new logic, called OTL (Object Temporal Logic), which we define in such a way as to make it compatible with the original OCL (Object Constraint Language) descriptive notation for asserting properties in UML. OTL formulas and statecharts are also comple-

mentary at the methodological level, since an OTL formula acts as an abstract specification of constraints and temporal relations that must hold among the states, events, and signals of the statechart machine associated with the same capsule. Thus, we propose a "dual language" approach: the OTL part is an abstract specification of the properties that are required to the behavior of the state machine. Hence, there is no redundancy among the information provided by the OTL formula and the statechart associated to a given capsule/component: on the contrary, they must be in the classical specification/implementation relation that is typical of dual language approaches to the development of reactive systems.

3 The OTL language

The Object Constraint Language (OCL) defined in UML can be used to state behavioral properties of a system and its parts. However, when dealing with time-dependent systems, OCL (in its current form or in the one proposed in [10] for OCL 2.0) needs to be extended to fully specify temporal aspects. OCL cannot explicitly predicate about the temporal properties of a system, so only some temporal properties can be modeled adequately. As an example, the invariant construct `inv` is used to specify a property that must hold in all the states of the system. This construct can be seen as the *Always* construct of temporal logic, whose parameter is a property that must hold at all times during the evolution of the system. However, not much else can be expressed as far as temporal properties are concerned, so several other kinds of important temporal properties of systems cannot be adequately specified (and therefore verified). In particular, it is not possible to specify the time distance between events, since time is not represented in OCL. This has a fundamental importance in time-critical systems, where the response to a stimulus must be guaranteed to occur within some specified time interval.

We propose Object Temporal Logic (OTL) as a temporal logic extension to OCL. Based on one fundamental temporal operator, OTL provides the typical basic temporal operators of temporal logics, i.e., *Always, Sometimes, Until*, etc. In addition, OTL allows the modeler to reason about time in a quantitative fashion. OTL is a part of a UML-based formalism, so it is totally integrated with the other UML notations. As far as the OCL 2.0 standard library is concerned, OTL extends it by adding three new classes, `Time`, `Duration` and `Interval` (see Fig. 1). Class `Time` models time instants, which are defined based on the current time taken as the time origin. Class `Duration` models duration of time intervals, i.e., the distance between two time instants. Therefore, a time `Interval` can be defined by its initial `Time` instant and its `Duration`. These classes inherit directly from class `OclAny`, which is the root of the hierarchy of the base types in OCL 2.0. The existence of both class Time and class `Duration` allows for a conceptually proper treatment of time and the definition of sensible operations between objects of the two classes. For

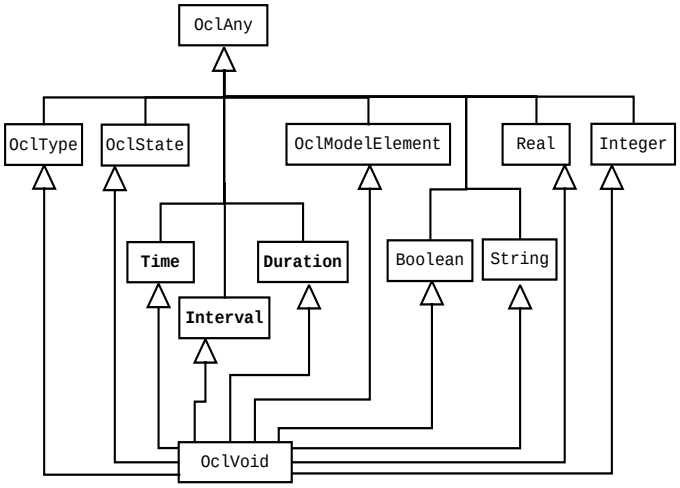

Fig. 1. The OCL standard library extended with types Time, Duration and Interval.

instance, class `Time` provides (1) an operation that checks the ordering between its objects, so we can say if a time instant precedes or follows another time instant, (2) an operation for finding the time distance between two instants, which returns an object of class `Duration`, and (3) an operation that takes a parameter `d` of class `Duration` and returns the `Time` object that lies at a time distance `d`. Class `Duration`, for instance, has sum and subtraction operations between its objects: the sum of two time distances is a new time distance, whose extension is the sum of the extensions of the original time distances. For instance, class `Interval` has operations for (1) finding if two intervals overlap, (2) finding if one interval contains another interval, and (3) building the union of two overlapping intervals. These operations allow modelers to use quantitative time. Depending on the application at hand, `Time` and `Duration` may be discrete or dense.

OTL formulas are evaluated with respect to an implicit current time instant. To allow for the evaluation of a predicate `p` at a time different than the current one, OTL introduces a new primitive as a method of class `Time`, in a way that is consistent with the OCL notation. Thus, given a time instant t, represented as an object `t` of class `Time`, the evaluation of `p` at time `t` is carried out as follows:

```
t.eval(p)
```

Method `eval` receives an OclExpression as the parameter (`p`) and returns a boolean value. Its meaning is that predicate `p` is evaluated at time `t`. With reference to the definition of OCL 2.0 [10], it is interesting to note that our extensions do not require any change in the metamodel. Types `Time`, `Duration` and `Interval` are simply three new types that are added to the OCL standard library as specializations of `OclAny`. The full definition of these classes is not reported here for space reasons. Based on method `eval`, all other temporal operators can be defined. Since `Time` is introduced as an OTL type, collections

of objects of class `Time` can be defined. For instance, if `T` is a set of objects of class `Time`, formula `T->forall(t: Time|t.eval(p))` is true if and only if `p` is true at all time instants in `T`. To provide modelers with expressive tools to describe time-critical systems, it is useful and convenient to define a set of temporal operators. For instance,

```
context C
  inv: Lasts_ie(p, t1, t2)
```

specifies that p holds in the interval from t1 (included) to t2 (excluded). This statement can be defined as a shorthand for

```
context C
  inv: let I: Interval = [t1..t2] in
          I->forall(t: Time| t=t2 or t.eval(p))
```

Note: here we use `forall`, which in OCL is defined only for collections. OTL classes `Interval` and `Duration` represent sequences that may possibly contain infinite time instants. Therefore we define methods `forall` and `exists` in this classes as well. Other operators can be defined similarly. For instance:

```
context C
  inv: Until(p1, p2)
```

is equivalent to:

```
context C
  inv: let I: Interval = [now..inf] in
      I->exists(t: Time | Lasts_ie(p1,now,t) and t.eval(p2))}
```

where `now` represents the current (evaluation) instant, while `inf` indicates the infinite. In addition, even the basic temporal operator `t.eval(p)` may be denoted with the more convenient and intuitive syntax `p@t`. This syntax is similar to the one used in standard OCL [1], where `p@pre` denotes the fact that `p` holds before a method is executed.

4 The development process: methodological guidelines

The description of real-time systems according to the proposed notation is meant to support a development process like the one described in Fig. 2. The fundamental idea is that the description of the software system, of the environment in which it operates, and of the user requirements is complete and precise enough to let the developers:

- Validate the specification by simulating it (at least with respect to its time behavior).
- Verify the specification through model checking, to ensure that a model satisfies the properties specified with OTL (i.e., the provided statecharts are a valid "implementation" of the system behavior).

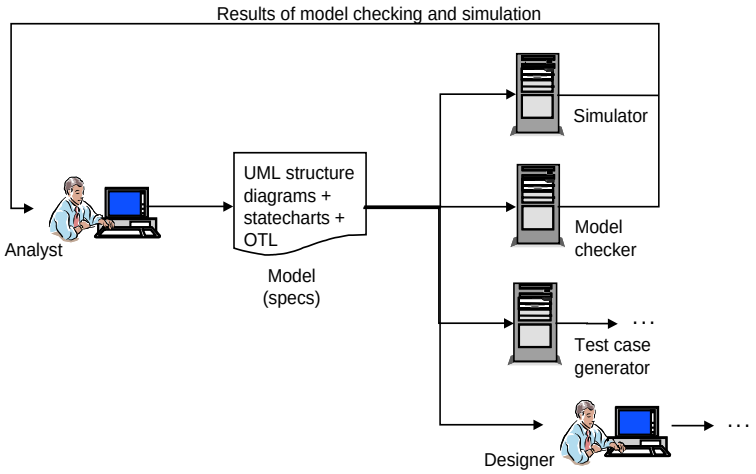

Fig. 2. The development process.

- Support verification by generating functional test cases that can guarantee (to a reasonable extent) that the implementation of the system is consistent with its specifications.

Some of the activities described above could require some sort of translation. For instance, models written in our notation cannot be fed directly to existing model checkers. In most cases, it should not be difficult to translate OTL statements into, say, TCTL [2] or other notations as required by the model checkers. However, as OTL is more expressive than TCTL, proving that a property of a system specified with OTL holds could be undecidable. In these cases, one could apply other techniques, like theorem proving or history checking [4]. The latter approach would be preferable, as long as it does not require particular skills in logics. In any case, as a fundamental prerequisite for the application of the process depicted in Fig. 2 the system model must be defined using the proposed notation. Guidelines for this step are therefore needed. Of course, every analyst is free to define the model following the procedure he or she likes best. However, a few reference points may be useful:

- The capsules reported in the collaboration diagrams represent the "domains" of the problem and the elements of the solution. These can be identified and described according to various methods, for instance the "problem frames" proposed by Michael Jackson [7].
- The behavior of each capsule can be described by statecharts so as to represent environment and system behavior. They take into account the indications provided by the domain experts, the users, and the analysts. It is important to note that the described behavior is in general the result of decisions taken by the users and the analysts together.
- The OTL specifications represent user requirements at the most abstract level. In practice they often concern the behavior of the system (including the "machine") from the user point of view. For instance, OTL statements

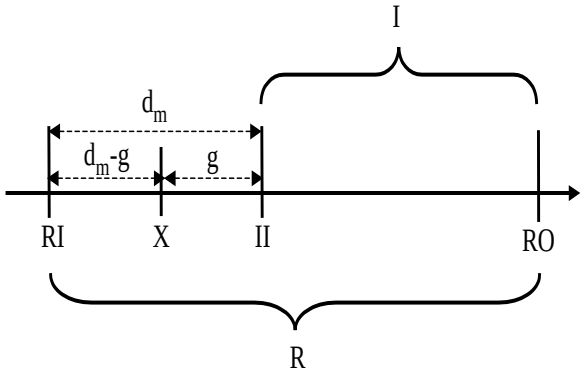

Fig. 3. GRC regions of interest.

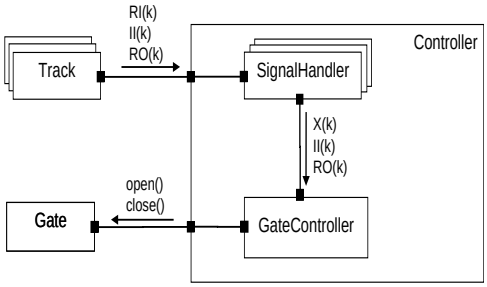

Fig. 4. GRC: the physical components and their connections.

will generally concern also elements that are not directly visible by the "machine". Here, the term "machine" indicates a computer or a network of computers, or more generally the IT solution provided by the implementers.

5 A case study

We illustrate our approach through a classic case study in the literature of embedded, time critical systems: the generalized railroad crossing (GRC) [6]. The system operates a gate at a railroad crossing I, which lies in a region of interest R (see Fig. 3). Trains travel through R on K tracks in one direction (it has been proved that having trains traveling in both directions does not change the complexity and relevance of the case study). Trains can proceed at different speeds, and can even pass each other. Only one train per track is allowed to be in R at any moment. Sensors indicate when each train enters and exits regions R and I. Point RI and RO indicate the position of the entrance and exit sensors for region R. II indicates the position of the sensor which detects trains entering region I. d_m and d_M are the minimum and maximum times taken by a train to cross RI-II zone. h_m and h_M are the minimum and maximum times taken by a train to cross zone I. g is the time taken by the bars of the gate to move from the completely open to completely closed position (or vice versa).

The system must be safe, i.e., the gate must be closed when trains are

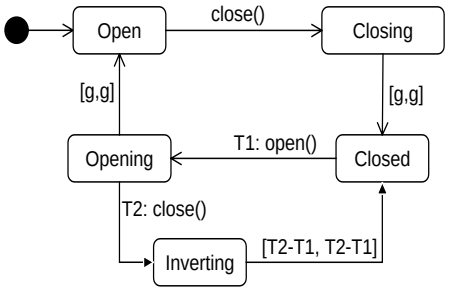

Fig. 5. The statecharts describing the behavior of the gate.

crossing the I region. Other required properties are discussed later.

The structure of the system is illustrated in Fig. 4. Note that the syntax employed here is not exactly conformant to UML, nevertheless, the correspondence with UML constructs (capsules, ports, etc.) should be evident.

The controller receives the RI, II, and RO signals from the sensors positioned on the tracks, and sends open and close commands to the gate. RI, II, and RO signals have a parameter indicating the identity of the source track. For space reasons, we give here already the internal structure of the controller, while in general it would be specified later, as a consequence of the required behavior of the whole system.

The gate behaves as described by the statechart reported in Fig. 5. It is possible to note that we have introduced in the statecharts a few non standard elements that are useful to deal with time and time constraints. In particular, some transitions are associated with labels that indicate at what time the transition occurred: for instance, the transition from *Closed* to *Opening* occurred at time T1. It is possible to specify time-dependent conditions for transitions: in particular, it is possible to constrain the occurrence of a transition Tr in an interval $[T_{low}, T_{up}]$, where T_{low} and T_{up} are relative to the time the system entered the source state of Tr. In our example the gate is closed exactly g time units after receiving the *close()* signal. It is also possible to mention time labels in the conditions: in our example if the gate has been opening for a time T_o and a *close()* signal is received, then the gate takes exactly T_o to reach the *Closed* state again. This is specified by "remembering" the instants T1 and T2 when the *open()* and *close()* signals are received (respectively), and constraining the system to reach state *Closed* exactly T2-T1 time units after the *close()* signal is received. Tracks behave as described in Fig. 6 (as a consequence of the behavior of trains). The controller is internally structured in two components: a signal handler and a gate controller. The former passes immediately the signals II and RO to the gate controller, while it generates a signal X $d_m - g$ time units after receiving a RI signal (see Fig. 7). The gate controller reacts to X signals, so that the gate gets completely closed exactly when the fastest trains enter the crossing zone. More precisely, the gate controller counts the trains that are in the R and I zones, by means of counters ccr and cir, respectively: whenever ccr becomes greater than zero a *close()*

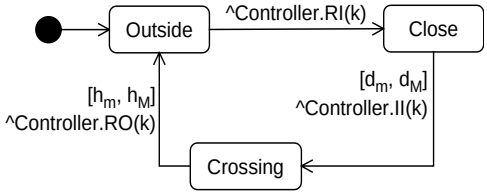

Fig. 6. The statecharts describing the behavior of the trains.

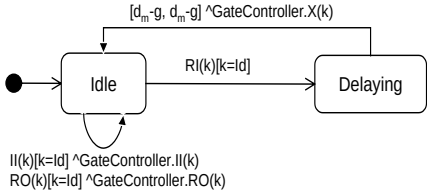

Fig. 7. The statecharts describing the behavior of the signal handler.

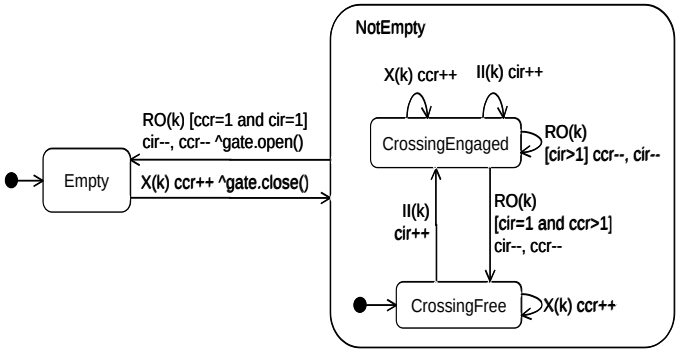

Fig. 8. The statecharts describing the behavior of the gate controller.

command is issued, when it returns null, an *open()* command is issued (see Fig. 8). The safety property can be expressed as follows:

```
context Gate
inv:
  self.Tracks->exist(oclInState(Crossing)) implies
        self.oclInState(Closed)}
```

This property does not make explicit reference to time, so it can be expressed using plain OCL. However, a gate that is always closed would satisfy the safety requirement. Therefore we would like to express also a utility property, i.e., the gate must be open if no train has been in the region R for an interval of g time units. This property can be expressed as follows:

```
context Gate
inv:
  self.Tracks->forall(Lasted_ei(oclInState(Outside),g))
     implies self.oclInState(Open)}
```

This property does not take into account that a train may be out of R for part

of an interval with duration g, and for the rest of the interval the train could be in the RI-Y region, where Y is a point in R preceding X. If this is the case, the gate should be open as long as the train is in Y-X. This property can be expressed as follows:

```
context Gate
inv:
  let D: Duration = [now-g, now] in
    D->exists(T: Time | self.Tracks->forall(
         Lasted_ei(oclInState(Outside),g) or
         (Lasted_ei(oclInState(Close),now-T) and
          (now-T)<(dm-g) and
          Lasted_ee(oclInState(Outside),now-g, now-T))))
     implies self.oclInState(Open)}
```

In the OTL statement above `now` indicates the current instant (i.e., if `inv` means $\forall$t, then `now` is t). Lasted(P,t1,t2) means that P was true in the interval [t1,t2], thus Lasted(P,d) = Lasted(P,now-d,now). Note that Lasted(P,k) is true for every k<0, and Lasted(P,t1,t2) is true whenever t2<t1.

Note that the OTL statements that express the safety and utility properties refer only to elements of the gate and the tracks, which belong to the problem domain, so they can be considered as part of the requirements specifications. The statechart of the controller (Fig. 8) can be considered the result of a preliminary design activity. The process sketched in Fig. 2 suggests that the model including the statechart of the controller should be verified with respect to the safety and utility properties by means of a model checker. Once it is proved that the model is correct, implementation can start. The component-based structure of the model provides a starting point for a component-based implementation of the system.

6 Conclusions

The development of real-time critical applications calls for a specific process and rigorous notation. We propose a "dual language" approach, where UML provides the constructs for modeling the structure of the system and the behavior of the system's components. A new descriptive language based on temporal logic, called OTL (Object Temporal Logic), allows the developer to assert properties of the system at an abstract specification level.

The literature already contains a few proposals for augmenting OCL to deal with time-dependent systems (see [8]). Some of them (e.g., [3]) deal with time only from a qualitative viewpoint, i.e., no notion of temporal distance between events is provided. Another proposal allows modelers to deal with time in a quantitative fashion, by extending the set of operators of OCL [5]. However, in the latter approach a discrete time is associated with the system states and events, while in our approach real values can be used to represent

time.

References

[1] OMG, *Unified Modeling Language Specification 1.4*, September 2001. URL: `http://www.omg.org`.

[2] Alur, R., Courcoubetis, C., and Dill, D. L., *Model checking in dense real time*, Information and Computation, Vol.104, N.1. (1993).

[3] Cengarle, M.V., and Knapp, A., *Towards OCL/RT*. In Proc. 11th Int. Symp. Formal Methods Europe, Berlin 2002, Springer LNCS 2391.

[4] Felder, M., Morzenti, A., *Validating real-time systems by history-checking TRIO specifications*. ACM TOSEM-Transactions On Software Engineering and Methodologies, vol.3, n.4 (1994).

[5] Flake, S., Mller, W., *An OCL Extension for Real-Time Constraints*. In T. Clark and J Warmer (Eds.), Advances in Object Modelling with OCL, Springer Verlag, 2001.

[6] Heitmeyer C.L., Jeffords R.D., Labaw B.G., *Comparing different approaches for Specifying and verifying Real-Time Systems*, in Proc. 10th IEEE Workshop on Real-Time Operating Systems and Software, New York (1993), 122–129.

[7] Jackson, M., "Problem Frames - analysing and structuring software development problems", Addison-Wesley ACM Press, 2001.

[8] Rammig, F.J., *OCL Goes Real-Time*. In Proc. of the Fifth IEEE Int. Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'02), IEEE (2002).

[9] Selic B., Gullekson G., Ward P. T., "Real-Time Object-Oriented Modeling", Wiley, 1994.

[10] Response to the UML 2.0 OCL RfP (ad/2000-09-03) Revised Submission, Version 1.5, June 3,2002, OMG Document ad/2002-05-09.

[11] Unified Modeling Language: Infrastructure, version 2.0, Updated submission to OMG RFP ad/00-09-01, September 2002.