Modeling Industrial Embedded Systems with UML[*]

João M. Fernandes
Dep. Informática
Universidade do Minho
Braga, Portugal
miguel@di.uminho.pt

Ricardo J. Machado
Dep. Sistemas de Informação
Universidade do Minho
Guimarães, Portugal
rmac@dsi.uminho.pt

Henrique D. Santos
Dep. Sistemas de Informação
Universidade do Minho
Guimarães, Portugal
hsantos@dsi.uminho.pt

ABSTRACT

The main purpose of this paper is to present how the Unified Modeling Language (UML) can be used for modeling industrial embedded systems. By using a car radios production line as a running example, the paper demonstrates the modeling process that can be followed during the analysis phase of complex control applications. In order to guarantee the continuity mapping of the models, the authors propose some guidelines to transform the use case diagrams into a single object diagram, which is one of the main diagrams for the next development phases.

1. INTRODUCTION

Blaupunkt Auto-Rádios Portugal (Bosch Group) is a company that produces radios for the automobile industry. Since its managers are continuously searching for ways to improve its production process, a project was established to pursue the following goals:

phase 1: Modeling the control system needed by the material flow on the production lines.

phase 2: Modeling the actual implementation of the control system.

phase 3: Diagnosing the mismatches between the needed system and the actual implementation, and optimizing the later based on the mismatches found.

phase 4: Prototyping the optimized system, using hardware-software co-design techniques and reconfigurable devices, namely Xilinx XC4000 family [1].

The previous experiences of the authors [2; 3; 4; 5] have shown that models based on Petri Nets were sufficient to specify the systems' control view. However, for modeling other aspects of the systems (data and function), it is important to consider genuine multiple-view models. The solution was to consider UML as a unified representation for embedded systems, since it is a standard notation that covers the most relevant aspects of a system.

UML was the notation used to specify both the needed system (phase 1) and the actual implementation (phase 2). With this approach, finding the mismatches between these two systems is easier, since both are specified with the same language. Additionally, the development of the needed system can be followed, since automatic code generation tools (for C and VHDL) are available.

The process model used within the project has the following characteristics: operational approach, refinement and transformation of the specifications, spiral model, reverse engineering, and automatic code generation for prototyping. This paper focus mainly on the steps followed during the 1st phase and shows the results achieved by the operational approach within the analysis phase of industrial systems.

2. THE PRODUCTION LINES

The production lines (designated Hidro lines) are used to manufacture car radios. Each car radio is placed on top of a kit, whose track along the lines are automatically controlled. The transport system is composed of several rolling carpets that conduct the radios to the processing sites.

The radios are processed in pipeline by the Hidro lines. The processing sites are geographically distributed in a sequential way, along the Hidro lines. Each Hidro line is composed of 5 transport tracks: 3 on the upper level ($\mathbf{L}_A$, $\mathbf{L}_B$, $\mathbf{L}_C$) and 2 on the lower level ($\mathbf{L}_D$, $\mathbf{L}_E$). The upper level tracks transport kits from left to right and the lower level tracks transport kits from right to left.

The track $\mathbf{L}_B$ is used mainly to transport radios between non-sequential sites. The upper tracks $\mathbf{L}_A$ and $\mathbf{L}_C$ are preferably utilized for sending the radios to the buffers of the sites (FIFOs that start at the sites). The lower tracks $\mathbf{L}_D$ and $\mathbf{L}_E$ are used for: (1) routing malfunctioning radios to the repairing sites; (2) feedbacking the sites that did not accept radios because their buffers were full; and (3) transporting empty kits to the begin of the line.

There is also a robot that receives radios from the previous production sub-processes (component insertion) and puts them on track $\mathbf{L}_B$. The transfers allow the change of kits between two neighbor tracks at the same level or between a track and an elevator. The 5 elevators establish the linkage between the upper and the lower tracks.

3. THE DESIGN FLOW

UML is a general purpose modeling language for specifying and visualizing the artifacts of software systems, as well as

for business modeling and other non-software systems [6]. UML is a standard language for defining and designing software systems, and is being progressively accepted as a language in industrial environments. UML is meant to be used universally for the modeling of systems, including automatic control applications with both hardware and software components, so it seems a good choice for embedded systems.

For modeling the needed system, the team followed the design flow depicted in fig. 1. It is presented in a sequential way (similar to the waterfall model), but, in practice, it is more iterative and incremental.
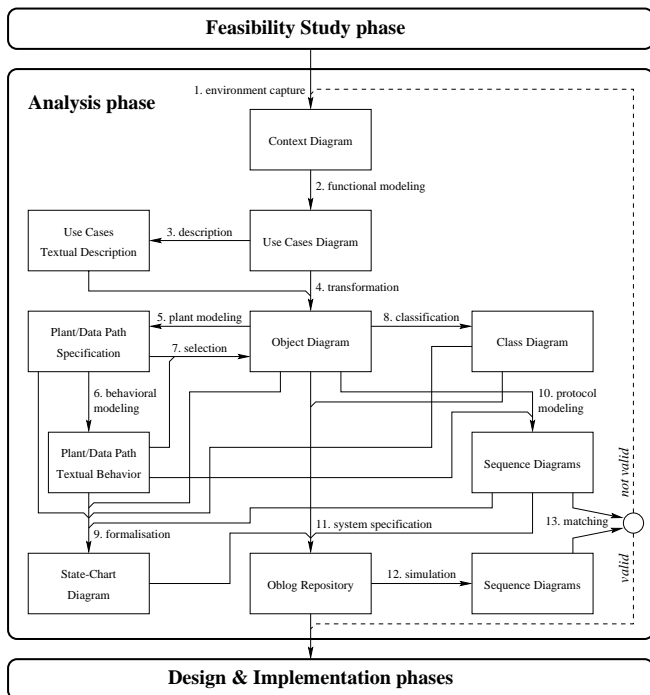

Figure 1: The design flow for the project.

The main views for specifying the system are captured by the following UML diagrams. **Use case diagrams** are used to capture the functional aspects of the system as viewed by its users. **Object diagrams** show the static configuration of the system, and the relations among the objects that constitute the system. **Sequence diagrams** present scenarios of typical interactions among the objects that constitute the system or that interact with it. **Class diagrams** store the information of ready-made components that can be used to build systems and specify the hierarchical relationships among them. **State-chart diagrams** are used to specify the dynamic behavior of some objects/classes.

The information that is represented in state-charts diagrams, object diagrams, and class diagrams is transformed into Oblog, which is a UML-based (extended subset) object-oriented modeling language that allows the system to be simulated and has automatic code generation capabilities [7]. The Oblog environment generates sequence diagrams, as a simulation output, that can be compared with those previously created to specify the system behavior in order to validate the system's requirements (step 13 in fig. 1).

Although the OMG's Real-time Analysis and Design working group has not come yet with a final proposal for directly incorporating real-time concepts into the UML standard (namely in what concerns the syntax for the OCL language), the authors are using UML for dealing with hard real-time systems. Up to now timed sequence diagrams and Oblog syntax have been used for the specification of the canonical latency and duration constraints, which are viewed as composites for more accurate categories of timed requirements (for performance and safety constraints specification).

4. CONTEXT AND USE CASE DIAGRAMS

The context diagram of the system is the first one to be built and it shows which actors interact with the system (fig. 2). This diagram defines the boundaries of the system. The actors are anything that interacts with the system, but do not belong themselves to it. The behavior of the actors need not to be specified to complete the system, but they must be considered to correctly build the system.

The next task was to define the use case diagram, which is a powerful technique for capturing the user's requirements. It is an easy-to-read diagram that divides the system in its functional points. A use case can be understood as a service or functionality that the system offers to its users.
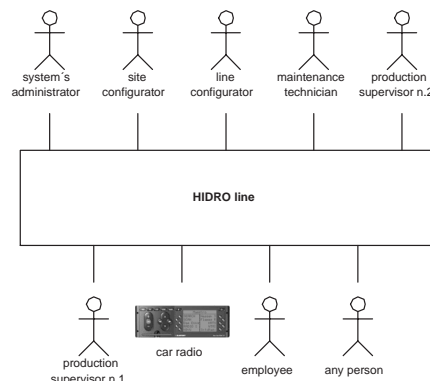

Figure 2: The context diagram.

The authors propose an extension to UML by adding a new tagged value to use cases, that was designated *reference*. Each use case can have a reference that follows a numbering scheme similar to the traditional DFD scheme. Each use case at the top-level is assigned a reference (*e.g.* 9), and if this use case is eventually refined, each one of its sub-use cases has a reference that uses the super-use case as a prefix (*e.g.* 9.2). This numbering scheme can be repeated to any depth, helps to relate all use case diagrams and is used during the transition from use cases to objects to ease the mapping between both models.

Fig. 3 shows the top-level use case diagram, showing which actors perform which functionalities. Since use cases can be further decomposed, the most important use cases (in this case, 9 and 10) were refined in other use case diagrams. This allows more detail to be added to the initial diagram and the project to follow a risk-driven process, where the most important functionalities of the system are first tackled.

After identifying all the system's use cases, the next step is to describe their behavior. There are some forms for doing it (informal text, numbered steps with pre- and post-conditions, pseudo-code and activity diagrams [8]) and, as an example, the descriptions for use cases 9 and 10 are next presented following the first proposal.

9. conduct car radio: Move radios along the tracks in order to make them available for processing at the sites.

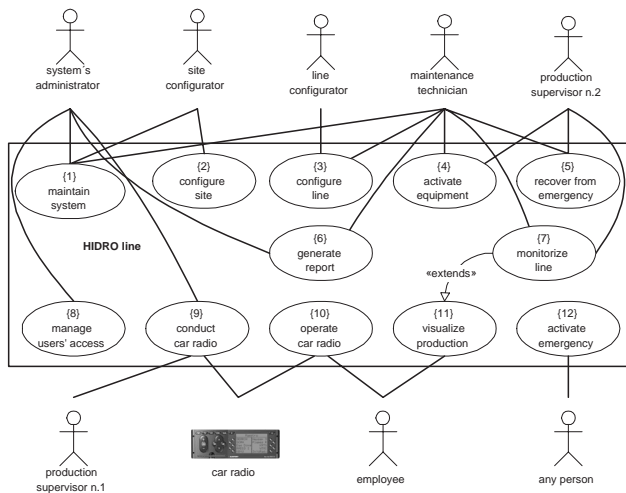**10. operate car radio:** Perform a set of predefined operations to produce radios.

Figure 3: The use case diagram.

5. OBJECT DIAGRAMS

Object diagrams are used to show the components that constitute the system. Transforming the use cases that divide the system in a functional way into objects is a critical task, since usually there is no direct one-to-one mapping from use cases to objects. Several use cases can give rise to one object and one use case can originate a couple of objects.

Douglass presents 11 useful strategies to find/identify the objects of a system [9], but they can not be directly applied to this purpose, because none of them is based on the use cases that were previously identified.

A strategy, composed of some guidelines, is needed to guide the developers on how to transform use cases into objects. There are some approaches for that transformation, but the majority of them is based on personal feelings and some kind of magic. The authors present a new systematic strategy for finding the objects of a system from its use cases, based on the OOSE's object types (interface, data and control) [10]. The transition from the use cases to the objects is a critical project step, since the system's architecture is starting to possess a shape. It is a creative task and some guidelines are given below to help the developers during this task. This transition consists in distributing the behavior specified by the use cases to the objects.

The main point is how to identify the objects that constitute the system. The first solution would be to consider each use case as an object [10], but this does not ease the localization of the modifications that must inevitably be executed during the system's life cycle (either during development or usage). Another alternative, also not considered in this work, is to use data objects to just store information and to totally put the dynamic behavior in control objects. It is better to avoid this solution, since it would be built a structure similar to those that result from applying any structured method, where the separation between data and functions is quite evident. To avoid the well-known problems associated with the structured approach, it is recommended to associate behavior to data objects.

The approach that the authors followed to transform use cases into objects consists of a 4-step rule set:

step 1: Transform each use case in three objects (one interface, one data, and one control). Each object receives the reference of its respective use case appended with a suffix (i, d, c) that indicates the object's category.

step 2: Based on the textual description for each use case it must be decided which of the three objects must be maintained to fully represent, in computational terms, the use case, taking into account the whole system and not considering each use case *per si*, as in a reducionistic approach.

step 3: The survival objects must be aggregated whenever there is mutual accordance for a unified representation of those objects.

step 4: The obtained aggregates must be linked to specify the associations among the existing objects.

This approach aims to obtain a holistic set of objects (resources of the system), so that the inter-relations amongst the objectified use cases can be successfully simplified in order to obtain a reduced number of relevant and pertinent system-level objects.

For control-dominated systems, this strategy makes natural the need to have a controller, which is an object of the application that is responsible for controlling the system process and the flow of information among the system components. The object diagram presented in fig. 4 is the result of applying the 4-step rule set to the use case diagram depicted in fig. 3. As a user-readability pursuit, it is always a good practice to encapsulate as much as possible the system's representations using packaging. Fig. 4 illustrates the usage of several packages that define, each one, decomposition regions which contain several tightly semantically-connected objects. These packages should be further specified, in what concerns its architectural structure. Objects 9.3.c and 10b.1.c are the system's controller.

Use cases specify the functionality of the system, whilst the object diagram is related to the structure of the system, which is used as the foundation for the design and implementation phases. The object diagram represents an ideal architecture for the system, because its construction was completely independent of any implementation issue (platform, programming language, processor, etc.).

6. CLASS DIAGRAMS

The majority of the methodologies do not pay too much attention to the object diagram. Usually, the class diagram is built firstly, but in this project the order was reversed. To develop embedded control systems, the authors believe that it is more important to have a good object model than a good class diagram, because the elements that do constitute the system are the objects and not their classes. This was the main reason to first identify the objects and to later select the classes to which those objects belong. Obviously, the best situation is having good object diagrams and good class diagrams. The authors are not advocating to roughly treat the class diagram or even to ignore it, but to direct the attention towards the construction of the object diagram.

This perspective that puts classes in an apparently secondary role may be classified by some specialists as object-based rather than object-oriented. However, the approach that firstly defines the objects and later the classes is somehow consistent with the bottom-up discovery of inheritance
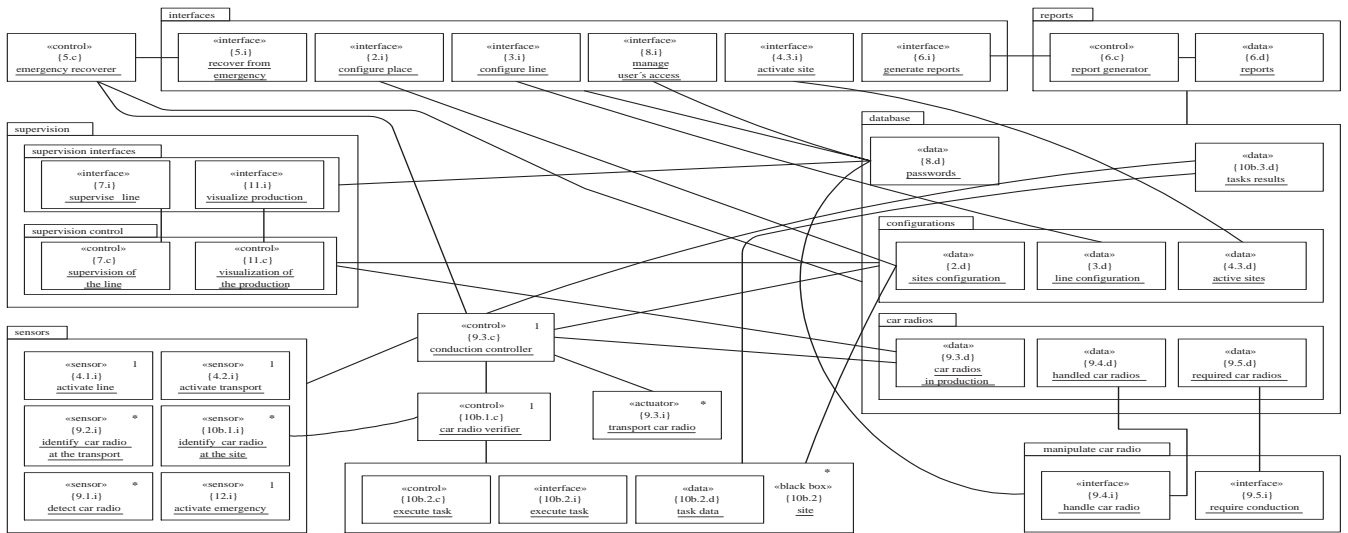
Figure 4: The object diagram.

to organize the classes [11]. Additionally, and without sub-estimating the benefits of using classes, some self-designated object-oriented methodologies start to relegate inheritance to a less important position [12].

Thus, it seems valid the approach proposed here: identify first the objects and then classify them. During the classification of objects the class structure is built, modified, or ideally just used. Reuse can be achieved in 3 different ways, during the classes discovery. First, if there are more than one object of the same class, their definition is specified in just one place. Second, if classes with similar properties are found, hierarchical relations among those classes can be defined. Finally, when a class is described, the developer can recognize the existence of that class in a library, which allows it to be immediately reused.

The class diagram is understood as a template for a set of applications that can be obtained from it. In other words, the class diagram is a high-level generalization of the system [13]. When the developers define the way classes are interrelated, they are indicating all the systems (or all the configurations) that can be obtained from those classes.

With this perspective, it is common, in several methodologies, to not build the object diagram, since it automatically results from the class diagram. Whenever an object diagram is constructed, it is necessary to guarantee that the relations expressed in the class diagram between two classes also exist between instances of those classes. This is the main reason that methodologies usually impose (or suggest) class diagrams to be first elaborated than object diagrams.

This implies an additional task in which it must be assured that there is consistency between the information that is described by both diagrams [9]. This fact can be interpreted as a symptom that some information is being unnecessarily replicated. For instance, the existence of the «singleton» stereotype in UML, which indicates that a given class can only have one instance, corroborates the perspective that sees the class diagram as a pattern for the systems, within a given application domain.

This approach seems adequate to develop business information systems or, more generally, any data-dominated system, where the objects are created and destroyed during the system life cycle. For example, in a system for bank accounts management, it is common that each account is always associated with, at least, one customer. This fact is indicated in the class diagram by associating the account class with the customer class. When an account object is created, it must be linked to, at least, one customer object. This approach is useful for business information systems, but does not offer many benefits for developing embedded systems, since normally the objects that constitute the system are not created and destroyed on the fly. An embedded system is generally composed of a set of fixed objects that are linked in some way. Thus, it is not crucial to indicate, for example, that objects of the controller class need to be linked with objects of the sensor class. If this information is relevant in some applications, it may be inadequate or even wrong in others. The authors see the class diagram as a repository of previously defined objects' specifications ("a raw material store"), that can be used to develop any application.

7. STATE-CHART DIAGRAMS

For those objects that have a complex or interesting dynamic behavior, a state diagram should be specified. For control embedded systems the application of Petri nets to the specification of the behavioral view has given origin to several meta-models intentionally developed to deal with the semantical particularities of that kind of systems [14; 15]. However, since UML was chosen as the notation for all the documentation of the project, UML's state-charts were used to describe state diagrams [16].

The crucial components of the control application are objects 9.3.c and 10b.1.c, and state-charts were produced for these objects to specify their dynamic behaviors. Object 9.3.c is responsible for controlling the movements of the radios along a Hidro line. Due to its great complexity, this object was decomposed into smaller objects, each one respon-

sible for coordinating one node (a set of plant resources). Since the different nodes of one Hidro line have not the same configuration, each kind of node requires a different state-chart. However, there are similarities among the different categories of node controllers, so a class hierarchy can best indicate their relations. For instance the state-chart of an upper node with 3 lines is similar to an upper node with 3 lines and one elevator to transport kits to the lower level. Fig. 5 depicts the top state-charts of an upper node with 3 lines and no elevator. Same states are super-states (their do-activities are prefixed with an "@"), but due to the lack of space the corresponding state-charts are not presented here. Similar state-charts were created for the other categories of node controllers.
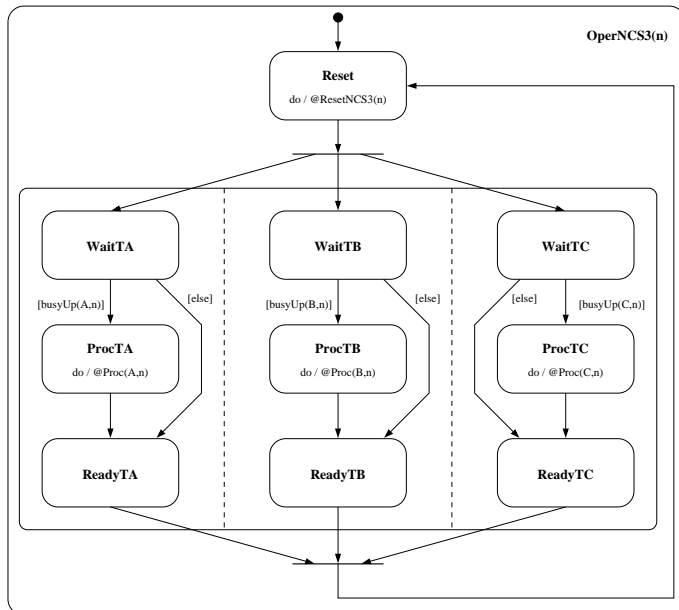

Figure 5: The top state-chart diagram.

8. CONCLUSIONS

This paper has presented how UML can be used in real industrial projects to model embedded control systems. The authors consider UML as an adequate unified representation language for industrial projects, since it is intuitive for non-technical people, it covers the main views of a system, it is independent of the platform and it is a standard.

The approach presented puts more emphasis on objects rather than on classes, which is one of its main divergences in relation to the traditional object-oriented approaches. The transformation from use cases into objects is one of the most important tasks within the development process. An holistic approach is followed during this transformation, so that it is possible to obtain, in a semi-automatic migration step, the object diagram that best maps the user's requirements into the system's requirements.

To ease the mapping of the models, tagged values (designated reference) are associated to the modeling elements (use cases and objects). This mechanism is useful, since it allows to circum-navigate throughout the whole complementary views of the system model, enabling the use of an operational approach within the spiral model-based analysis phase of system development.

This UML-based modeling approach was validated with a real industrial case study. Although this paper just covers the analysis phase, the authors are using UML models in the design and implementation phases. The authors are aware that the approach needs to be applied to more projects to gain experience and to improve some of the guidelines, namely in what concerns real-time constraints and model refinements during the design phase.

9. REFERENCES

[1] R.J.Machado, J.M.Fernandes, A.J.Esteves, H.D.Santos. *Hardware Design and Petri Nets*, chapter 11: "An Evolutionary Approach to the Use of Petri Net based Models: From Parallel Controllers to HW/SW Co-Design". A.Yakovlev, L.Gomes, L.Lavagno (eds.), Kluwer. To be published in 2000.

[2] A.J.Esteves, J.M.Fernandes, A.J.Proença. *Embedded System Applications*, chapter 3: "EDgAR: A Platform for Hardware/Software Codesign", pp.19–32. Kluwer, Jun.1997.

[3] J.M.Fernandes, M.Adamski, A.J.Proença. VHDL Generation from Hierarchical Petri Net Specifications of Parallel Controller. *IEE Proceedings: Computers and Digital Techniques*, 144(2):127–37, Mar.1997.

[4] R.J.Machado, J.M.Fernandes, A.J.Proença. An Object-Oriented Model for Rapid Prototyping of Data Path/Control Systems — A Case Study. In *9th IFAC Symp. on Information Control in Manufacturing*, vol.2, pp.269–74, Jun.1998.

[5] R.J.Machado, J.M.Fernandes, A.J.Proença. Hierarchical Mechanisms for High-level Modelling and Simulation of Digital Systems. In *5th IEEE Int. Conf. on Electronics, Circuits and Systems*, vol.3, pp.229–32, Sep.1998.

[6] G.Booch, J.Rumbaugh, I.Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[7] L.F.Andrade, J.C.Gouveia, P.J.Xardoné. Architectural Concerns in Automated Code Generation. In *OOPSLA Midyear Conference*, 1998.

[8] G.Schneider, J.P.Winters. *Applying Use Cases: A Pratical Guide*. Addison-Wesley, 1998.

[9] B.P.Douglass. *Real-Time UML: Developing Efficient Objects for Embedded Systems*. Addison-Wesley, 1998.

[10] I.Jacobson, M.Christerson, P.Jonsson, G.Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, 1992.

[11] J.Rumbaugh, M.Blaha, W.Premerlani, F.Eddy, W.Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall International, 1991.

[12] D.Budgen. *Software Design*. Addison-Wesley, 1994.

[13] A.Lyons. UML for Real-Time Overview. Technical report, ObjecTime Limited, Apr.1998.

[14] R.J.Machado, J.M.Fernandes, A.J.Proença. Specification of Industrial Digital Controllers with Object-Oriented Petri Nets. In *IEEE Int. Symp. on Industrial Electronics*, vol.1, pp.78–83, Jul.1997.

[15] M.Sgroi, L.Lavagno, Y.Watanabe, A.Sangiovanni-Vicentelli. Quasi-Static Scheduling of Embedded Software Using Free-Choice Petri Nets. In *1st Workshop on Hardware Design and Petri Nets*, pp.26–45, Jun.1998.

[16] D.Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8:231–74, 1987.