EFFICIENT SYSTEM MODELING OF COMPLEX REAL-TIME INDUSTRIAL NETWORKS USING THE *ACCORD/UML* METHODOLOGY

Sébastien Gérard[1], Nikos S. Voros[2], Christos Koulamas[2], François Terrier[1]

[1]LETI-DEIN - CEA/Saclay
F-91191 Gif sur Yvette Cedex France
Phone: +33 1 69 08 62 59
Fax: +33 1 69 08 83 95
sebastien.gerard@cea.fr

[2]*INTRACOM S.A.*
254 Panepistimiou Str., Patras 264 43, Greece,
Tel.: (+30 61) 465107, Fax: (+30 61) 465070
voni@intracom.gr

Engineers are more and more often faced to the hard problem of developing more sophisticated real-time systems in a world where time to market constraints are constantly increasing. Object oriented modelling with UML *brings significant answers to these issues. However, the real-time behavior specification of an application is not yet completely satisfying. Available methods provide a good support for parallelism modelling of an application but are often poor to express real-time features like deadlines, periods and priorities. In this paper, we present a specific* UML *use supporting qualitative (e.g. multitasking, data sharing, etc) and quantitative (e.g. deadline, period, etc) aspects of real-time behavioral specification. For that purpose, we introduce the concept of the* UML *active object and present a structured way to use* UML *Statecharts in order to describe the behavior of an active object without losing any object properties.*
Keywords *: Real-time, UML, Embedded systems, Active Object.*

1. Introduction

During the last years *UML* has become the lingua franca among system modelers all over the world. Although its presence in the software domain has been successful, *UML* still lacks significant semantics that will allow its dominance in specific domains, like the one of real-time systems; the description of the real-time behavior of such systems is not completely satisfying yet. Available methods (like *UML/SDL* [1], *UML/RT* [2], *RT/UML* [3] or *OCTOPUS* [4]) provide good support for modeling

---

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35409-5_23

B. Kleinjohann (ed.), *Architecture and Design of Distributed Embedded Systems*

the parallelism in an application, but are often poor to express quantitative real-time features such as deadlines, periods, priorities etc.

The methodology introduced in this paper is based on the *ACCORD/UML* approach ([5] [6] [7]) and its main contribution is on the behavioral specification of objects and on the integration of real-time properties to the *UML* models of the system under development. Realtimeness within *UML* models is added from the early specification stages and is transferred consistently down to the final implementation stages where C/C++ code is produced.

The rest of the paper is organized in two parts. In the first one we present a structural way to use UML Statecharts; in the second one the effectiveness of the proposed methodology is depicted through the design a real world application borrowed from the domain of real-time industrial networks.

2. A specific use of *UML* Statechart for real-time system design

In *UML*, a Statechart owns a context that may be either a classifier (like, classes or use cases) or a behavioral feature (like operations or methods). Within the *ACCORD/UML* approach, Statecharts are only used at two levels of granularity to design the behavior of an application:

- *Class behavior* is described through a restrictive and specialized use of UML Statechart largely based on protocol Statechart as defined in UML semantics;
- *Operations behavior* is described via an alternate view of Statecharts. We introduce this view not to define our proper action language but simply because of the lack of action language in *UML* (at this time).

2.1. Class behavior model : protocol and triggering Statecharts

Within the *ACCORD/UML* approach, the Statechart attached to a class aims at modeling its behavior and can be reckoned under two points of view (abstraction): *protocol view* and *triggering view*. In *UML*, objects of an application communicate through message passing that is the result either of a signal raising or of an operation invocation.

2.1.1 Protocol view of a class' behavior

The Statechart describing the protocol view of a class intends to describe all the possible behaviors of a class' instance when it receives a message in the form of a called operation. This specific view allows the designer to specify which operation calls are possible for each state of the object. Indeed, the protocol view describes "what a class can do". The Statechart describing the protocol view of an object refers only to *protocol-transitions* (transitions between the states of the Statechart describing the protocol view) as illustrated in Figure 2. The behavior specified within a protocol Statechart is available regardless of the object's type: standard or real-time active object.

A transition has usually two distinct parts: the left-side part which specifies the triggering condition of the transition; and the right-side part which describes the actions to execute when the transition is fired. A protocol-transition owns only the left-side part specification, i.e. a trigger event (which has to be typed *CallEvent*) with possibly a guard. The right-side part of the transition, i.e. the effect specification of a transition, is empty. Actually, the action sequence specification is implicit to a protocol-transition specification. When such a transition is fired everything happens as if an internal event was sent triggering the execution of the method implementing the operation associated to the call event which has triggered the transition. This rule implies that each operation defined in the interface of an object has to be attached to one method implementing its behavior.

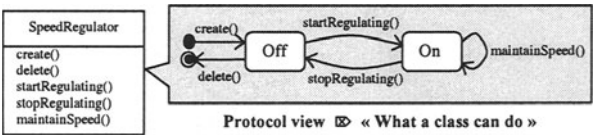

Figure 1 : Protocol view of the global behavior of the *SpeedRegulator* class.

The syntax of a protocol-transition label is then: *event-name '(' comma-separated-parameter-list ')' '[' guard-condition ']'* with the following WFR[1] applied in the context of a protocol-transition *self.trigger.oclIsTypeOf(CallEvent)*.

2.1.2. Triggering view of a class' behavior

Apart from *CallEvent* (which was used in protocol view), *UML* defines additional event types (*SignalEvent*, *ChangeEvent*, *TimeEvent*, *CompletionEvent*) that can also trigger Statechart transitions. In order to manage such events and to specify the additional behavioral requirements specific to active objects, *ACCORD/UML* defines a second view of the class' behavior, the triggering view.

The triggering view is also defined through Statecharts. It defines the reactions of an object in one of the following cases: (a) when the object receives signals that are declared to be sensible, (b) when it reaches a particular state (specification of completion transitions), (c) when it detects a change in the system through a specific boolean expression (*ChangeEvent*) and, (d) when a timer expires. The triggering view of an object focuses on "what a class must do".

The semantics of the triggering view transitions (called *triggering-transitions*) is mainly based on the principle of protocol-transitions i.e. in normal conditions a transition firing involves the execution of the method combined with an operation. As opposed to protocol-transitions, the action sequence of triggering-transitions has to be explicitly specified, and must be a single action (always of type *CallAction*). Moreover, the operation attached to this call action, has to belong to the described object interface. For the left-side part specification of a triggering-transition, all *UML* event types are allowed except for *CallEvent*.

The syntax of a triggering-transition label is the following one:

[1] Well –Formedness Rules.

event-name '(' comma-separated-parameter-list ')' '[' guard-condition ']' / operation-name('comma-separated-parameter-list')
with the following WFR applied in the context of a triggering-transition:

[1] *self.trigger.oclIsTypeOf(SignalEvent)* or *self.trigger.oclIsTypeOf(ChangeEvent)* or *self.trigger.oclIsTypeOf(TimeEvent)* or *self.trigger.oclIsTypeOf (CompletionEvent)*

[2] *Transition currentTransition = self; self.effect.oclIsTypeOf(CallAction)* and *self.stateMachine.context* γ *exists (feature= currentTransition.effect.operation)*

Figure 3 illustrates the modelling of the triggering view of the *SpeedRegulator* class. On the structural specification of the class, we can see that the class is able to receive the *OnOff* signal. The designer may also specify the behavior of class instances after receiving a specific signal type (depending always on their current state). For example, in state *Off* and on receipt of the *OnOff* signal, the *regulator* object executes the method implementing its *startRegulating()* operation.

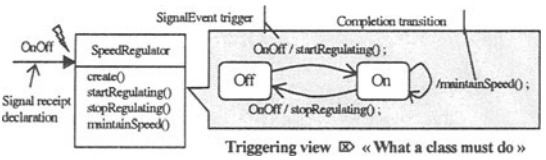

Figure 2: Triggering view of the global behavior of the *SpeedRegulator* class.

The triggering view of an object's behavior is not independent of its protocol view. There are two principal rules to respect during modeling of the triggering view in relation to an existing protocol view:

1. the set of states that an object may have all along its life are fully defined in the specification of its protocol view. Therefore, the triggering view must not have states not defined within the protocol view. This means that a triggering-transition may exist only if it exists a protocol-transition with same source and target states and whose transitions involve the execution of the same operation;

2. a triggering-transition from state S_1 to state S_2 with the label, *evt [g]/opeName('comma-separated-parameter-list')*, is "lawful", if the corresponding protocol view defines a protocol-transition from state S_1 to state S_2 with the label *opeName('comma-separated-parameter-list') [g]*.

The Statechart describing the class' global behavior is constituted of states and protocol-transitions defined in its protocol view, and by triggering-transitions defined in its triggering view. The two views are not two different Statecharts composed one way or another to give the class' global behavior. Actually, they are two abstractions of the class' global behavior, each one focusing on a particular aspect of its behavior. Figure 3 depicts the global behavior of the *SpeedRegulator* class stemming from the previously defined protocol and triggering view definitions.

The main interest of such behavioral specification structuring is to separate the responsibilities of the class' behavior specification. Indeed, the protocol view

defining the "what can do" of a class is independent of its nature: standard or real-time active. While the triggering view defining the "what must do" of a class is only available for real-time active classes. Real-time active classes can also be easily reused, either as real-time active objects or as standard objects. In the latter case, the reactive aspects of the object behavior specified in its triggering view are inhibited.

The behavioral specification described in the previous sections was limited to the description of the control logic of a class. In order to go further in the behavioral description, the user needs to model the algorithmic part of the application under development. Unlike all OO approaches for real-time system development, the *ACCORD/UML* method proposes to avoid mixing the class' control logic with its algorithmic specification. For that purpose, the transition firing of a class' behavior, protocol-transition or triggering-transition, implies the execution of the method implementing the operation specified, implicitly or explicitly, in the label of the fired transition. Actually, there are no other actions specified in the transitions of a class' behavior than a call to one operation of the class interface. All algorithmic descriptions, and therefore all executable code parts, are postponed within the methods implementing class' operations.

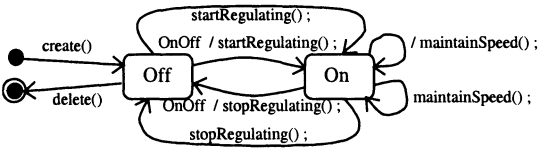

Figure 3: Global behavior of the *SpeedRegulator* class.

2.2. Operation behavior

After having specified the class' control logic, the designer models the behavior of the class' operations. Up to now, in *UML* there are not well-defined ways to specify algorithms. Since Statecharts may be used to specify behavioral features such as methods i.e. implementation specification of class' operations, *ACCORD/UML* elaborates the use of Statecharts for defining the behavioral specification of object's operations. More specifically, methods are described through a specific use of Statechart decomposition[2] in sequences of *UML* elementary actions: *send action*, *call action*, *create action*, *terminate action*, *destroy action*, *return action* and *assignment action*.

The actual goal of this approach is not the description of complex computation algorithms. The designer has to keep in mind, when he or she models the algorithm Statechart of an operation, that the communication between the object and other real-time objects of the application must be specified in order to underline the synchronization points between all real-time objects of the application. A real-time object may communicate with other real-time objects in two ways: either by signal sending or by operation invocation (involving synchronization or not with other objects).

Towards *UML* semantics for communication via signals, *ACCORD/UML* clarifies following things: the communication by signal sending is asynchronous and relies on

[2] The decomposition follows the action sequences as defined in *UML* Statechart package.

a broadcast mechanism. In other words, a signal communication involves one sender which does not know the receivers (classes being willing to receive a signal have declared it in their structural model) and N receivers which do not know the sender. Signal sending will generate to all potential receivers a signal event and the targets will react according to their triggering view specification by a method execution.

The second possible communication type is operation call. This communication mode can be either *asynchronous*, *synchronous* or in *ACCORD/UML delayed synchronous*. The difference between the two last ones is that for a pure synchronous operation call, the caller waits as soon as it has sent the message while with a delayed synchronous operation call, the caller can continue its execution and waits for the return event only when it needs to use the result. For that last specific communication mode, the sender uses a special communication mechanism called *reply box* (*Rbox*). When an object sends a delayed synchronous message to another object, it attaches to its request an instance of the *Rbox* class. The called object will put its answer in this reply box, and the caller can at any time verify if the answer is ready and take the response as soon as it becomes available. This is a special kind of variable needed to receive a result from a called object. The *Rbox* notion is similar to the notion of future in ABCL [8] or continuation box (*Cbox*) in Act++ [9].

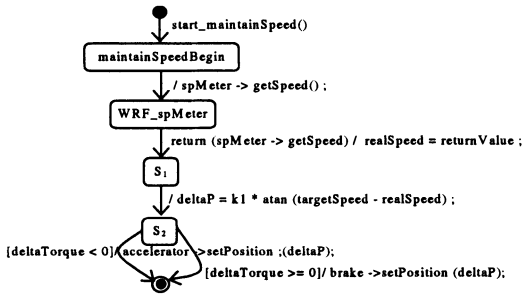

Figure 4: Behavior of the method implementing the *maintainSpeed()* operation.

The Statechart describing an operation algorithm is always starting with an initial state connected to a state named <operation-name>Begin and labeled start_<opeName>'('comma-separated-parameter-list')'. In fact, this label specifies an event, and particularly it is an internal event of the class' behavior. This internal event may be generated under two conditions: either when a protocol-transition labeled <opeName>'('comma-separated-parameter-list')' is fired or when a triggering-transition labeled Sig'('comma-separated-event-parameter-list')' [g]/<opeName>'('comma-separated-parameter-list')" is activated.

For example, when the regulator object reaches the *On* state, it fires automatically the completion transition labeled "/maintainSpeed()" which involves the generation of the internal event *start_maintainSpeed()*. This event is directed implicitly towards the state machine describing the behavior of the method implementing the *maintainSpeed()* operation involving its execution.

2.3. Summary of behavioral specification

To summarize both previous sections, we will consider that the behavior of a class is a state machine owning different AND-states, one for the class' behavior itself (protocol and triggering views that constitute the control logic specification as described in section 2.1) and one per operation behavioral specification (the algorithmic part specification as described in section 2.2). The result of such specification is given for the *SpeedRegulator* class in Figure 5.

If the SpeedRegulator behavior had been modeled following usual OO approaches, it could give something like the Statechart presented in Figure 6. It is evident that the main drawback of such modeling approach consisting of a Statechart specifying all class' behavioral aspects, is that it impairs the object feature of the class. In these conditions for example, it is very difficult to use inheritance features of the object paradigm, because as we can see in Figure 6, control logic as well as algorithmic specifications, that is usually contained inside the class' operation body, are mixed in the same Statechart specification.

Moreover, the implementation of an operation is also often dispatched on different transitions of the same Statechart. It is obvious that inheritance of an operation from a parent class and its behavioral refinement is not an easy task. The structured way of using UML Statecharts in order to specify the class' behavior presented in section 2, allows a UML model to conserve its OO features. For example, if a class inherits an operation, it has only to redefine the AND-state defining its behavior.

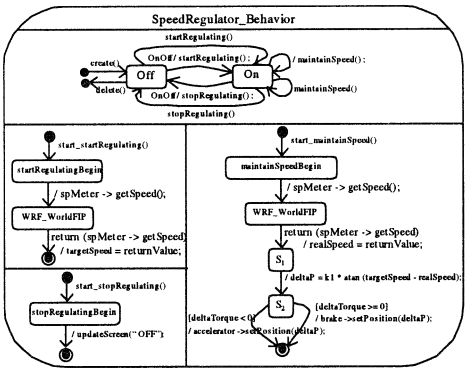

Figure 5: *ACCORD/UML* specification of *SpeedRegulator* behavior.

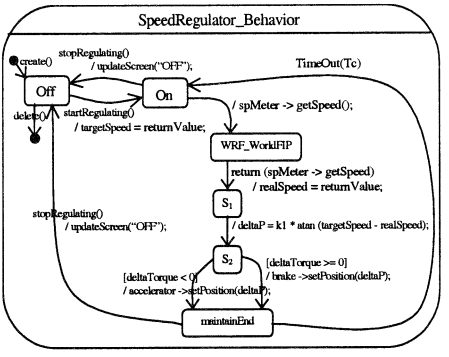

Figure 6: Usual OO specification of *SpeedRegulator* behavior.

Finally, regarding the proposed Statecharts structuring for describing the behavior of a class, it is important to clarify what the RTC[3] assumption of *UML* Statecharts holds in our case too. Actually, this assumption is adapted to the real-time active object paradigm [10] in order to respect the concurrency policy we have adopted within our execution model, that is to say, the '1 writer and the N readers" concurrency policy. This mechanism is not presented in this paper due to its complexity and it is fully described in [11]. The application principle of the RTC

[3] Run To Completion.

assumption on the real-time active object paradigm relies on the fact that the RTC assumption is relaxed in some AND-state of the resulting class' behavior, more particularly the AND-state describing the behavior of operations which are specified as writing operation in the class' structural specification.

3. Case study: *UML* modeling of a real time industrial application

In the rest of the paper we will present how the *ACCORD/UML* methodology can be used for the design of a real world application. The example is borrowed from the domain of industrial networks and its goal is to extend the scenario described in Figure 4 and Figure 5 using fieldbus networks.

From specification point of view, the models describing the application must reflect inherent concepts of fieldbuses like: maximum time between network events, periodic (data arrival at specific periods of time) and episodic (unpredictable but still bounded) traffic behavior, skews and jitter (the variation around the period of the arriving messages) produced during the data updates.

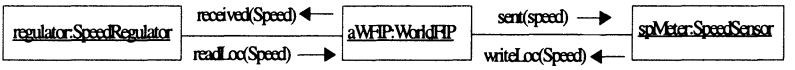

Figure 7 : Distributed case collaboration diagram.

In Figure 7, a single fieldbus connecting two application entities running on different nodes (a producer and a consumer) is presented. We assume that the consumer application entity is *regulator*, an instance of the *SpeedRegulator* class. This object needs at some point of execution the current speed value (that is produced from the *spMeter* object). Assuming that the *spMeter* object (an instance of *SpeedSensor* class) is running on a different node, we have a slightly different model of the operation Statechart presented in Figure 4. In fact we have a distributed implementation of the *maintainSpeed()* operation based on the distribution of the speed values by the fieldbus[4].

Concerning the WorldFIP class behavior, it is reduced just to periodically updates – at precise intervals with a period Tcyc – where the local image of speed variable at regulator node is updated with the equivalent local image of the variable at the node of spMeter. The operation writeLoc() is used from the producer to update its local image of speed variable, while the readLoc() is used from the consumer to retrieve the current local image of the same distributed variable. In each speed value broadcasting, a WorldFIP object sends a sent() message to the producer and a received() message to the consumer of the speed value. Figure 8 and Figure 9 illustrate the new behaviors specification for both SpeedRegulator and SpeedSensor classes with respect to the previous discussion.

[4] In the specific implementation, fieldbus is WorldFIP [12], and the behavior of its periodical services component is based on MPS (manufacturing Periodical/aperiodical Services) services of WorldFIP.

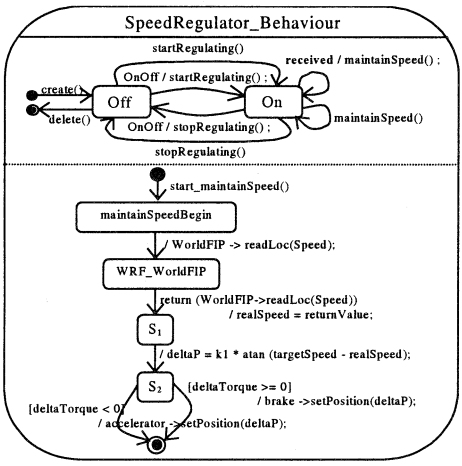

Figure 8: Distributed behavior *SpeedRegulator*.

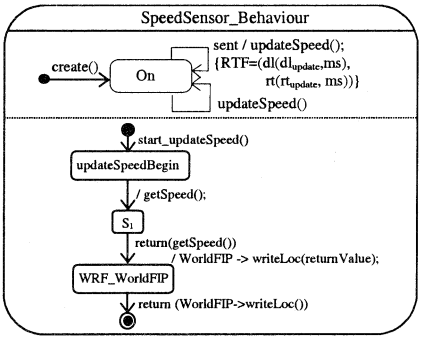

Figure 9: Distributed behavior *SpeedSensor*.

The previous discussion dealt with period and delay timing constraints. But in the described distributed solution there is also a skew introduced, and that is between the time a speed value is captured by the measuring algorithm of *spMeter* and the time that value takes place in the regulation algorithm executing at the *SpeedRegulation* site. In a completely asynchronous operation of a consumer and a producer entity, this skew could be from zero to rt_{update} time and if this skew defines a constraint, then a synchronization mechanism between the producer and consumer Statecharts should be implemented. This synchronization is achieved by the usage of the *sent()* and *received()* messages from the *WorldFIP* object along with a ready time constraint attached to the operation using the produced value in the consuming entity in order to delay the consumption up to the time of circulation. The actual timing and the corresponding constraints are shown in the sequence diagram of Figure 10.

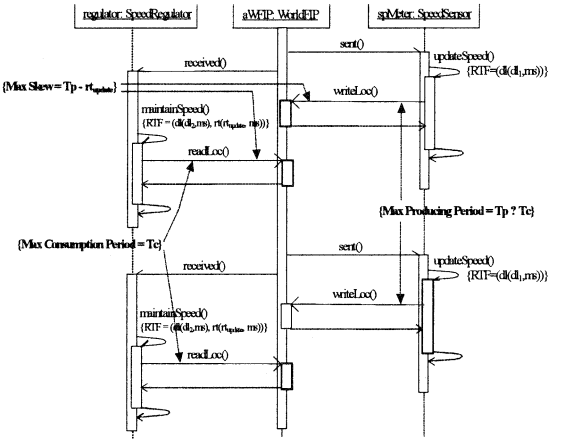

Figure 10: Distributed case sequence diagram.

4. Conclusions

In the previous sections we presented an approach for modeling real time applications with *UML*. The *ACCORD/UML* design approach described does not extends the *UML* semantics in order to fulfill the design needs of real time applications; it proposes a design practice which relies on a specific use of *UML* Statecharts and the concept of *UML* tagged values. As it was exhibited in the case study section, the designer is able to adopt the design framework proposed by *ACCORD/UML* in order to describe real time aspects like periods, skews, jitter and deadlines. In this context, the high-level system models of an application borrowed from the domain of real time industrial networks was illustrated.

The *ACCORD/UML* method is supported by Objecteering 4.30b[5]. The available configuration involves a set of modules allowing the generation of the real-time behavior of an object from its Statechart specification into C++ code. Moreover, a specialized C++ generator has also been developed in order to support the concepts of the extended active object. Finally, with respect to the underlying real time operating system (RTOS), two layers between the application and the RTOS, namely *ACCORD/UML* Kernel and *ACCORD/UML* virtual machine are available too; the first one implements mechanisms supporting active object semantics and above all the mechanism allowing to schedule application tasks respecting an EDF[6] policy; the second one gives the application independence as far as the underlying RTOS is concerned. The latter, exists for Solaris and VxWorks5.2.

Acknowledgment

This work has been performed in the framework of the AIT-WOODDES Project N° IST-1999-10069. AIT-WOODDES project is partly funded by the European Commission. The authors would like to acknowledge the contributions of their colleagues from Peugeot Citroën Automobiles – France, Mecel – Sweden, Commissariat à l'Energie Atomique – France, Verilog France, I-Logix Ltd – Israel, Intracom S.A – Greece, Uppsala University – Sweden and Kuratorium OFFIS e.V. – Germany.

References

[1] P. Leblanc, "Object-Oriented and Real-Time Techniques: Combined Use of OMT, SDL and MSC," in *Current Issues in Electronics Modeling Series*, 96b.

[2] B. Selic, G. Gullekson, and P. T. Ward, *Real time Object-oriented Modeling*: John Wiley & Sons, Inc., 94.

[5] Objecteering is trademark of SoftTeam.
[6] EDF : Earliest Deadline First.

[3] B. P. Douglass, *Real-Time UML : Developing Efficient Objects for Embedded Systems*, 98.

[4] M. Awad, J. Kuusela, and J. Ziegler, *Object-Oriented Technology for Real-Time Systems: A Practical Approach Using OMT and Fusion.* Upper Saddle River, NJ 07458, USA: Prentice Hall, 96.

[5] F. Terrier, A. Lanusse, D. Bras, P. Roux, and P. Vanuxeem, "Concurrent objects for multitasking," *L'objet*, vol. 3, pp. 179-196, 97.

[6] A. Lanusse, S. Gérard, and F. Terrier, "Real-Time Modeling with UML : The ACCORD Approach," presented at "UML98" : Beyond the Notation, Mulhouse, France, 98.

[7] S. Gérard, A. Lanusse, and F. Terrier, "A Train Control Modeling with the Real-Time Object Paradigm," presented at ECOOP Workshop, Bruxelles, 98.

[8] A. Yonezawa, *ABCL: An object-oriented concurrent system*: MIT Press, 90.

[9] D. G. Kafura and K. H. Lee, "ACT++: Building a concurrent C++ with actors," *Journal of Object-Oriented Programing (JOOP)*, pp. 25-37, 90.

[10] F. Terrier, G. Fouquier, D. Bras, L. Rioux, P. Vanuxeem, and A. Lanusse, "A Real Time Object Model," presented at TOOLS Europe'96, Paris, France, 96b.

[11] S. Gérard, "Modélisation UML exécutable pour les systèmes embarqués de l'automobile," in *GLSP*. Paris: Evry, 00.

[12] CENELEC, "EN50170 : General Purpose Field Communication System," CENELEC 1996.