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

Luigi Lavazza\textsuperscript{1,3}, Sandro Morasca\textsuperscript{2}, Angelo Morzenti\textsuperscript{1}

\textsuperscript{1} Politecnico di Milano, Dipartimento di Elettronica e Informazione, P.zza Leonardo Da Vinci, 32, 20133 Milano, Italy
[lavazza,morzenti]@elet.polimi.it

\textsuperscript{2} Dipartimento di Scienze Chimiche, Fisiche e Matematiche, Università degli Studi dell’Insubria, Via Valleggio 11, I-22100, Como, Italy
sandro.morasca@uninsubria.it

\textsuperscript{3} CEFRIEL, Via Fucini 2, I-20133, 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.

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. It is unlikely that a single notation may describe all of these aspects adequately, so several notations have been proposed in the past. In the last few years, UML [1], which has achieved a wide popularity thanks to its graphical, easy-to-use set of integrated notations and extensive tool support, 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 [11]), but their introduction is still tentative, incomplete, and not well integrated with the other aspects of UML. The introduction of time features must not be carried out only in some parts of UML (e.g., the operational diagrams), because the modeling of time aspects has a profound impact on the meaning of all parts of a UML system representation. 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 have the following characteristics:
• High rigor of syntactic and especially semantic definition: this will make the model of a time-critical system an effective, unambiguous communication means across different actors of the development process, e.g., requirements analysts, designers, etc.
• High integration and consistency with the rest of the UML notation: time cannot be introduced as an add-on to the existing notation, because it deeply influences the semantics of a model and many aspects of a UML model may be affected by time.
• High level of abstraction: UML was introduced with the goal of being a common set of notations for a number of software development activities, starting from requirements analysis, so it 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 introduce timing aspects in UML that addresses 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 tree-shaped hierarchy of parts and subparts, where the root corresponds to the entire system being modeled, and the leaves 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, 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, so there is no redundancy between the information provided by the OTL formula and the statechart.

Our proposal is based on the existing, consolidated versions of UML and OCL. At any rate, we do not foresee any major problems in using our extensions with the new versions of UML and OCL [9,10,11], as our extensions are based on simple and general concepts and appear to comply with the directions of the draft proposals.

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

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 the one proposed in [9] 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. As an example, the construct inv is used to specify an invariant property of a system, and 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. It is not possible to specify the time distance between events, which 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 part of a UML-based formalism, so it is totally integrated with the other UML notations. OTL 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, which is the root of the hierarchy, 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 proper quantitative treatment of time and the definition of sensible operations involving objects of the two classes. For instance, class Time provides (1) an operation ‘≤’ 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.

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 [12]. 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 [9], 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 than the current one, OTL introduces a new semantic primitive ($eval$) as a method of class $Time$, with a notation consistent with OCL. Given an object $t$ of class $Time$ representing a time instant, the evaluation of $p$ at time $t$ is denoted as:

$$t.eval(p)$$

Method $eval$ receives an $OclExpression$ as the parameter ($p$) and returns a boolean value. Its meaning is that predicate $p$ is evaluated at time $t$.

![Figure 1. The OCL standard library extended with types Time and Offset.](image)

All other temporal operators can be defined based on method $eval$. Since $Time$ is introduced as an OTL type, collections of objects of class $Time$ can be defined by means, among others, of our $futrInterval$ and $pastInterval$ operators. 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, which can be defined by means of OCL constructs. As an example, formula

```ocl
context C
inv: Lasts(p, d)
```

specifies that $p$ holds in the interval lasting $d$ time units from the current time. This statement can be defined as a shorthand for the following expression, where the term $now$ denotes the time with reference to which the (sub)formula is interpreted, and
As additional notational conventions, we use $\text{inf}$ to denote the infinite Offset value (i.e., the largest possible value for Offset), and we abbreviate the basic temporal operator $t.\text{eval}(p)$ into the more convenient and intuitive syntax $p@t$, which is similar to the one used in standard OCL [1], where $p@\text{pre}$ denotes the fact that $p$ holds before a method is executed. A number of operators can be likewise defined to refer to the future (e.g., $\text{Futr}, \text{SomF}, \text{AlwF}, \text{WithinF}, \text{Until}$, whose intuitive meaning and formal definitions are in the table below) and the past (e.g., the corresponding operators $\text{Past}(p,d), \text{SomP}(p), \text{AlwP}(p), \text{WithinP}(p,d), \text{and Since}(p,q)$). Even though they do not add expressive power, it is widely recognized that the availability of operators that reference the past allows one to write shorter, more readable, and more intuitive specifications.

<table>
<thead>
<tr>
<th>name of operator</th>
<th>intuitive meaning</th>
<th>formal definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{Futr}(p,d)$</td>
<td>d time units in the future</td>
<td>$p@(\text{now} + d)$</td>
</tr>
<tr>
<td>$\text{SomF}(p)$</td>
<td>sometimes</td>
<td>let $I: \text{Set(Time)} = \text{now.futrInterval(}\text{inf}\text{)}$ in $I-&gt;\exists(t: \text{Time}</td>
</tr>
<tr>
<td>$\text{AlwF}(p)$</td>
<td>always</td>
<td>let $I: \text{Set(Time)} = \text{now.futrInterval(}\text{inf}\text{)}$ in $I-&gt;\forall(t: \text{Time}</td>
</tr>
<tr>
<td>$\text{WithinF}(p,d)$</td>
<td>within d time units</td>
<td>let $I: \text{Set(Time)} = \text{now.futrInterval(d)}$ in $I-&gt;\exists(t: \text{Time}</td>
</tr>
<tr>
<td>$\text{Until}(p,q)$</td>
<td>p holds until q occurs</td>
<td>let $I: \text{Set(Time)} = \text{now.futrInterval(d)}$ in $I-&gt;\exists(t: \text{Time}</td>
</tr>
</tbody>
</table>

As a shorthand, we use the constructs $\text{futr}(v,d)$ and $\text{past}(v,d)$ (where $v$ is any term, and $d$ is a term of class Offset) as terms to denote the value of a term $v$ at a distance of $d$ time units in the future or in the past from now. For instance, $\text{futr}(v,d)$ is defined as $v@(\text{now}+d)$.

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 the formula $\text{Lasts_ie}(p,d)$ states that property $p$ holds from now (included) to now+d (excluded).

3 A Case study

We illustrate our dual language approach with a small fragment of the specification of a digital energy and power meter, derived from the actual specification of the device developed for the Italian Energy Board [14] in the TRIO object oriented temporal logic language [5]. 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 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).
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 $\delta$, to permit it to reach a stable state. The cell activation lasts only $\delta_1$ time units and is repeated after $\delta_2$ time units. At the instant of sampling the photocell may detect either a transparent or opaque portion of the disk. The consumption of an energy quantum is detected when the disk moves from a transparent portion to an opaque one, or vice versa.

The energy meter includes, besides the G_Ferraris and the Disk, a device, called Reader, which 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.
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 $\gamma$ 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 successive 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 the definitions of protocols, which can be easily inferred by the reader.
The G_Ferraris rotates the disk so that the angle of the disk is always proportional to the energy used and may also generate spurious vibrations that affect the disk. However, their amplitude of such vibrations 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 the maximum amplitude of vibrations, and k 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 the following attributes:

- 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.

The definition of gamma can be written as follows (in plain OCL):

```ocl
class Disk
  invariant: gamma = 3.14159/self.n
```

The operations associated with the disk are:

• `alfa()` computes the sum of the vibrations, noise and the normalized angular position `Phi`, i.e., what is observed by the photocells.

• `pos_F()` is a boolean function that is true if the disk (i.e., `Phi`) is in a position that will be read by the photocell as full and false otherwise.

Operation `alfa()` can be formalized as follows:

```ocl
class Disk
  operation alfa():Real
    pre: True
```

Operation `pos_F()` is formalized by the following OCL statement, where `mod` is the modulo operation:

```ocl
class Disk
  operation 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)
```

The behavior of the photocells is specified by the simple statechart in Figure 6.

![Figure 6. Statechart of class photocell.](image)

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

```ocl
class Photocell
  operation 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;

• to detect transitions from a transparent sector to an opaque sector and vice versa.

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`, `empty1`, `full2`, and `empty2` be predicates denoting the detection of opaque or transparent regions on either photocell.

```ocl
class Reader
  def full1: Boolean = Pos1Port.Full()
  def full2: Boolean = Pos2Port.Full()
  def empty1: Boolean = not Pos1Port.Full()
  def empty2: Boolean = not Pos2Port.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(full1 and empty2, risingEdge1)
def confirmedFallingEdge: Boolean = fallingEdge2 and Since(empty1 and full2, fallingEdge1)
```

Finally, the detection of an energy quantum occurs if and only if a “confirmed edge”—whether rising or falling—is detected and results in sending a “token” message to `CostAssign` through the `TokenPort`. This is specified in OTL as follows:

```
inv: TokenPort^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).

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
```

![Figure 7. Statechart of class Reader.](image-url)
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 reported in Figure 8.

\[
\text{TokenPort.token() / } nq = nq + 1 \quad \text{^TotPort.add(DatePort.current_tariff())}
\]

Figure 8. Statechart of class CostAssign.

Every time the CostAssign unit receives a token, it increments variable \( nq \), which represents the total number of tokens 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 (represented by the attribute EnergyUsedReported) is monotonically nondecreasing. The following OTL statement describes this requirement via the usual definition of monotonicity, i.e., the current value of EnergyUsedReported is greater than or equal to its value in any of the time instants in an arbitrary interval preceding the current time. \( D \) (the length of the interval) is a constant value defined in the context of class CostAssign:

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

- The cost of the energy consumed at constant tariff increases proportionally to the consumed energy and the tariff. Given an arbitrary time interval of fixed length \( IL \), and a possible value of the applicable tariff (from a minimum predefined constant value \( \text{minTariff} \) to a maximum \( \text{maxTariff} \)) 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 the time interval:

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

\( IL \) is a constant defined in class Totalizer.

- 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 the energy corresponding to a quantum, say quantumEnergy:

\[
\text{context CostAssign}
\]
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, and the one associated with Reader, can be used in the analysis and verification of the modeled 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. [16] 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. [17], 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 [18] 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 making any explicit reference to elements of the operational model.

5 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.

Our proposal is not only a notation, unrelated to the development process; on the contrary, it 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

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