A Proposed Extension to UML: A Hierarchical Architecture of Temporal-Assertion-Components

A. Teitelbaum[1], R. Gallant[1], H.G. Mendelbaum[1,2], G. Vidal-Naquet[3]
[1]Jerusalem College of Technology - POB 16031 - Jerusalem 91160, Israel
[2]Université Paris V (René Descartes), IUT- 143, av. de Versailles, Paris 75016, France
[3]Ecole Supérieure d'Electricité, rue Joliot Curie, Plateau du Moulon, 91192 Gif-s/Yvette Cedex, France
E-mails : a_hay@jct.ac.il, rgallant@alum.mit.edu, mendel@jct.ac.il, vidalnaq@supelec.fr

Abstract

This paper proposes a double extension to the UML 2.0 new notation, for Real-Time Applications, using the Temporal-Assertion Components of the Arts'Codes method (Applicative Real-Time Systems based on Component Design) [9] , by adding :

- ✓ *Real-Time Components for an architectural design,*
- ✓ *and Temporal-assertions (temporal extended-OCL [2-5, 25]) for an a priori validation of the design and for the verification at the run-time.*

The Arts'Codes components describe the expected properties at run time (Assertion-Guards) and at the completion of an application (Goals to fulfill). They are executed in parallel and can communicate.

These Arts'Codes Components are proposed as synchronous parallel subsystems [1,9,10], which are described in the form of a hierarchical homothetic diagram. They can be viewed as an extension of the UML active objects, by:

- *adding Temporal-assertions that define the Goals of the component (a priori rules defining the properties it has to fulfill at the end of its work),*
- *adding Temporal-assertions that define the Assertion-Guards (Exception rules defining the verifications to test during its work, and what to do in case of ill-functioning),*
- *adding interface mechanisms which allow communication with other parallel components,*
- *using the already existing OO UML data-attributes, function-methods and a 'run' manager-function (active-behavior method) .*

The hierarchy of components facilitates the validation of the whole system : each component has its own assertion-properties to fulfill (goals and run-time-guards), and the property of the main component (the applicative system) is a composition of the properties of all the subcomponents (so allowing the validation of the whole designed system).

This extension is described using a case study[24].

***Keywords**: Component design, Temporal Logic, Real-Time Systems, UML extensions.*

I. Introduction

Many graphic notations for software development have been proposed. Each one represents another view (another approach of the description of a system). The following list is by no means exhaustive, but is indicative of the wide variety of notations in use: Petri-nets [14] for the control view; statecharts, an economical finite state machine representation [15] for the dynamic behavior view, DARTS [12] for the viewing of static data flow between tasks, object-oriented diagrams [16] such as in ROOM [13] to show the architecture of a system as linked parallel objects, etc… The Unified Modeling Language (UML), [11,13] adopts a pluralistic attitude toward the multiplicity of notations, in two respects:

1. Several diagrams and notations are incorporated within UML, addressing various aspects of system development. UML is not a method, and thus does not address the way these diagrams are to be used. On the contrary, there are several diagrams, each one with its own distinct syntax and semantics, which can be used interchangeably to convey different views of the same information (e.g., Statecharts and Activity Diagrams, Sequence Diagrams and Collaboration Diagrams).
2. UML provides several extension mechanisms, which, in effect, enable authors of non-UML notations to use UML as a meta-language to describe non-UML notations.

UML's versatility, notwithstanding, and, in part, because of this versatility, UML has two fundamental limitations.

1. As stated above, UML is not a method, and thus does not address or constrain the way the various diagrams are to be used. This freedom from constraint is intentional, as it widens the user base and frees the user to select a process appropriate to business, technical and cultural circumstances. On the other hand, this freedom from constraint, leaves the user as to whether a given model meets critical requirements. We

will elaborate on this point in our survey of existing approaches to the modeling of components (see section II, below).

2. The authors of UML emphasize the orthogonality of different aspects (e.g., static versus dynamic views). Thus, UML intentionally adopted different diagrams to express orthogonal aspects, in keeping with Parnas' "separation of concerns" doctrine. Philosophically this makes sense, but operationally, this results in a formidable cognitive overhead of drawing and navigating among the various diagrams. Undoubtedly, frequent context switching among different views and notations poses a cognitive burden that discourages model-based engineering: the official taxonomy of UML 2.0 references not less that 13 kinds of diagram and notations (see figure A-5 p.546 of [19]):
   - for the structures: class diagram, component diagram, object diagram, composite structure diagram, deployment diagram and package diagram, and activity diagram,
   - for the behaviors: use case diagram, state machine diagram, sequence diagram, collaboration diagram, interaction overview diagram and timing diagram.

Arts'Codes proposes a definition of components, and a methodology [9], contributing to specification, realization and validation, endeavored to capture the structural and behavioral design of R-T systems in a unique kind of diagram with minimal notation.

Arts'Codes is not unique in the endeavor to reduce the cognitive dissonance associated with multi-notation models. For instance, Dori's Object Process Methodology (OPM) [17] as its name suggests, proposes a single diagram organized around process hierarchy. Arts'Codes, on the other hand, organizes the application around component hierarchies. For distributed or embedded real-time applications, we believe that components provide more natural and effective view of the system architecture, than processes. Furthermore, the architecture we propose supports a robust and verifiable implementation.

So, in this paper, we propose an extension of UML in order to include the concept of hierarchical R-T components.

II. The concept of R-T Components

Various definitions of the term Component have been proposed. The degree of rigor in each definition is indicative of the prominence of the concept Component in the lexicon of the authoring body.

II.1 The UML component definition

UML 2.0 [19 section 4] defines the term *component (* or more precisely the *Basic Component)* as follows:

"A modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment. A component defines its behavior in terms of provided and required interfaces. As such, a component serves as a type, whose conformance is defined by these provided and required interfaces (encompassing both their static as well as dynamic semantics)."

II.1.1 The UML component Structure definition

Regarding conceptual representation of the external interfaces and internal structure of a Component UML 2.0 leverages "the general improvements in CompositeStructures (around Parts, Ports and Connectors)." [19, section 8.3.1, p. 143].

Regarding graphical representation of the external interfaces and internal structure of a Component UML 2.0 exemplifies various representations within the Structure Diagram, each example appropriate to a different level of detail (19, figures provide various options within the Structure Diagram notation [19, section 8.3.1, figures 85-88, pp. 140-141].

Since UML 2.0 uses the classifier box for representation of both *Classes* and *Components,* the stereotype mechanism must be used to denote a particular box as a *component.* The "component" stereotype (either iconic or textual or both) performs this denotation. The "subsystem" stereotype denotes large-scale components. In addition to stereotypes, there are, of course, features that distinguish Components from other Classifiers.

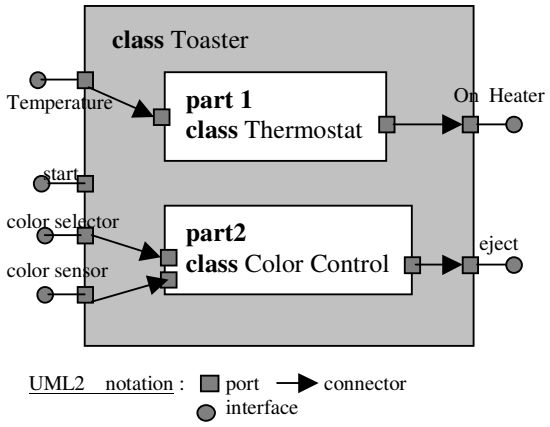
For features such as *parts*, *ports*, *connectors*, *required interfaces*, *provided interfaces*, using *realizations* and *implementing artifacts* UML 2.0 recommends the use of additional, stereotyped compartments in the *classifier box*.

Parts of components have a very general definition; it can be every kind of instances contained in the component: internal classes, subcomponents etc. [19, section 8.3.1, p. 136].

Let us give an example of UML 2.0 component structure (based on [21] and [26] examples), in figure 0 we show a class Toaster which contains two parts: class

Thermostat and class Color control, connected through ports to various interfaces: temperature sensor, start button, color selector, color sensor, Heater and ejector.

figure 0: a main toaster class:



In this diagram, we see the structure of the system, but the behavior of the various classes is not described.

II.1.2 The UML component behavior definition

UML 2.0 defines component behavior in two respects:

1. Any realization of a component must conform to its behavior specification "in terms of provided and required interfaces. As such, a component serves as a type, whose conformance is defined by these provided and required interfaces (encompassing both their static as well as dynamic semantics).

2. With respect to entities external to the component, there is a formal behavioral contract: "of the services that it provides to its clients and those that it requires from other components or services in the system in terms of its provided and required interfaces." [*ibid.,* p. 137]

UML 2.0 provides ample means to specify component behavior, either implicitly or explicitly. Behavior may be specified implicitly "by means of its publicly visible properties and operations." For more precise and explicit definition "a behavior such as a protocol state machine may be attached to an interface, port and to the component itself…. behaviors may also be associated with interfaces or connectors to define the 'contract' between participants in a collaboration e.g. in terms of use case, activity or interaction specifications." [*ibid.*, p. 138].

What is missing in UML 2.0 is an explicit link between architecture and behavior. It is all well and good to view behavior at the various levels as a series of contracts between collaborating elements. But there is no requirement, or guidance in UML 2.0 regarding concordance between architecture and behavior goals. The justification for this is that UML is just a notation standard and not a development methodology. Nevertheless, it is legitimate to ask (and to answer) what notation is required to support construction and verification of architectures that comply with behavioral goals. The developer of reliable components needs an answer to this question.

In addition to the technical issue, there is the aforementioned cognitive issue. For any system of non-trivial complexity, it is difficult, if not impossible to to perform a complete mental integration the static view, as expressed in structure diagrams with the dynamic view, that may be expressed in statecharts, activity diagrams and/or interaction diagrams.

It is precisely these two issues that Arts'Codes is intended to fill.

II.2 The Interface "Façade" definition : Prior to the promotion of the component to a first-class entity, the object-oriented community sufficed with a definition of a component as "an object + an interface"**.** Object-oriented practitioners have used the "façade design pattern" [20], to distinguish between the external interface and the underlying implementation. Component users interact with the public operations of a façade class, which delegates the implementation of these operations to hidden constituent classes of the component. This pattern facilitates implementation of component-oriented applications in classical object-oriented languages.

II.3 The ITEA's Interface definition : The group "Information Technology for European Advancement" [21] has given a much more elaborate definition of components. In addition to the "syntactic interface level", it defines a "semantic interface level" as well as a "synchronization interface specification level." These latter two characteristics clearly encompass behavioral aspects. However, the ITEA's proposed use of UML to represent components is restricted to structural aspects. Accordingly, when using the UML Class notation to represent component *blueprints*, ITEA specifies two specific list compartments: provided interfaces and required interfaces. Similarly, for component *instances*, ITEA uses the UML component notation, in conjunction with the UML interface notation, modified to distinguish between provided and required interfaces.

II.4 The Meyers' Behavioral-contract definition :

In keeping with the capabilities and emphasis of Eiffel, Bertrand Meyers [22] stresses the behavioral aspects of component definition as a contract definition. Such concepts as pre and post conditions are directly represented in Eiffel, and Meyers demands no less of a rigorous definition of components behavior.

II.5 Arts'codes summary-definition:

Arts'codes is concerned with the Real-Time Components for embedded systems. Consequently, this definition is based on the concepts of the active-Object Oriented Programming i.e.: parallel execution of entities which encapsulate a data structure (attributes) and functions (operations) specific to the processing of these data.

However the Arts'codes definition of R-T Component extends the basic definition of active objects as follows:

a/ R-T Component oriented programming can fit the active-Objects to the operation of a physical device or a subsystem, thus allowing design methods oriented towards device/subsystem architectures and not only towards abstract-data-type management;

b/ a R-T system can be composed hierarchically (homothetic view) in components containing subcomponents, and/or can be composed in a network of components and allow communication between them;

c/ R-T components can add some specific interfaces and mechanisms to communicate with other parallel components;

d/ R-T components can have an explicit description of their behavior (implemented in what we call "manager") which coordinates the functions it has to perform depending on conditions, states, events and communication with other components; and subcomponent activation.

e/ R-T components can add some features to the active-Objects, such as Assertions defining *a priori* the Goals that the Component has to fulfill, or verifying on-line the good-working during the execution. These assertions are related to the behavioral functioning of the components-managers. The hierarchy of components allows the validation of the system : each component has its own properties to fulfill (goals and run-time-guards), and the property of the main component (the applicative system) is a composition of the properties of all the subcomponents (so facilitating the validation of the whole designed system).

III Presentation of the Arts'Codes' Temporal-Assertion Components

Arts'Codes follows Meyers [22] in the prominence given to dynamic behavioral aspects of components, and provides specific notation to support this emphasis. But Arts'Codes also proposes a static architectural view of the composition of components (hierarchically or in network).

The remainder of this paper describes Arts'Codes and suggests how it can be integrated in UML : by expressing its notation using UML extension-mechanism, by adding to UML a new kind of component--diagram, and by expressing the coherence between its component--diagram with the other UML diagrams.

III.1 Component Design :

In order to describe an ArtsCodes' component, the engineer will first specify different aspects of each embedded component of the system to be built (software and/or hardware) using 6 types of specification :

```
COMPONENT   X
{     ATTRIBUTES : ...
      SUBCOMPONENTS: ...
      METHODS :
      GOALS : ...
      MANAGER : ...
      GUARDS : ...
}
```

In ITEA parlance[21], these specifications constitute a Component *blueprint*. In UML, this blueprint can be represented by the Class symbol, with a different class list compartment for each specification type. However, for any complex application, the inclusion of all 6 list compartments would be unwieldy. UML diagrammatic notation permits the hiding of list compartments. Given that Arts'Codes method proposes 6 such compartments, such hiding would be encouraged, in accordance with the purpose of a given diagrammatic view. For example, for a view representing system functionality, only the *METHODS* compartment would be exposed. For a view emphasizing control structure, the *MANAGER* and *GUARDS* compartment would be exposed. For a view related to system validation, the *GOALS* compartment would be exposed.

a) *ATTRIBUTES*: here the signals, variables, clocks, and the interface of the component with other components, are given:

external :{
in\out *:*
****variable-type*** *variable-name;*
****flag*** *flag-name;*
****[flagged] signal*** *signal-name;*
****variable-type*** *pipe-name[size];*
****clock*** *clock-name;*
signal *Guard_Exception;}*
local*: {variable-list }*

b) *SUBCOMPONENTS:*
subcomponents*:*
{ component-type subcomponent-instance*
(external variable-list) } // links definitions

c) *METHODS:* Description of the actions and functions to perform in the component (hardware or software). They are described (in any executable language : C++, Java etc.) as functions activated by the behavioral Manager when some conditions are fulfilled;

d) *GOALS* : This describes the Aims of the component, meaning the properties that the component must fulfill at the end of its work, here specified in our extended OCL to Engineering Temporal notation [4,5,25] (see Appendix) :

if (signal\condition*: name)* ***then***
{ ***property****: *flag-name;* ***signal: ****signal-name;}*

This corresponds in mathematical PTL (propositional Temporal Logic notation [2,3] to
"Condition ⋁ signals ==> Property ⋀ signals" (Condition or signals true) imply that (Property and signals are true).

e) *MANAGER* : This is the component behavioral manager, which describes the ordering of its operations and the activation of different subcomponents; it describes the logical controller properties. They can be specified in our Engineering OCL Temporal notation [4,5,25]: ***if (signal\condition****: name)* ***then***
{ ***action****: *action-name;* ***signal: ****signal-name;}*

This corresponds in mathematical PTL (propositional Temporal Logic notation [2,3]) to
" Condition ⋁ signals ==> Actions ⋀ signals " (Condition or signals true) imply that (Actions and signals become true).
In fact, they can be expressed in any language for execution (C++, Java etc.). But for the validation/verification process, the execution of the manager is supposed to be compatible with the "synchrony hypothesis " [1].

f) *GUARDS* : This is he watch-dog of the Component which ensure the correct-working properties (also specified in our Engineering OCL temporal-extension) that the component must satisfy during all its execution, and the reactions it has to do in case of ill-functioning (Exception).
if (signal\condition*: Guard_Exception name)* ***then***
*{****action****: *repair-action-name;* ***signal:*** *exit-signal;}*

This corresponds in mathematical Temporal Logic [2,3] to
(Exception_Condition ⋁ signal ==> repair_Actions)
⋁ (Exception_Condition ⋁ signals ==> exit_signal)
⋁(Exception_Condition ⋁ signals ==>repair_ Actions ⋀ exit_signal)

NOTE1 *:* for specification purpose and mathematical proving of the consistency of the components[3-5], we propose that the Goals, the Behavioral manager and the exception-Guards can be specified in an engineering OCL temporal-extension which is sufficiently close to the automation engineer vernacular specification.
For instance <> can be replaced by "*later*" or "*henceforth*", [] can be replaced by "*always*", () can be replaced by "*next*", (-) can be replaced by "*before*", etc. "*A Until C"* means that action *A* is performed until the first clock-cycle ("synchrony hypothesis " [1]) when *C* becomes true (which is the usual interpretation of the English word "Until"), etc.

example of engineering Temporal OCL notation:
if(Switch_gatedown **and** beginning **and** (**until** clock(x)=3)) **then** (OKdown **and later** let_in)
or in mathematical Temporal logic
[](Switch_gatedown ⋀ beginning ⋀ (***U*** clock(x)=3))
==> (OKdown ⋀ <>let_in)
(straightforward translation)

NOTE2 *:* for specification and execution purpose [7-9], the Goals, the Behavioral manager and the exception-Guards can be described as parallel automata.
example
/Switch_gatedown/ /clock(x) >= 3/ /state 1 /
→ / let_in/ /state 3 /
which are similar to statecharts specification and can be executed by an automata-based operating system [9], but can also be expressed any executable language (C++, Java etc.).

III.2 Components composition

The Arts'Codes model enables the interleaving of the **static** architectural structure (structural model) and the **dynamic** behavior (behavioral model), and facilitates human cognition by providing clear correspondence between different graphical views, and an homothetic view (with the same notation) at all the levels : components and subcomponents.

III.2.1 Static Architecture

The system to design can be composed into a hierarchy of Arts'Codes components and subcomponents, or into a network of communicating Arts'Codes components (using the interfaces through the external *ATTRIBUTES*) Strict adherence to a hierarchical structure and fixed control mechanisms (which we elaborate in the remainder of this paper), ensures that the system's **structure** does not change often, and hence, it is relatively **stable**.

III.2.2 Dynamic behavior

It allows to draw inside the manager: It describes the **dynamic management rules** of the components. It defines the system reactions for internal and external inputs, by transmission of internal and external outputs. The subcomponents are activated in the states of the hierarchic automaton.

So the system's **behavior** described in the Manager of the component, may have to be **changed** (or adapted) many times.

IV Introducing Temporal-Assertion components Diagram for UML

In UML, from a static architectural point of view, we have the concepts of package and objects, and therefore we have the class diagram to describe them graphically.

We propose to add the component-design diagram. A package can group a hierarchy of components and each component can contain subcomponents or objects.

In addition, a component has a behavioral manager described as a statechart extended to the control of subcomponents/assertions and entry/exit gates. As is evident from the case study (see figure 2, below), these gates are visually similar to, but semantically different from UML 2.0 entry/exit points [19, section 15.3.8, p. 471]. Entry/exit points define transitions to and from states of stateCharts for a given component. In Arts'Codes, the state machine of a component is encapsulated in its manager, which can induce state transitions in the manager of a subcomponent (or of an upper-component) through the entry/exit gates, which work as control links. So, the state-machines of the various managers are independent and parallel. And the manager's extended statechart describes the own component behavior and the activation of the subcomponents. A key feature of Arts'Codes is the absolute control exercised by the statechart of the Manager over its subcomponents. The manager, does not merely transmit events to subcomponent statecharts. If this were the case, the subcomponent would discard the received event, if at the instant of receipt, it was in a state for which no reaction to that event were defined. In Arts'Codes the manager can always activate or deactivate the subcomponent statechart, and force it into a particular state (using entry/exit gates), regardless of the subcomponent's current state. This insures system reliability, and verifiability. This requires that each subcomponent statechart support the "State Pattern," [23] in which there is a top level state, from which there are transitions to each of the other lower level states, each triggered by a different event . At the last level, the last component manager of the hierarchy is similar to an UML object with its statechart describing its own behavior without activating other components.

IV.1 Example (informal description):

Let us give the example of a toaster control [24] based on the color of the bread-slices and not only on the timing and temperature. The user chooses the bread-slices color that he wants using a button called "color-selector," introduces the bread-slices and pushes the button "start". The toaster stops in three cases: either when the color is attained, or when it does not work correctly (temperature too high) and after a certain time (a maximum time-out condition is raised).

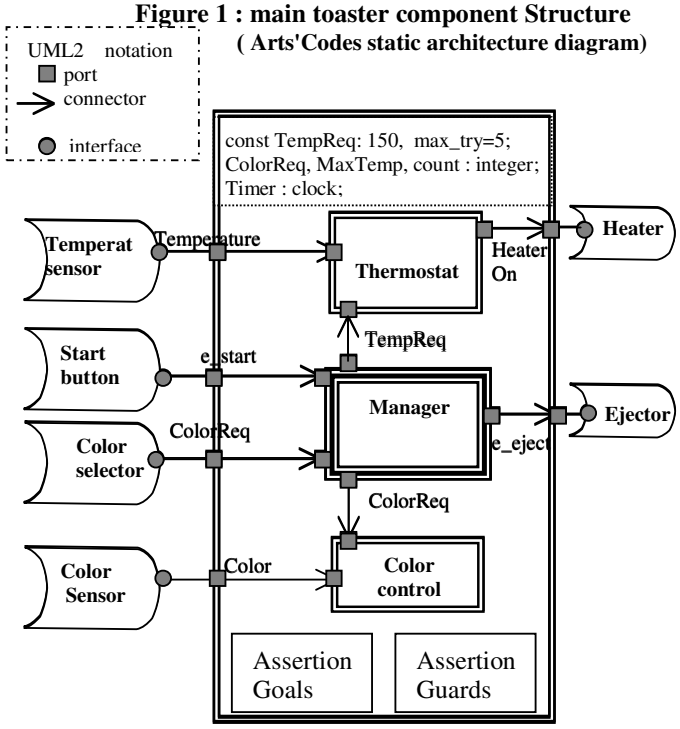
IV.2 Graphic representation

Component structure:

In figure 1, we see the **main package** of the toaster-controller, which contains the main component (double-border box), its manager (triple-border box), and its subcomponents (double-border boxes: thermostat and color-control). We see also the connectors (arrows) between these components and with the I/O virtual devices (half- round-boxes) through data ports and interfaces. By convention, in Arts'Codes, the UML package notation is used to denote the overall system boundary (main component). The half-round-boxes representation of I/O virtual devices is not standard-UML, but is a permitted extension (iconic stereotype). Similarly, the double border of subcomponents, and the triple border for the Manager component are iconic stereotypes. Iconic, rather than textual stereotypes have been chosen to articulate, the Arts'Codes metaphor of

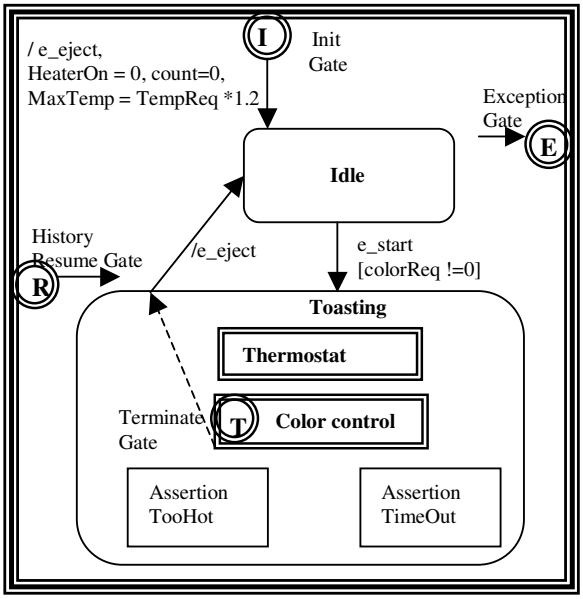
plug-replaceable components, whose socket pins connect to external hardware devices. Insofar as the virtual I/O devices support communication protocols, it serves an implementation of UML 2.0 interfaces. The ports allow support of data exchanges between the various elements (components, managers, virtual I/O devices).
Regarding data and signal flow, Arts'Codes overloads the association name to indicate the connectors (e.g., Temperature labeling the association connecting the Temperature Sensor with the Thermostat). This is consistent with UML 2.0 [19 section 17.2.2] which allows attachment of information items to associations.

Figure 1 : main toaster component Structure
(Arts'Codes static architecture diagram)



Manager's statecharts extensions:
In figure 2, we see the extended statechart of the main-component's manager.
In this extended statechart , the control of the entrance and the exit from a manager are represented by "*Gates*" (double-bordered small circles). For instance in Figure 2, the manager begins by its "*Init I-Gate*", sending the "*e_eject*" event and resetting the "*HeaterOn*" flag, in order to be sure that the toaster begins empty and cold, then it enters the "*Idle state*". When the toaster is started (receiving the "*e-start*" event), it checks if the colorRequired is not 0 then it enters the "*Toasting state*". This statechart is extended to allow the parallel activation of subcomponents (double-bordered rectangles : thermostat and color-control) in the "*Toasting state.*"

Figure 2 : main toaster component Manager
(Arts'Codes dynamical behavior diagram)



Another extension is the presence in the "*Toasting state*" of single-bordered rectangles for the assertions to test during the execution of this state (see "*TooHot* " or "*TimeOut*" at the bottom of Figure 1). Assertions can be tested when entering the state (pre-condition), during the execution of this state (run-condition), or when exiting the state (post-condition). If the assertion becomes true, it means a malfunctioning of the component, so an exception guard must be raised: the context will be saved and the manager will exit by the "*Exception E-Gate*" (see Figure 2), in order to enter the "Guards-chart" (see Figure 8).

Example of Assertion (*if*) and Guard (*then*):
(see Formulae 1) If the Guard detects that everything is OK or if it repaired the problem, then it will come back to the manager execution, coming back through the "*History Resume R-Gate*". If the problem is not repairable, it will come back through the "Initialize I-Gate" to reset the application (here, doing "e_eject" and coming back to the "Idle state").

Formulae 1: Assertion/Guards of "Toaster Manager"

```
"TimeOut Exception ":
        if (time > timeOut)
        then (reset(HeaterOn) and Critical_Gate);
"TooHot Exception ":
    if (Temperature > MaxTemp)
    then (reset(HeaterOn) and incr(count) and
          state=cooling);
    if (state=cooling and count>max_try)
    then (Critical_gate);      //ending the application
    if (state=cooling and Temperature < TempReq)
    then ( Resume_gate);
```

NOTE :
UML 2.0 supports four types of behavioral constraints: DurationConstraint, IntervalConstraint, TimeConstraint and InteractionConstraint, as well as pre and post-conditions. Richer support for real-time constraints and specification may be found in the emerging OMG standard for Schedulability [19].

Finally, if the toaster worked well, the "*Toaster manager*" will finish normally by receiving an acknowledgment from the "*Terminate T-Gate*" from the subcomponent "*Color control*", and it will exit the "*Toasting state*" to go back to the "*Idle state*" while sending the "*e_eject*" event.

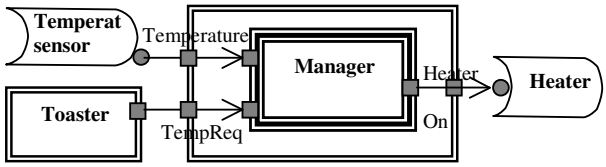
The Toaster-Behavior represented by the Toaster manager statechart (in Figure 2) can be translated easily in our engineering Temporal OCL notation (for validation purpose) :

Formulae 2: Behavior of the "Toaster Manager"

```
"Toaster-Behavior":
if (init_Gate)
    then (state=Idle and e_eject and not HeaterOn and
          count=0 and MaxTemp=TempReq*1.2);
if (state=Idle  and  e_start  and colorReq)
    then (state=Toasting and
       Thermostat.Init_Gate and Color_control.Init.Gate);
if (state=Toasting and
          Color_control.Terminate_Gate)
    then (state=Idle and e_eject );
```

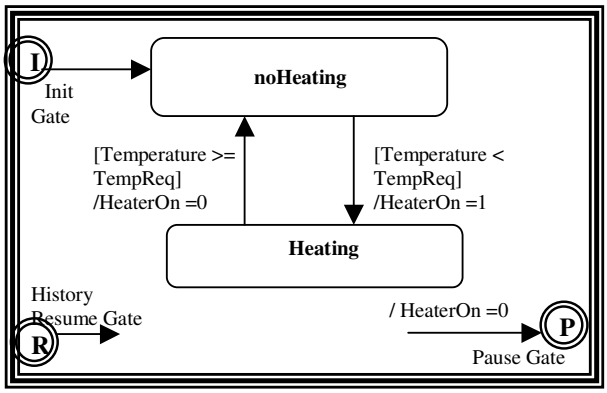
Going down in the hierarchy of the components, we can draw the thermostat subcomponent static architecture (see figure 3a) and its manager (see figure 3b).

Figure 3a : thermostat subcomponent architecture



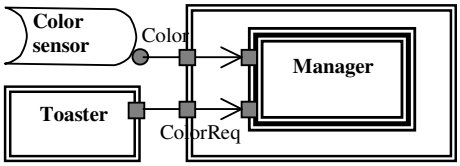
In figure 3a, we see the I/O links of the thermostat component with the "Temperature sensor" and with the "Heater" device, and with the upper level component "Toaster".

figure 3b: the manager of the thermostat



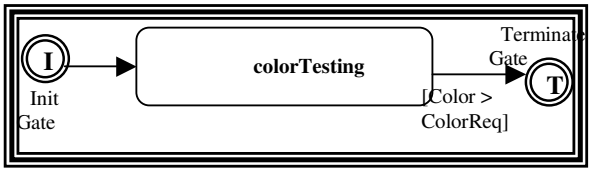
In figure 3b, we see the thermostat manager, which is a classical statechart with no activation of further subcomponents.

Finally while designing the "color control" subcomponent, we shall have the **figure 4a giving its static architecture**:



the figure 4a shows that it receives data from the "Color sensor" device and from the upper level component "Toaster", but does not deliver outside any data, it will only use the dynamic "Terminate" gate to finish its work in the Behavioral description of its manager (see figure 4b and 2).

Figure 4b: the manager of the color control subcomponent



IV.4 Discussion on the Gates extension:

In the Arts'Codes extended statecharts of the managers, we have seen, in the example, an important feature: the gates which allow to make the communication between components, by exchanging data or by sending/receiving signals.

The gates may be viewed as a non-standard adornment of the UML statechart. They are motivated by the Arts'Codes concern with reliability. In the standard use of statecharts, communicating components send events to other components. The sender has way of assuring that the receiver will react to the event in the way the sender intended, or, for that matter, that the receiver will react at all. For reliable control, it augments the controlling abilities of the manager. We must guarantee that manager will, initiate, terminate, or continue the operation of a subcomponent. The appearance of a gate in the statechart of a manager, is, in effect, a constraint on the behavior and design of the statechart of the subcomponent. The subcomponent statechart must provide hooks (in the form of specific transitions and triggering events) that allow the manager to exercise control according to its gates.

V Validation and Verification using Arts'Codes component diagrams

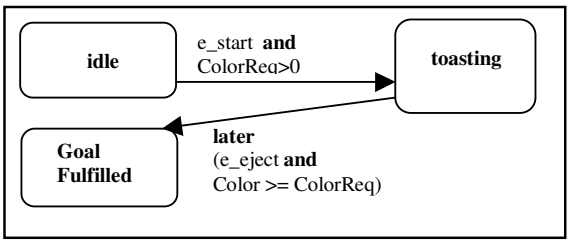
In our proposal, the diagrams can be easily translated into temporal-assertion formulae. The components' assertions for the goals validation, the run-time verifications, the execution behavior and the exception-guards are written in a unified temporal notation, which is a minimal extension of OCL [25] (see our example-formulae 1 to 6, and the Appendix below). And is also sufficiently close to the mathematical Temporal notation, so this enables to have the same notation in the design, the proving and the execution.

Originally, OCL was designed as a constraint specification notation and not as an execution language. Here we used an OCL-Temporal extension for this purpose in the Goals, and the Exception-Guards, but we use it also to describe the component behaviors in order to make the proving by comparing the result of the behaviors' execution to the goals and the guards.

V.1 a priori Validation using Goals-Charts and global-properties

The Goals can be described graphically with the same kind of diagram, the difference with the manager chart is that it can contain Temporal Logic operators (such as "**later**"or **always** or **Until** etc.) instead of actions

Figure 5: the Goals of the toaster



This can be translated in engineering temporal notation:

Formula 3:

"***Toaster-Goals***"
if (e_start **and** ColorReq>0)
then (**later** (e_eject **and** Color >= ColorReq));

In fact the validation of the work of the toaster must be obtained through the validation of the goals of its subcomponents. The Arts'Codes hierarchy of components facilitates the validation of the system : each component has its own goals assertion-properties to fulfill, and the property of the main component (the applicative system) is a composition of the properties of all the subcomponents (so allowing the validation of the whole designed system):

In this example, the validation will be done, if the following global-property is true:

Formula 4: global-property of the toaster

in mathematical temporal notation :

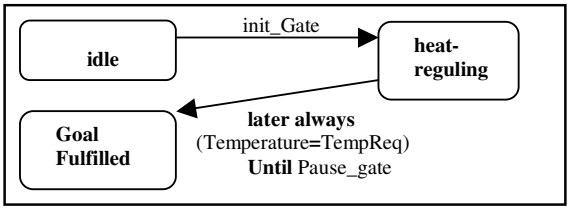
Thermostat-Goals ∧ *Color_control-Goals* ∧ *Toaster-Behavior* ==> *Toaster-Goals*

or, in our OCL extended Temporal notation:

if (Thermostat-Goals ***and*** Color_control-Goals ***and*** Toaster-Behavior) ***then*** (Toaster-Goals)***;***

and so, we need to represent also the goals of the subcomponents:

Figure 6: The Goals of the Thermostat:

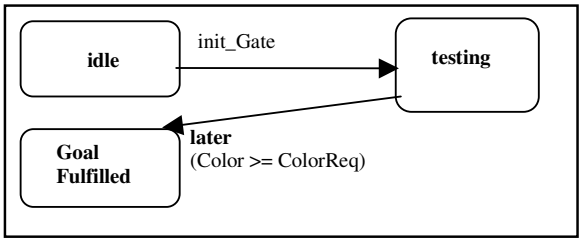


This can be translated in engineering temporal notation:

Formula 5:

"Thermostat-Goals" **:**
if (init_Gate)
then (later always (Temperature**=**TempReq)
Until (Pause_Gate) **);**

in the same way, let us represent **The Goals of the Color control** in **Figure 7:**



This can be translated in our engineering OCL-temporal notation: **Formula 6:**

"Color_control-Goals" **:**
if (init_Gate) **then (later(**Color >= ColorReq) **);**

The global-property of the system (see formula 4) can be proved by combining all the sub-goals and the main behavior. For instance, using an automatic prover such as the STEP prover of Stanford [3,4,5]. But manually and intuitively, we can see that the global-property can be fulfilled (i.e. the system is designed properly) : the main goal wants to obtain e_eject **and** Color >= ColorReq when it starts with a certain color requirement (ColorReq >0);

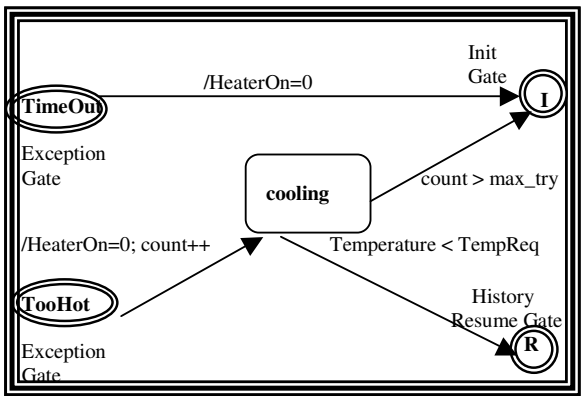
the main Toaster-Behavior activates the Thermostat and the Color_control components when it starts, the goal of the Thermostat-goal shows that it maintains a certain required Temperature when it starts, and the Color_control-goal shows that it terminates when the ColorRequired is reached, finally the main Toaster-Behavior activates e_eject on termination of the Color_control component. So the main Goal is fulfilled.

NOTE: The validation of a component is based on the assumption that its subcomponents have been validated earlier and that their goals are reliable.

V.3 Run-Time Verification using Exception Guards Charts

The exception handling of the toaster (see formulae 1) can be described with the same kind of chart, it is activated by the manager during the execution, and comes back to the manager after handling :

Figure 8: The exception guards chart



When the "Toaster Manager" is in the "Toasting state" (see Figure 2), it will test (Formulae 1), at each cycle of execution, the two assertions "TooHot" and "TimeOut". If one of them becomes true, it will raise an exception i.e : save the context of the manager, and activate the corresponding guard (i.e. it will enter the corresponding Exception gate of Figure 8), and execute the following treatment.

For instance, in case of "TimeOut" Exception, it will close the Heater and go back to the manager through its Init gate which will reset the Manager.

In Case of "TooHot" Exception, it will try to cool the toaster, if there were too many tries (max_try), it will go back to the manager through its Init gate which will reset the Manager. If there were not too many tries, it will wait until the temperature went down to less than TempReq, then it will go back to the manager through the Resume Gate and continue the normal work.

VI comparison with statecharts, Room and OPM

The **statechart** is a visual formalism for complex systems defined by Harel [15], and was specially adapted to reactive systems. Reactive systems are characterized by being *event-driven, continuously having to react to*

external and internal stimuli.

And he adds [15] " *what makes the problem especially acute is the fact that we need tools fitting nicely into human being's frame of mind".* This idea is one of the main goals of this Arts'Codes paper, to fit the graphic program representation to the visual mental one.

His approach for program description is based on State diagrams: *Much of the literature seems to agree that states and events are a priori a rather natural medium for describing the dynamic behavior of a complex system.*
Globally speaking, he added to the known State Diagram two main ideas: Hierarchy and Concurrency: *The two essential ideas enabling this extension are the provision for 'deep' descriptions and the notion of orthogonality.*
The **Statecharts** [15] describe well the dynamic of behaviors, but not the components architecture as in ROOM and in Arts'Codes. For this, we must use the class diagrams of UML to see the objects; and the message sequence diagrams to see the interactions between the objects.
The **ROOM** *(Real-Time Object-Oriented Model)* methodology was developed specifically for dealing with distributed real-time systems based on the object paradigm. It was defined by Bran Selic[13] and it *uses a variation of the basic statechart formalism.*

One of the reasons of this modification was that "*it is generally unrealistic to apply the concept of broadcast communications across a lossy wide-area network".*

This model is defined in the following way:
Behavior specifies the dynamic aspects of a system, while structure deals mainly with architectural issues: how is the system decomposed, what is the relationship between the components, etc. Inheritance is both a reuse and an abstraction facility.

The main structural element is the actor: it *represents an active concurrent activity with a specific responsibility*, and it is *completely hidden from its environment and other actors by an encapsulation shell*.

The **ROOM** method was our principal inspiration for Arts'Codes, but there are certain key differences between the two.

1. ROOM considers the behavior role simply as a regular sub-actor, it doe's not have a special rank in its components. In the Arts'Codes approach the Manager is the Component master, without it no subcomponent activation is possible, and it has access to all components features.
2. The subcomponents activation by the manager's behavioral states in Arts'Codes increases significantly the expressiveness of the subcomponents' parallelism and synchronization, enabling "what you see is what you get (WYSIWYG)" and avoiding undesirable side effects at the execution stage. Moreover, the subcomponents activation is shown by their links to the gates, enabling in such a way more explicit graphical expressiveness.
3. ROOM approaches to the behavior description is like a flat state diagram, with no distinction between normal and exception partition. It does not guide the developer to validate (ab)normal behavior. Arts'Codes adds Assertions to the components, in order to track and to manage with abnormal behavior.
4. The encapsulated ROOM states are not supported in Arts'Codes. Our approach is that encapsulation is made by components, and a state is only the component status, it is not an independent entity. Although Arts'Codes supports state hierarchy, it is only for expression improvement.
5. The interconnections between the ROOM actors are provided by ports, which abstract the protocols. The protocols have to be defined in a Message Sequence Chart separately. In Arts'Codes each variable is interconnected automatically and the detailed connection is provided. This approach meets the standardization of embedded development requirements. The ROOM approach is more suitable for versatile network interconnection. Arts'Codes enables network or controllers interconnections through the Virtual Devices.
6. While ROOM adopts the run-to-completion programming model, Arts'Codes is supported by an execution platform which adopts the Synchrony Hypothesis, alleviating in such a way synchronization and timing efforts.
7. In ROOM the priorities are attached to events, in Arts'Codes the priority is attached to the component and inside the component to the Manager's state-transitions. This approach enables determinism, because an event may enable many transitions and the reaction order must be fixed.
8. Arts'Codes introduces the "socket" concept in order to enable Reusability of components. A component is a type until it is inserted in a socket, creating in this way a new instance of this component type.

OPM [17] proposed by D. Dori gives another solution of the integration of Structural and Behavioral views. He uses the classical UML class diagram to describe object-processes, and allows zooming in a class box of the diagram in order to show the internal statechart. This method resembles to ours for the hierarchical zooming. However, the hierarchy is organized around processes rather than components. As we have already noted, a component-based hierarchy is more compatible with the physical architecture of embedded systems. On the other hand, Arts'Codes tries to reduce the number of diagrams and the complexity of the various notations, we try to have homothetic diagrams in the sense that at all the levels of the hierarchy the diagrams and the notation are the same.

V Conclusion

In fact we have designed all these hierarchical charts (component chart, manager chart, Goal chart, exception guard chart) in an homothetic same way, so that they have the same notation at different levels

We propose that this architecture can be a contribution to UML 2.0 by adding this kind of homothetic diagram which gives an explicit representation of behaviors for components, and a possible validation by composing all the assertions of the subcomponents. Furthermore, these diagrams allow verification at run-time using the exception-guards assertions. All these assertions are written in a unified extension of OCL to Temporal and Boolean logic. The behavior of each component is also described in this extended OCL.

This hierarchical architecture and the automatic translator that insures that "as-built" behavior conforms to requirements and can be verified.

In summary, we stress that :

- ✓ enforcement of a same architectural style, for simplicity, concerning all components, managers and exception guards, with each manager having absolute control over all its immediate subcomponents,
- ✓ and the creation of a supporting tool that translates this architecture into production quality components, and allows a-priori validation and run-time verification,

all this gives a robust and verifiable system.

Bibliography on related works

[1] Gérard Berry and Georges Gonthier "The Esterel Synchronous Programming Language: Design, Semantics, Implementation",. *Science of Computer Programming* vol. 19, n°2, pp 87-152, 1992.

[2] Z. Manna, A. Pnuelli , *The Temporal Logic of Reactive and Concurrent Systems*, Springer Verlag, New-York, 1992

[3] Z. Manna and the STeP group," STeP: Deductive-Algorithmic Verification of Reactive and Real-time Systems", *8th Intern. Conf. on Computer-Aided Verification*, LNCS, vol. 1102, Springer-Verlag, pp. 415-418, July 1996 http://www-step.stanford.edu/

[4] G.Vidal-Naquet and H.G.Mendelbaum "Validation of Temporal-Component based Embedded Systems" *Proceedings of the ECOOP Conf.*, SIVOES Workshop, Budapest, 2001

[5] G.Vidal-Naquet, H.G.Mendelbaum "TCOM : a Temporal Component Oriented Methodology for the industrial Automation Engineer" *Proceedings of the ISPRA Conf.*, Cadiz, Spain, 2002, WSEAS Press M.Fisher "A survey of Concurrent Metatem, the language and its applications", M.Fisher@mmu.ac.uk

[6] Rajeev Alur and David Dill " A Theory of Timed Automata," *Theoretical Computer Science*, 126:183-235, 1994 [4]

[7] H.G. Mendelbaum & R.B. Yehezkael " Using 'Parallel Automaton' as a Single Notation to Specify, Design and Control small Computer Based Systems," *Proceedings of the 8th Annual IEEE International Conf. on the Engineering of Computer Based Systems (ECBS)*, Washington D.C., IEEE April 2001 M.Fisher, S. Kono, and M. Orgun (eds) Journal of Symbolic Computation, Special Issue on Executable Temporal Logics, 22(5), Academic Press, Nov/Dec. 1996

[8] T. Hirst, R.B. Yehezkael, H.G. Mendelbaum,"Some Theoretical Results on Parallel Automata, Conflict, Complexity", *JCT Research Report 2003*, available at http://sukka.jct.ac.il/~rafi

[9] A.Teitelbaum, "A unified methodology for the formal design and execution of Real-Time applications", JCT research Seminar, 5/2/2002 A. Teitelbaum, H.G. Mendelbaum, "SPHAX operating system", *JCT Internal Report*, (2001) Teitelbaum A., Mendelbaum H.G., "Arts'Codes : Generation of Parallel-Automata Real-Time Systems, using a Unifying Diagrammatic Component Oriented Design Methodology," to be published in *Proceedings of the*

IEEE Conference SwSTE03, Tel-Aviv, 2003

[10] F. Boulanger, G. Vidal-Naquet "An object Execution model for reactive modules with C++ implementation", *Proceedings of ECOOP'96*, Linz, July 1996 , Max Mühlhäuser editor, dpunkt.verlag (1997)443-449

[11] Booch G. *Object-Oriented Analysis and Design (OOAD)*, Addison Wesley Pub.(1993)
UML Resource Center, Unified Modeling Language., Standard Software Notation Guide, version1.1,http://www.rational.com/uml/index.jtmpl, 15 Feb. 1999
H.G. Mendelbaum, R. Gallant, J-F. Brette, Ch. Ducateau, "Java-prototyping of hardware/software CBS using a Behavioral OO Model", *Proceedings of the IEEE conference on Computer Based Systems*, Edinburgh, Apr. 2000

[12] Gomaa, H.: "A Software Design Method for Real Time Systems (DARTS), *CACM*, vol.27, n°9 (1984) 938-949
Mendelbaum, H.G., Finkelman, D. "CASDA : Synthesized Graphics Design of Real-Time Systems.", *IEEE Computer Graphics and Applications*, vol.9, n°1 (1989) 40-46

[13] Selic B., Rumbaugh, J. : "Using UML for Modeling Complex Real-Time Systems," http://www.objectime.com/new/uml/index.html, ObjectTime Ltd/Rational Software Corp. White Paper, (March 1998)
Selic B " Protocols and Ports : Reusable Inter-object behaviour patterns", *Proceedings of the IEEE 2nd ISORC Symposium*, St-Malo, France, (May 1999)
Selic B., Gullekson G., Ward P., *Real-time Object-Oriented Modeling (ROOM),* John Wiley Pub. (1994)

[14] Peterson, J.L., "Petri Nets," *Computing Surveys*, vol.9, n°3 (1977), 223-243

[15] Harel, D. and al., "Statemate: a Working Environment for the Development of Complex Reactive Systems," *IEEE Trans. Software Eng.*, vol.16, n°4 (1990) 403-414
Harel D. and E. Gery, "Executable Object Modeling with Statecharts," *IEEE Computer*, (July 1997)31-42

[16] Mendelbaum, H.G., "Introduction to a CAD Object-oriented Method for the Development of Real-time Embedded Systems," *Proceedings of the 1st Israeli-IEEE Conf. On Software Engineering* , Herzlya (1986)

[17] Dori D., "OPM: Object-process methodology," Technion, (1998)
http://brd4.ort.org.il/~zbarzilay/OPM1.ppt

[18] Lavi J.and J. Kudish, *Systems Modeling & Requirements Specifications: The ECSAM Method for Computer-Based Systems Analysis and Modeling*, Dorset House Publ., New-York, 2003

[19] OMG, "UML: Superstructure, version 2.0, Final Adopted Specification", ptc/03-08-02
http://www.omg.org/cgi-bin/apps/doc?ptc/03-08-02.pdf
OMG, "UML Profile for Schedulability, Performance, and Time Specification", ptc/2003-03-02, Draft Available Specification, April 2003.
http://www.omg.org/cgi-bin/apps/doc?ptc/03-03-02.pdf

[20] Gamma E., R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley Longman, 1995

[21] Task 1.4 "Definition of Components and Notation (D.1.4.4)", Software Development Process for Real-Time Embedded Software Systems (DESS),Version 02 - Public, ITEA, Dec. 2001

[22] Bertrand Meyers " Contracts for Components," *Software Development Magazine*, July 2000

[23] B. Douglass, *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns*, Addison Wesley Longman, 1999

[24] R.C. Martin "UML Tutorial: Complex Transitions" Eng. Notebook Column, *C++ Report*, Sept. 98,
http://www.objectmentor.com/resources/articles/cplxtrns.pdf

[25] www.rational.com/media/uml/resources/media/ad970808_UML11_OCL.pdf
www.it.bond.edu.au/inft821/UML-Object Constraint Language/

[26] M. Bjorkander, C. Kobryn "Architecting Systems with UML 2.0", IEEE Software J., (July/Aug. 2003), pp. 57-61

Appendix:

GRAMMAR FOR OCL, EXTENDED TO INCLUDE TEMPORAL ENGINEERING EXPRESSIONS
(based on OCL Specification, v 1.1 31 [25])

The grammar description uses the EBNF syntax, where "|" means a choice , "?" optionality and "" means zero or more times.*
The temporal extensions are shown in **bold** characters (it can be seen, that they are minimal and concern only 4 lines).

```
expression := logicalExpression
ifExpression := "if" expression  "then"
expression   "else" expression "endif"|";"
logicalExpression := relatExpression
(logicTemporalOperator relatExpression)*
relatExpression := additiveExpression
( relatOperator additiveExpression )?
additiveExpression :=
multiplicativeExpression
( addOperator multiplicativeExpression )*
multiplicativeExpression := unaryExpression
( multiplyOperator unaryExpression )*
unaryExpression := ( unaryOperator
postfixExpression )
| postfixExpression
postfixExpression := primaryExpression (
("." | "->") featureCall )*
primaryExpression := literalCollection
| literal
| pathName timeExpression? qualifier?
featureCallParameters?
| "(" expression ")"
| ifExpression
featureCallParameters := "(" ( declarator
)? ( actualParameterList )? ")"
literal := <STRING> | <number> | "#" <name>
enumerationType := "enum" "{" "#" <name> (
"," "#" <name> )* "}"
simpleTypeSpecifier := pathTypeName
| enumerationType
literalCollection := collectionKind "{"
expressionListOrRange? "}"
expressionListOrRange := expression
( ( "," expression )+
| ( ".." expression )
)?
featureCall := pathName timeExpression?
qualifiers?
featureCallParameters?
qualifiers := "[" actualParameterList "]"
declarator := <name> ( "," <name> )*
( ":" simpleTypeSpecifier )? "|"
pathTypeName := <typeName> ( "::"
<typeName> )*
pathName := ( <typeName> | <name> )
( "::" ( <typeName> | <name> ) )*
timeExpression := "@" <name>
actualParameterList := expression ( ","
expression )*
logicTempoOperator := logicalOperator |
tempoOperator | logicalOperator tempoOperator
logicalOperator := "and" | "or" | "xor" |
"implies"
tempoOperator := "Since" | "Until" |
"next"|"previous"|"always"|"later"|"before"
collectionKind := "Set" | "Bag" |
"Sequence" | "Collection"
relationalOperator := "=" | ">" | "<" |
">=" | "<=" | "<>"
addOperator := "+" | "-"
multiplyOperator := "*" | "/"
unaryOperator := "-" | "not"
typeName := "A"-"Z" ( "a"-"z" | "0"-"9" |
"A"-"Z" | "_")*
name := "a"-"z" ( "a"-"z" | "0"-"9" | "A"-
"Z" | "_")*
number := "0"-"9" ("0"-"9")*
string := "'" ( (~["'","\\","\n","\r"])
| ("\\"
( ["n","t","b","r","f","\\","'","\""]
| ["0"-"7"] ( ["0"-"7"] )?
| ["0"-"3"] ["0"-"7"] ["0"-"7"]
)
)
)*
"'"
```