A Dual Language Approach to the Development of Time-Critical Systems

Luigi Lavazza [1],[2]

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

Sandro Morasca[3]

Dipartimento di Scienze della Cultura, Politiche e dell'Informazione
Università degli Studi dell'Insubria
Como, Italy

Angelo Morzenti[4]

Dipartimento di Elettronica e Informazione
Politecnico di Milano
Milano, Italy

Abstract

Developing time-critical systems requires expressive, rigorous, easy to use notations to describe the time-related features of the systems, in a way that is formal enough to support and automate activities like property verification and test case generation. We propose a dual-language approach provided with a descriptive formalism for specifying the properties of a system and its components in addition to the typical UML (and UML-RT) diagrams. This description consists of a formula of a new logic, called OTL (Object Temporal Logic), which is an extension of OCL. The approach is applied to a case study derived from the authors' industrial experiences.

Keywords: time-critical system, verification, testing, UML, OTL, OCL

[1] Thanks to everyone who should be thanked
[2] Email: lavazza@elet.polimi.it
[3] Email: sandro.morasca@uninsubria.it
[4] Email: morzenti@elet.polimi.it

1 Introduction

The development of time-critical systems requires the availability of notations that are expressive, rigorous, easy to use, and provided with software tools at the same time. Time-critical software systems are usually complex and need to be modeled and analyzed from several different perspectives, such as their functional behavior, their temporal behavior, and their structure. In the last few years, UML [1] has been increasingly used for the development of complex systems such as real-time software, even though UML was not originally conceived for modeling real-time systems. Only recently were timing features added to the UML notation (see for instance the introduction of Time in the proposal for UML 2.0 [4]), but their introduction is still tentative, incomplete, and not well integrated with the other aspects of UML. So, the practical application of UML to the real-time domain is hindered by UML's lack of a complete set of constructs to express time-related constraints and properties, as well as by its lack of formal semantics. An adequate solution to these problems will need to go one step forward and provide high rigor of syntactic and especially semantic definition, as well as high integration and consistency with the rest of the UML notation. Moreover a high level of abstraction is needed, so that the notation can be used in phases in which high-level properties of a system are described, but not its inner functioning.

In this paper, we propose an extension to UML for introducing timing aspects to address these problems via a set of carefully thought and balanced time-related notations that are integrated and consistent with UML notation, so they can be used by practitioners in industrial environments with minimal overhead and can support suitable development methods for time-critical systems.

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 mutual inclusion and communication, via a small set of fundamental constructs: capsules correspond to components; ports and protocols model abstract interfaces; and connectors correspond to communication relations. The partitioning of a complex system into a set of parallel components (i.e., parts) that communicate via connectors results in a composition hierarchy, where the leaves correspond to the components that are directly modeled in an operational style with a state-transition machine.

We also propose a descriptive formalism to specify the properties of a system and its components, whose style is thus complementary to that of the leaf-level capsules statecharts. This description consists of a formula of a new logic, OTL (Object Temporal Logic), which is fully compatible with the original OCL (Object Constraint Language) descriptive notation for asserting

properties in UML.

In our proposal of a "dual language" approach, OTL formulas and statecharts are also complementary at the methodological level. 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, so there is no redundancy between the information provided by the OTL formula and the statechart.

Our proposal is based on general concepts that appear to comply both with the consolidated versions of UML and OCL, and with the directions of the draft proposals [2,3,4].

This paper is organized as follows. Section 2 describes OTL, Section 3 describes the application of our dual-language approach to a case study, while Section 4 concisely compares our approach with the ones existing in the literature. Conclusions are in Section 5.

2   The OTL language

OCL 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 the one proposed in [2] for OCL 2.0) needs to be extended to adequately specify temporal aspects. It is not possible to reference different time instants in a single OCL formula, so only invariant properties can be formalized, which at most include references to attribute values before or after method execution. Important temporal properties of systems that make reference to the time distance between events cannot be adequately specified, thus making it impossible to specify that the response to a stimulus must be guaranteed to occur within some specified time interval.

We propose 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 totally integrated with the other UML notations: it simply extends the OCL 2.0 standard library by adding two new classes, `Time` and `Offset` (see Figure 1) which directly inherit from class `OclAny`, and no changes in the metamodel are required. Class `Time` models time instants, which are defined based on the current time taken as the time origin. Class `Offset` models the distance between two time instants. An `Offset` `d` that is added to a `Time` object (see below the '+' operator for class `Time`) is interpreted as a displacement towards the future if `d` is positive, towards the past if `d` is negative. Other basic time-related concepts, such as the notion of a time interval can be easily

defined in terms of the concepts of `Time` and `Offset`.

The existence of both classes `Time` and `Offset` allows for a conceptually sound quantitative treatment of time and the definition of sensible operations involving objects of the two classes. For instance, class `Time` provides (1) an operation '$\leq$' that checks the ordering between its objects; (2) an operation '`dist`' for finding the (positive, null, or negative) time distance between two `Time` objects, which returns an object of class `Offset`; (3) an operation '+' that takes a parameter d of class `Offset` and returns the `Time` object that lies at a time distance d in the future if d is positive or in the past if d is negative; and (4) an operation called `futrInterval` that takes a parameter of class `Offset` and returns a `Collection` all of `Time` points within a distance d in the future (symmetrically, the operator `pastInterval` returns the `Collection` of all `Time` points within a distance d in the past). Class `Offset` has sum and subtraction operations between its objects.
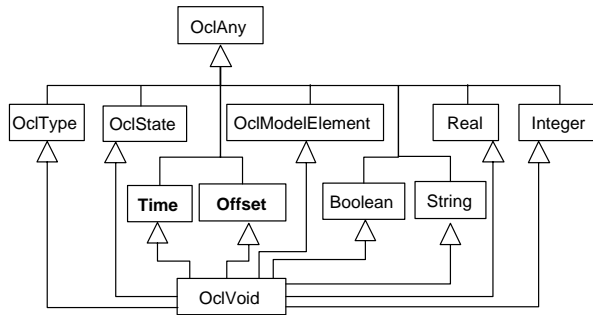

Fig. 1. The OCL standard library extended with types Time and Offset.

`Time` and `Offset` may be discrete or dense, depending on the application at hand. From a methodological viewpoint, continuous time is useful when modeling the evolution over time of intrinsically continuous physical entities (e.g., a temperature or a voltage) that are external to the device or system under development and that must be monitored or controlled. The use of continuous entities is indispensable even for just expressing the user requirements, and *a fortiori* for analyzing and proving their satisfaction in the System Requirements analysis [6]. On the other hand, discrete time will suffice to model parts corresponding to digital, synchronous devices and in general in the UML artifacts related with detailed specification, design and implementation of the device under development.

The adoption of a possibly dense time has implications on the semantics of the OTL language, because OCL assumes (see [2], Appendix A on semantics) that quantified variables range only over *finite* sets and defines the meaning of quantification in terms of finite iterations, like in the iterative statements

of programming languages. In the OTL language, instead, the semantics of quantification over time cannot be based on finite iteration, but must be defined in the same way as in more conventional mathematical logics that include arithmetic. We do not expect any technical difficulty in providing this kind of semantics for OTL, but we do not include this in the present work, mainly for space reasons, and leave it as a further development.

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 from the current one, OTL introduces –consistent with the OCL notation– a new primitive as a method of class `Time`. Method `eval` receives an `OclExpression` as the parameter (`p`) and returns the (boolean) value of `p` at time `t`. This is denoted as `t.eval(p)` or, more concisely, as `p@t`.

All other temporal operators can be defined based on method `eval`. In particular, properties can be expressed on collections of objects of class `Time`, i.e., on time intervals. For instance, formula `context C inv:  Lasts(p, d)` specifies that `p` holds in the interval lasting `d` time units from the current time, as defined in Table 1.

A number of operators can be likewise defined to refer to the future (e.g. `Futr`, `SomF`, `AlwF`, `WithinF`, `Until`, whose intuitive meaning and formal definitions are in Table 1, where `inf` denotes the infinite `Offset` value) and the past (e.g., the corresponding operators `Past(p,d)`, `SomP(p)`, `AlwP(p)`, `WithinP(p,d)`, and `Since(p,q)`). Even though they do not add expressive power, it is widely recognized that operators referencing the past make shorter, more readable, and more intuitive specifications possible.

For operators that refer to time intervals we add a suffix to indicate explicitly if the extremes of the interval are included; we use the letter 'i' to denote inclusion, and letter 'e' to denote exclusion, so formula `Lasts_ie(p,d)` states that property `p` holds from `now` (included) to `now+d` (excluded).

3   A Case Study

We illustrate our dual language approach with a fragment of the specification of a digital energy and power meter, developed for the Italian Energy Board [11] in the TRIO object oriented temporal logic language [7]. This device is certainly critical, although not "safety-critical", because it is installed in millions of copies, so its precision and reliability are crucial. The meter is composed of a magnetic transducer (called G_Ferraris after the name of its inventor) that converts the electric energy flowing through the line into the rotation of a disk. In the peripheral part of the disk, transparent and opaque

operator	intuitive meaning	formal definition
Lasts(p,d)	for d time units in the future	now.futrInterval(d)->forall(t: Time | t.eval(p))
Futr(p,d)	d time units in the future	p@(now + d)
SomF(p)	sometimes	let I: Set(Time) = now.futrInterval(inf) in I->exists(t: Time | p@t)
AlwF(p)	always	let I: Set(Time) = now.futrInterval(inf) in I->forall(t: Time | p@t)
WithinF(p,d)	within d time units	let I: Set(Time) = now.futrInterval(d) in I->exists(t: Time | p@t)
Until(p,q)	p holds until q occurs	let I: Set(Time) = now.futrInterval(inf) in I->exists(t:Time | q@t and Lasts(p,t-now))

Table 1
Derived operators.

portions are evenly alternated, so the disk position and velocity (which are respectively proportional to energy and power consumption) can be detected by a photocell, as shown in Figure 2 (a).
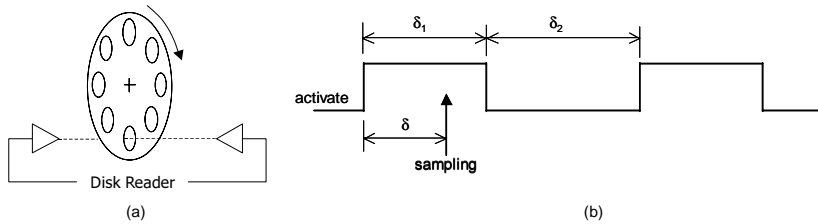

Fig. 2. (a) Rotating disk and photocell; (b) Activation and sampling of photocells.

To minimize its wear, the photocell is activated only for a small fraction of the total working time of the meter, as shown in Figure 2 (b). Once the photocell is activated, its signal is sampled with a delay δ, to permit it to reach a stable state. The consumption of an energy quantum is detected when the disk moves from a transparent portion to an opaque one, or vice versa.

A device called "Reader" issues the sampling command for the photocells and detects the full/empty position of the disk from the reading of the photocell signal. A further device, called CostAssign, determines the cost for the client of each consumed quantum of energy, based on the current time, date, and applicable tariff, provided by two other components called Tariff

and Calendar. A final device, called Totalizer, computes the total amount of the invoice to be sent periodically to the client. The overall structure of the energy meter is shown as a simplified version of a UML-RT collaboration diagram in Figure 3, where we have omitted unnecessary details for the sake of readability. We have also adopted the convention of giving the same name to pairs of connected ports.

The environment of the energy meter is represented in Figure 3 by capsule Environment, which provides the meter with the stimuli, i.e., amount of energy used and noise, which are general functions of time, with the provision that the energy used is monotonically nondecreasing and the noise is limited in absolute value, as specified formally later in the paper.
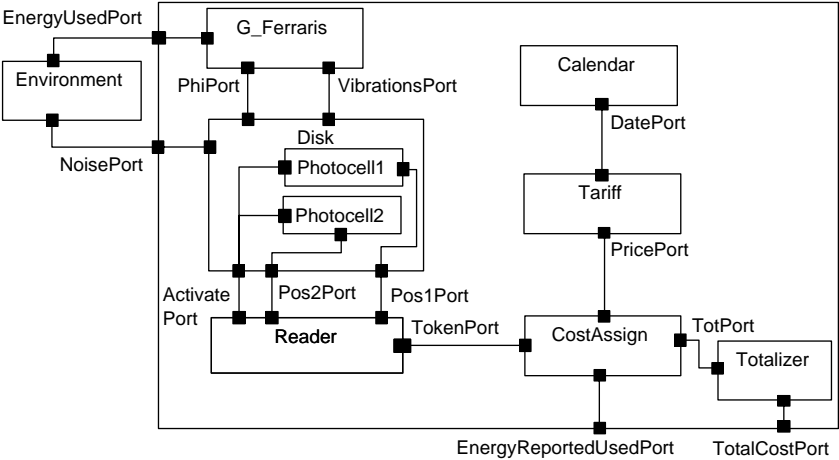

Fig. 3. Collaboration diagram of the system.

The device is subject to vibrations, which may cause minimal changes in the position of the disk even if no energy is being consumed and the disk should be perfectly still. These spurious transitions are filtered out by the Reader device via a second photocell placed at an angular distance from the first one equal to $\gamma/2$, where γ is the angle of each opaque or transparent peripheral region of the disk, as shown in Figure 4 (a). This ensures that only one of the two photocells may generate spurious transitions, so the signals from the second photocell can be used to confirm transitions detected by the first one: a rising edge (i.e., a switch from "empty" to "full" signal) from the first photocell is confirmed by the subsequent rising edge of the second one, and similarly for the falling edge, as shown in Figure 4 (b).

The class diagram of the system is given in Figure 5. For space reasons we have omitted protocol definitions, which can be easily inferred by the reader. The G_Ferraris rotates the disk so that the angle of the disk is always pro-
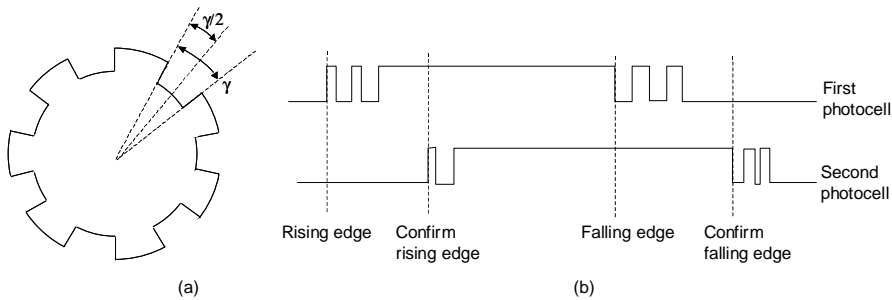
Fig. 4. (a) Opaque and transparent regions on the disk; (b) Confirmation of edges.
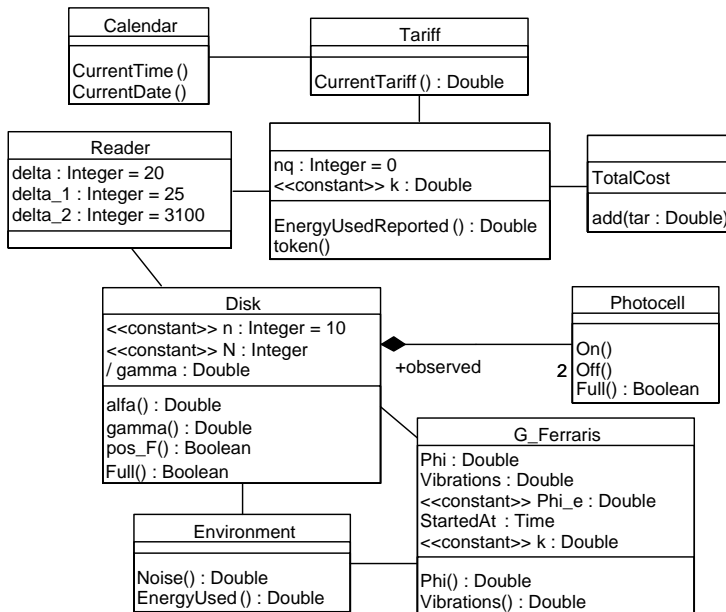
Fig. 5. Class diagram of the system.

portional to the energy used. It may also generate spurious vibrations, whose amplitude is known to be limited. Such behavior can be specified in a purely declarative way by means of the following OTL statements:

```
context G_Ferraris
    inv: abs(self.Vibrations) <= Phi_e
    inv: now >= StartedAt implies
        Phi = Phi@StartedAt + k * EnvironmentPort.Energy_used()
```

where `Phi` represents the rotation angle of the disk, `Phi_e` is the maximum amplitude of vibrations, `k` is the constant ratio between `Phi` and the energy used. Attributes `Vibrations` and `StartedAt` represent the generated vibrations and

the time at which the G_Ferraris was activated, respectively.

The disk is characterized by `n`, the number of transparent sectors of the disk; `gamma`, the size (expressed as an angle) of each sector (see Figure 4 (a)); `N`, a coefficient used to make the position of the disk independent from the model of meter considered. `gamma` can be defined as follows (in plain OCL):

```
context Disk inv: gamma=3.14159/self.n
```

Class `Disk` is equipped with method `alfa()`, which computes the sum of the vibrations, noise and the normalized angular position `Phi`, i.e., what is observed by the photocells. Operation `alfa()` can be formalized as follows:

```
context Disk::alfa():Real
 pre: True
 post: Result = EnvironmentPort.Noise() +
          PhiPort.Phi()/self.N + VibrationsPort.Vibrations()
```

`pos_F()` is also a method of the disk: it is a boolean function that states if the disk (i.e., `Phi`) is in a position that will be read by the photocell as full, and false otherwise. It is formalized by the following OCL statement

```
context Disk::pos_F():Boolean
 pre: self.oclInState(Active)
 post: let X:Double = mod(alfa(),(2*gamma)) in
             (0 <= X and X < gamma) and Result = True) or
             (gamma <= X and X < 2*gamma) and Result = False)
```

where `mod` is the modulo operation.

The behavior of the photocells associated with the Disk is specified by the very simple statechart reported in Figure 6 (a).
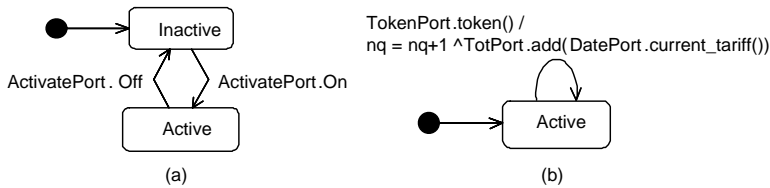
Fig. 6. Statecharts of class photocell (a) and class CostAssign (b).

For the photocell it is important to specify the `Full()` method:

```
context Photocell::Full():Boolean
  pre: self.oclInState(Active)
  post: (observed.pos_F()and Result = True) or
        (not observed.pos_F()and Result = False)
```

The photocell can be asked to provide the position only if the cell is active. The Reader has to accomplish two main tasks: to activate the photocells periodically, and to detect transitions from a transparent sector to an opaque sector and viceversa. The behavior of the `Reader` class is modeled by the statechart in Figure 7, which implements the strategy for filtering out spurious transitions. Let `full1` and `empty1` be predicates denoting the detection of opaque or transparent regions on the first photocell (`full2` and `empty2` are defined in a similar way).

```
context Reader
  def full1: Boolean = Pos1Port.Full()
```

```
def empty1: Boolean = not Pos1Port.Full()
```

The rising and falling edges on the first photocell are defined by predicates `risingEdge1` and `fallingEdge1` in the following formulas (`risingEdge2` and `fallingEdge2` are defined similarly for the second photocell):

```
def risingEdge1: Boolean = full1 and Since(not full1, empty1)
def fallingEdge1: Boolean = empty1 and Since(not empty1, full1)
```

Predicates `confirmedRisingEdge` and `confirmedFallingEdge`, are defined in terms of the previous predicates as follows:

```
def confirmedRisingEdge: Boolean =
        risingEdge2 and Since(not risingEdge2, risingEdge1)
def confirmedFallingEdge: Boolean =
        fallingEdge2 and Since(not fallingEdge2, fallingEdge1)
```

Finally, the detection of an energy quantum occurs at every "confirmed edge" – whether rising or falling– and results in sending a `token` message to `CostAssign` through the `TokenPort`. This is specified in OTL as follows:

```
inv: TokePort^token() = (confirmedRisingEdge or confirmedFallingEdge)
```

Message sending can be indicated in OCL only in post-conditions (as the time scope of the event is the execution of an operation). In OTL we can define precisely when the message sending occurs, so we are not constrained to use this construct only in post-conditions.

The OTL formula above is part of the model of the `Reader` component of the energy meter, and contributes to specify its behavior, implemented through a statechart (Figure 7).

Fig. 7. Statechart of class Reader.

The activation of the photocells can be specified by means of OTL statements

like the following (where `delay` is some constant):

```
context Reader
 inv: ActivatePort^On()@StartedAt+delay
 inv: ActivatePort^On()@now implies
   (not ActivatePort^Off()@now and
    Lasts_ee(not (ActivatePort^On() or ActivatePort^Off()), delta1) and
    ActivatePort^Off()@now+delta1)
 inv ActivatePort^Off()@now implies
   (not ActivatePort^On()@now and
    Lasts_ee(not (ActivatePort^On() or ActivatePort^Off()), delta2) and
    ActivatePort^On()@now+delta2)
```

`Reader` provides messages `token` to the `CostAssign` unit, which uses them as specified by the statechart in Figure 6 (b). Every time the `CostAssign` unit receives a token, it increments the variable `nq`, which represents the total number of token received, and calls the `add` operation of the `Totalizer` unit, specifying the current tariff, based on the current date and time.

The total amount of the energy consumed is computed by multiplying the number of tokens `nq` by a constant `k` (the energy "quantum"). The `Totalizer` unit is thus able to compute the price of the energy consumed.

A few global properties of the model can also be expressed with OTL.
The amount of used energy reported by the device is monotonically nondecreasing. The following OTL statement describes this requirement via the usual definition of monotonicity, where `D` (the length of the interval) is a constant value defined in the context of class `CostAssign`:

```
context CostAssign
 inv: let DS = Set(Offset) = [0..D] in
         DS -> forall(d: Offset |
            EnergyUsedReported() >= past(EnergyUsedReported(), d))
```

The cost of the energy consumed at constant tariff increases proportionally to the consumed energy and the tariff. Given an arbitrary time interval of length IL, in which the applicable tariff is constant, the variation in the total cost of the consumed energy is the product of the tariff and the energy consumed during interval IL:

```
context Totalizer
 inv: let TS = Set(Integer) = [minTariff .. maxTariff] in
   TS -> forall (tr: Integer | Lasted(Tariff.CurrentTariff()=tr,IL)
   implies   TotalCost - past(TotalCost,IL) = tr*
                         (CostAssign.EnergyUsedReported()-
                          past(CostAssign.EnergyUsedReported(),IL)))
```

The difference (in absolute value) between the energy reported used and the energy actually used over any time span of a predefined constant length `TSL` *(say, a week or a month) is invariably less than the energy corresponding to a quantum, say* `quantumEnergy`:

```
context CostAssign
 inv: abs(EnergyUsedReported()-past(EnergyUsedReported(),TSL)-
   (Environment.EnergyUsed-past(Environment.EnergyUsed,TSL)))
   < quantumEnergy
```

This property guarantees that the consumer does not have to pay more than the due, while the energy company does not get paid less than due.

These global properties, as well as any property possibly attached to component capsules, can be used in the analysis and verification of the system.

4   Review of the literature

A few proposals have appeared in the recent literature to introduce timing features in UML in a rigorous, consistent way. A number of these do not deal with metric time, so we do not review them here. Among the proposals that explicitly deal with metric time, a few representative ones may be considered. Flake et al. [8] provide a state-oriented temporal extension to OCL. A formal concept of state sequence is introduced, on top of which temporal properties are specified. The syntax is kept consistent to the OCL syntax. However, the authors themselves note that the OCL syntax may be somewhat clumsy, and different constraint languages may be used, provided that a translation mechanism into OCL is used. Dense time is used by Roubtsova et al. [9], whose approach affects class diagrams and statecharts diagrams, but not OCL, based on the idea that providing OCL with a concept of path would be outside the framework of OCL. Temporal properties are expressed as a so-called specification class, associated with a formal constraint. A specification class is in specification relationship with an actual class. An interesting aspect is that most approaches tend to use a syntax based on temporal logic, instead of an OCL-like syntax. Sendall and Stroheimer [10] introduce timing features on UML statecharts again and provide five kinds of properties to capture the temporal aspects of the specifications of a time-dependent system, namely the times at which events may occur, the durations of activities, and the frequency of state transitions, so the approach can be used for specifying real-time and performance properties. None of the above mentioned approaches, however, provides a logic that allows the designer to properly describe the system requirements without referencing elements of the operational model explicitly.

The OMG document "UML Profile for Schedulability, Performance, and Time Specification" [5] defines the standard way of introducing time into UML models. However, it seems that the whole document was conceived mainly to support design and implementation, rather than specification. Accordingly, it provides *mechanisms* –like clocks and timers– for dealing with time in an operational style, but does not address the problem of specifying the properties of the system at an abstract level. Using this profile, one can write statecharts that specify the time behaviour of the system in reaction to events generated by clocks and timers. This is a suitable way of describing implementations,

but is hardly applicable in the problem domain, where timers and clocks are not present.

5   Conclusions

The development of real-time critical applications calls for a specific process and rigorous notation. We propose a "dual language" approach: the structure of the system and the behavior of the system's components are modeled via UML, while a new descriptive language based on temporal logic, called Object Temporal Logic, allows the developer to assert properties of the system at an abstract specification level. Our proposal supports a systematic and rigorous development, centered on explicit, possibly formal requirements specification, and requirement validation and verification through analysis, possibly in the form of property proving via deductive methods or model checking.

References

[1] OMG, "Unified Modeling Language Specification Version 1.5", March 2003, formal/03-03-01. URL: http://www.omg.org.

[2] "Response to the UML 2.0 OCL RfP, Revised Submission,Version 1.6", January 6, 2003, OMG Document ad/2003-01-07.

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

[4] "Unified Modeling Language: Superstructure version 2.0, 3rd revised submission to OMG RFP ad/00-09-02", OMG document ad/2003-04-01, 10 April 2003.

[5] "UML Profile for Schedulability, Performance, and Time Specification", OMG Adopted Specification, ptc/02-03-02, March 2002.

[6] Gargantini, A. and Morzenti, A., *Automated Deductive Requirements Analysis of Critical Systems*, ACM TOSEM, **10** (2001), 225–307.

[7] Morzenti, A. and San Pietro, P., *Object-Oriented Logic Specifications of Time Critical Systems*, ACM TOSEM, **3** (1994), 56–98.

[8] Flake, S. and Mueller, W. *Formal semantics of static and temporal state-oriented OCL constraints.* SoSyM 2(3), October 2003.

[9] Roubtsova, E., Van Katwijk, J., Toetenel, W. and De Rooij, R., *Real-Time systems: specification of properties in UML*, in Proceedings of ASCI 2001, May 2001, 188–195.

[10] Sendall, S., and Strohmeier, A., *Specifying concurrent system behaviour and timing constraints using OCL and UML*, in Proc. of UML 2001, LNCS 2185, October 2001, 391–405.

[11] Morzenti, A., Paci, M. and Veroni, F. "Specificha TRIO dell'Unità di Elaborazione Periferica", 19/1/93, Draft version 3, In Italian.