The International Workshop
On Discrete-Event System Design, DESDes'01,
June 27-29; Przytok near Zielona Gora, Poland

SIMULATION AND TARGETING USING OORT*

Sérgio LOPES, João MONTEIRO

Industrial Electronics Departament, Engineering School, University of Minho,
Campus de Azurém, 4800-058 Guimarães, PORTUGAL,
<*sergio.lopes, joao.monteiro*>*@dei.uminho.pt*

Abstract. *The development of embedded systems requires both tools and methods which help the designer to deal with the higher complexity and tougher constrains due to the different hardware support, the often distributed topology and time requirements. Moreover, the last steps of each version of the design, namely, simulation and targeting, should be made easier and faster to execute, in order to facilitate the correction of problems and the issue of a new more correct version, thus increasing the frequency of an iterative engineering process. This has a major impact on the overall costs and final product quality, and therefore the use of a CASE tool that supports the chosen methodology becomes an obvious advantage. We have applied the Object-Oriented Real-Time Techniques (OORT) method, which is oriented towards the specification of distributed real-time systems, to the implementation of the Multiple Lift System (MLS) case study. This paper describes briefly the method and presents our experience in the simulation and targeting of developed system, namely the difficulties we had and the success we have achieved.*

Keywords*: Distributed Systems Specification, Software Engineering, Discrete-Event Systems Control, Simulation, Targeting.*

1. INTRODUCTION

Real-time systems are very complex because they are often distributed, run in different platforms, have temporal constraints, etc. The development of these systems demand high quality and increasing economic constraints, therefore it is necessary to minimise their errors and its maintenance costs, and deliver them in short deadlines.

To achieve these goals it is necessary to verify a few conditions: decrease the complexity of the systems through hierarchical and graphical modelling for high flexibility in the maintenance; protect the investments with the application of international standards in the development; to apply early verification and validation techniques to reduce the errors; and, reduce the delivery times by automating code generation and increasing the level of reusability. Finally, its necessary to have a tool that provides these conditions. The present work was developed with the *Object*GEODE[i] toolset, that supports the OORT method.

The OORT method [14] is organised according to the diagram of figure 1 and applies the Unified Modelling Language (UML), Message Sequence Chart (MSC) and Specification and Description Language (SDL). The UML language is a de jure standard (see [5] for details) and it is defined in [10]. The MSC was defined [8] as complement to SDL, both international standards by ITU-T. [13] provides an introduction to MSC. SDL is defined by [6], [7] and [9],

* This work was supported by the Fundação para a Ciência e a Tecnologia PRAXIS XXI program of the Ministério da Ciência e Tecnologia of the Portuguese Government.

however [11] is a more comprehensive reference, while [4] is a handy summary of the language. The use of both languages together is guided by [10].
In this work we have applied the OORT method to the modelling of a case study – the Multiple Lift System (MLS). A description of a MLS architecture using UML is presented in [2]. The analysis model uses UML to model the system's environment, and MSC to specify the behaviour of the system. The system's architecture is defined in SDL. The detailed design uses SDL for the concurrent objects specification and UML for the passive components description. The MSC language supports the test design activity. For the simulation of the designed system we used the *Object*GEODE simulator, and finally we use the C Code Generator for the targeting. Each of this steps in the systems engineering process is described in the following sections.

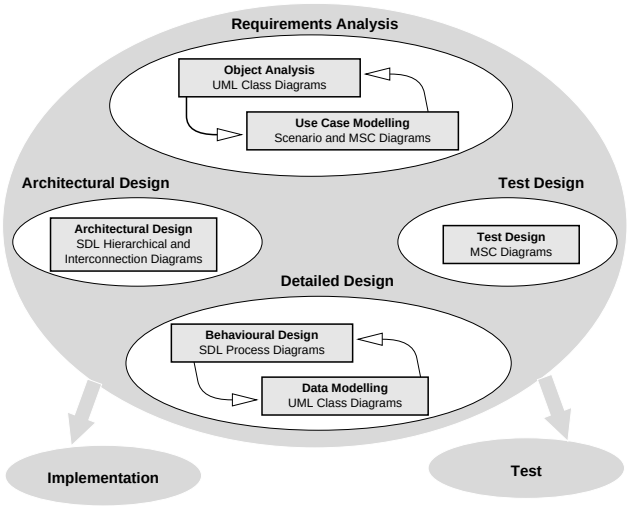

Figure 1. The OORT method.

2. REQUIREMENTS ANALYSIS

In the requirements analysis phase, the system environment is modelled and the user requirements are described. The analyst must concentrate on **what** the system should do. The environment where the system will operate is described by means of UML class diagrams – **object modelling**. The functional behaviour of the system is specified by MSCs organised in a hierarchy of scenarios - **use case modelling**.
The system is viewed from the exterior as a black box with which external entities (system actors) interact. Both the object model and use case model must be independent of the solutions chosen to implement the system.

2.1. Object Analysis

In the description of the system environment the class diagrams are used to express the application and services domains. This is done by identifying the relevant entities of the application domain (physical and logical), their attributes, and the relationships between them. It is also necessary, for the sake of simplicity and expressiveness, to group entities and their relationships in different modules that reflect different perspectives of the system, as is supported by [16]. Generally speaking, there is one module for each of the actors that interact with the system, one for some basic system composition and other to express certain environment relationships.

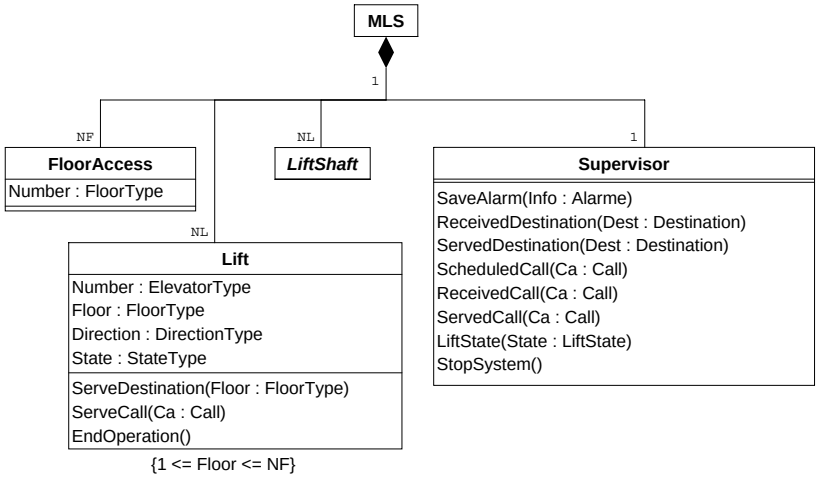

Figure 2. System Architecture UML Class Diagram.

The generic system architecture is modelled in figure 2. In order to keep simple modules, each of the component classes are refined in different diagrams.

2.2. Use Case Modelling

The use case model is composed by the scenario hierarchy and MSC diagrams. The scenario hierarchy should contain all the different expected scenarios of interaction between the system and its environment. The goal it is to model the functional and dynamic requirements of the system. First, the main scenarios are identified, and then they are individually refined in subsequent more detailed scenarios until the terminal scenarios can be easily described by a chronological sequence of interactions between the system and its environment.

One problem of this approach is the scenario explosion. To deal with that difficulty we apply composition operators that combine hierarchically the several scenarios. Nevertheless, the problem is only diminished but not completely solved. It is still necessary to choose well the scenarios, namely to chose those which are the most representative of the system behaviour.

The system operation is divided in phases that are organised by composition operators, and each phase is a branch in the scenario hierarchy. Figure 3 shows the `Trip` phase scenario hierarchy, in which we have a `Floor Crossing` terminal scenario which is illustrated in figure 4.

A constant concern must be the coherence between the use case and the object models. See [14] for more details.

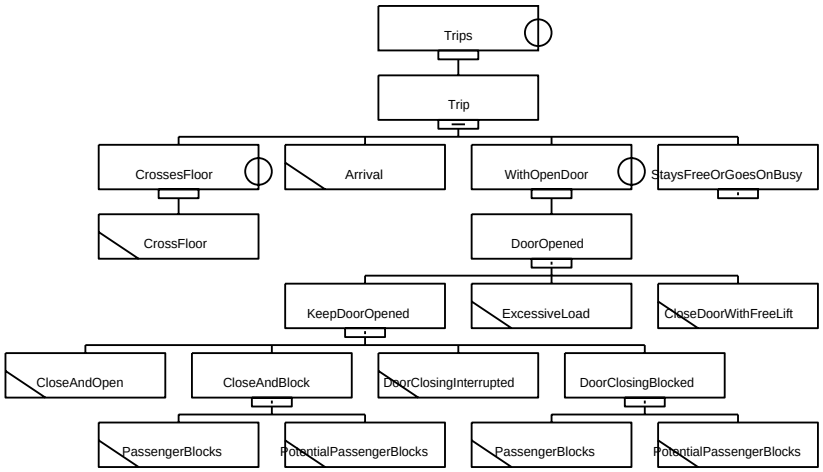

Figure 3. Scenario Hierarchy for the Trip Sub Scenario.

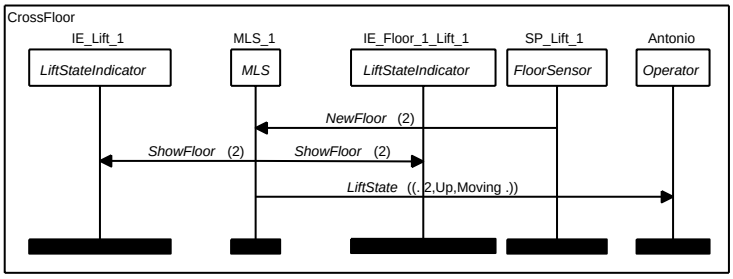

Figure 4. Abstract MSC for the Floor Crossing Scenario.

3. ARCHITECTURAL DESIGN

In this phase the system designers specify a logical architecture of the system (as opposed to the physical architecture). The SDL language covers all aspects of the architecture design.
The system is composed of **concurrent objects** (those which have an execution thread) **and passive objects** (those which implement a set of functions invoked by concurrent objects). In the architecture design phase, the concurrent objects that compose the system are identified and organised hierarchically. This is accomplished by a combination of refinement and composition. The refinement is a top-down process in which higher level objects are divided in smaller and more detailed objects, always trying to keep a good modularity. The composition is a bottom-up process in which designers try to group objects in such a way that favours reutilization and that maintains a good encapsulation of the architectural objects. Figure 5 illustrates the SDL object's hierarchy of the MLS.
In the architectural design, the real characteristics of the environment where the system will operate should be considered, as well as the efficiency aspects. On the other hand, the SDL model should be independent of the real object distribution on the final platform.

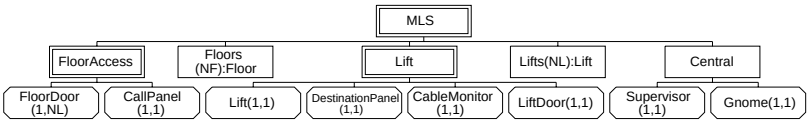

Figure 5. MLS SDL Hierarchy Diagram.

At the first level, the system actors are considered through their interfaces, and modelled as channels between the system top level objects and the outside world. Figure 6 shows the top level of the MLS architecture.

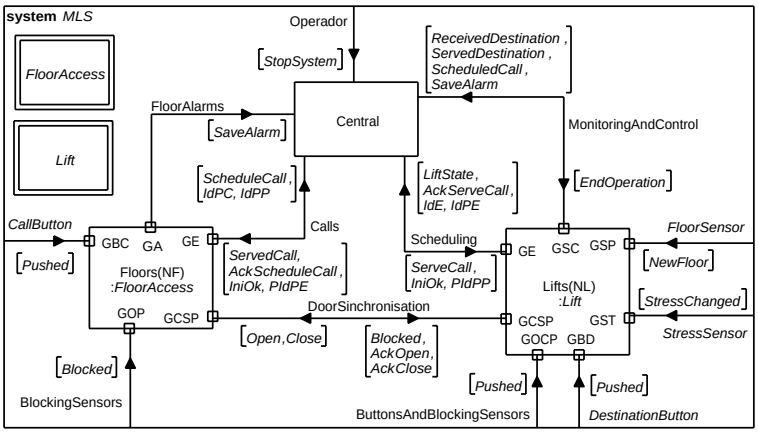

Figure 6. SDL Interconnection Diagram of the Top Level of the MLS Hierarchy.

Some passive objects are also defined, such as signals with complex arguments, Abstract Data Types (ADTs) associated with internal signal processing, and operators to implement the I/O communication with the outside world (instead of signals).
The use of SDL assures the portability of the system architecture, since the communication service is independent of the real object distribution, the communication channels are dynamic, and the objects can be parameterised.

4. DETAILED DESIGN

The description of concurrent and passive objects that constitute the system architecture is done in the detailed design phase. In other words, it is described **how** the system implements the expected services, and it should be independent of the final platform where the system will run.

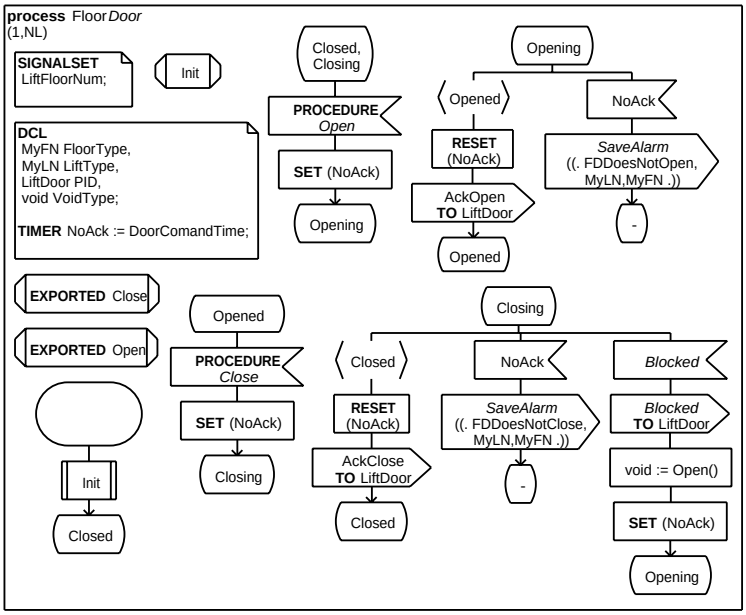

Figure 7. SDL Process Diagram of the Floor Door Process.

4.1. Concurrent Objects Design

The concurrent objects are the terminal objects of the SDL hierarchy. They are SDL processes and are a kind of Finite State Machine (FSM), with its states and state transitions, called **process diagrams**. The process diagrams are built by analysing the input signals of each process defined in the architecture model and how the answer to those signals depends on the previous states. The SDL has a set of mechanisms to describe the transitions that allow a complete specification of the process behaviour. In the figure 7 is shown a process diagram.
The reuse of external concurrent objects is supported by the SDL encapsulation and inheritance mechanisms.

4.2. Passive Objects Design

Some passive objects are identified during the analysis phase. Generally they model data used or produced by the system, and they are included in the detailed design to provide services to concurrent objects. There are also passive objects that result from design options, such as data management, user interface or equipment interface and inclusion of other design techniques.

Although the SDL ADTs provide a way to define passive objects they are better defined by UML classes. So the ADTs from the SDL detailed design model are translated to UML classes and organised in detailed design class diagrams.
The reuse of external passive objects is facilitated by the UML encapsulation and inheritance mechanisms. These characteristics of UML, and also SDL, allow for the use of other techniques of design in certain systems. For instance, in the case of embedded systems, it can be useful to use VHDL to design some physical parts.

4.3. Portability

The multi-tasking, the communication and the time management are implemented by the SDL virtual machine, and therefore are independent of the physical platform and RTOS on which the system will run. The system maintenance is kept at the SDL specification level, thus it is easier to correct and change the system. However, the portability depends largely on the language chosen to implement the passive objects.

5. TEST DESIGN

In this phase, the communication between all the elements of the system architecture is specified by applying detailed MSCs to describe the sequences of messages exchanged between them, in all the scenarios that compose the use case model. This is done by refining the abstract MSC of each terminal scenario from the analysis according to the SDL architecture model. Consequently, the test design activity can be done in parallel with the architecture design and serve as requirements to the detailed design phase.
In the intermediate architecture levels, the detailed MSCs represent integration tests between the concurrent objects. The last step of refinement correspond to unit tests that describe the behaviour of processes (the terminal SDL architecture level).
The process level detailed MSCs can be further enriched by including in each process behaviour detailed graphical elements such as states, procedures and timers.
Figure 8 shows the integration test corresponding to figure 4 abstract MSC, and figure 9 represents the respective unit test for one of the blocks.
This phase can be a very long and resource consuming, thus substantially increasing the system development cost. However, it is decisive to the system success.
The use case model reflects the user perspective of the system. The test design should be spread to cover aspects related to the architecture, such as performance, robustness, security, flexibility, etc.

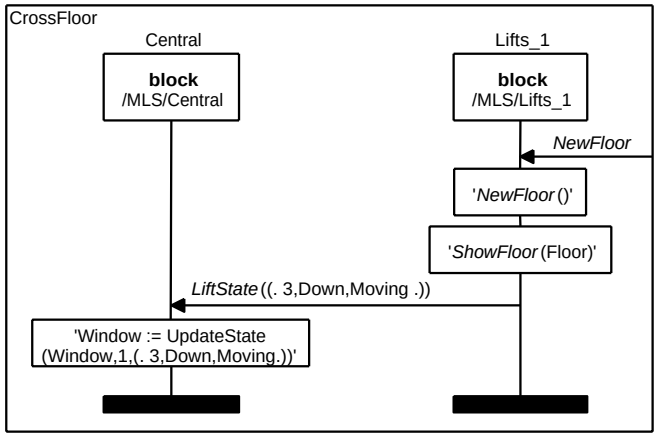

Figure 8 –Detailed MSC with Floor Arrival Integration Test.

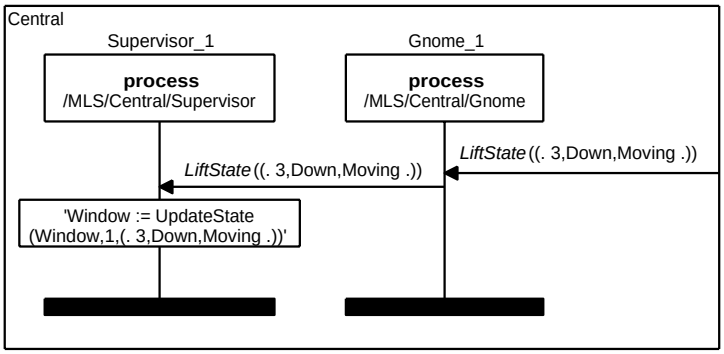

Figure 9. Detailed MSC with Floor Arrival Unit Test of Block Piso.

6. SIMULATION

With the *Object*GEODE simulator one can simulate SDL models, comparing them with MSCs that state the expected functionalities and error situations, and generating MSCs of the actual system behaviour. The execution of an SDL model is a sequence of steps, firing transitions from state to state.

The simulator has three operation modes: **interactive** - in which the user acts as the system environment and monitors the system's internal behaviour; **random** - the simulator executes the SDL model picking randomly one of the transitions possible to fire; **exhaustive** - the simulator automatically executes the model and explores all the possible system states.

The interactive model can be used to do the first tests to verify in a detailed way some important situations to correct and complete the overall behaviour of the system, i.e., to verify that the system really works. This mode was very useful to detect some flaws in ADTs whose operators were specified in textual SDL. For instance, the ADTs responsible for the calls dispatch, which are heavy computational, needed a little touch in the algorithms. As the SDL simulator has a granularity of one transition, we can not go step by step inside the operations executed during the transition from one state to another. But we can see that one transition does not follow the expected path or that some variable does not have the value it was supposed to have after that transition. Therefore, with that information, we can inspect more closely the operators called by the transition and verify their code correctness, but most of the times it is immediately evident which operator is wrong. This mode is specially suited for rapid prototyping.

Obviously, this is not an adequate way to simulate a large number of cases. After a certain level of confidence in the overall application behaviour is achieved, we can test for a larger number of scenarios, in order to detect dynamical errors such as deadlocks, dead code, unexpected signals, signals without receiver, overflows, *etc*. To do this we simulate in the random mode, to verify if the system is being correctly built. This mode allows to do the system verification.

However, we can do that with the exhaustive simulation, in fact we can do everything with the exhaustive simulation, but it would not be efficient? The exhaustive simulation requires a lot of computer resources and takes a lot of time. It's not something you can do everyday. The introduction of this mode between the interactive and the exhaustive is a very good solution because we can save a lot of time. We can detect most of the errors in a much quicker way.

The exhaustive simulation allows to make the validation of the system, *i.e.*, to verify if the system meets the requirements. We can check if it implements the expected services, by detecting interactions that do not follow some defined properties, or interaction sequences that

are not expected.

7. TARGETING

The implementation of the designed system is made easier by the code generator of the *Object*GEODE, which automatically translates the SDL specification to C code. The generated code is independent of the target platform in which the system will run. The SDL semantics, including the communication, process instance scheduling, time management and shared variables, is implemented by a dynamic library. That library is also responsible for the integration with the executing environment, namely the RTOS. By default the communications are implemented through TCP/IP sockets.

In order to generate the application, it is necessary to describe the target platform in which the system will be executed, This is done by means of a mapping between the architecture of the SDL specified system and the architecture of the C code implementation.

The SDL architecture consist in a logical architecture of structural objects (system, blocks, processes, etc...) in which the lower objects (the processes) implement the behaviour of the described system. The physical implementation of that description consist in a hierarchy of the following objects: **node** - all the software executed by one processing unit with multi-tasking OS; **task** - unit of parallelism of the OS. One task can correspond to one of the SDL objects: system - Task/System (TS) mapping; Block - Task/Block (TB) mapping; process - Task/Process (TP) mapping; Process Instance - Task/Instance (TI) mapping.

In the TI mapping the complete application is managed by the target OS. In the TP mapping, the OS is in charge of the interaction between processes, whilst the management of the several process instances inside the task is done by the SDL virtual machine of *Object*GEODE. In the case of TB mapping, the OS manages the communication between blocks, while the management of the SDL objects inside each block is done by the SDL virtual machine. Obviously, the TS mapping is the only one possible for operative systems without multi-tasking and in each node the SDL virtual machine manages all the application. For the MLS, the TP mapping was chosen.

After the code is generated, the user only has to supply the missing code for the parts that interact or depend directly on the platform. Therefore, the user can choose the language which best suits his needs and then link that code with the generated code. The ADT operators that do not interact with external devices, can be coded algorithmically in SDL, and thus the respective C code will be generated. By default, to each ADT operator corresponds one C function which interface is automatically generated. The figure 10 illustrates the application generation scheme in a very simplistic manner.

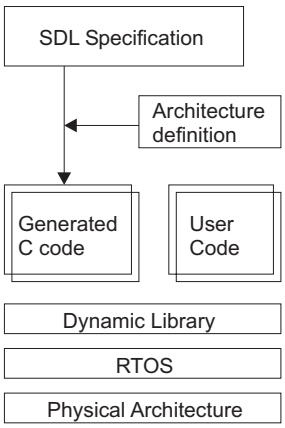

Figura 10. Simplified strategy for the application generation.

8. CONCLUSION

The simulation is a very important phase of the system's development because it allows the costs reduction by decreasing the number of missed versions, *i.e.*, it helps the designers to get closer to the "right at first time". The three simulation modes can be used by the order presented, *i.e.*, in the order of the increasing level of system correctness.
The code generated by the *Object*GEODE toolset is optimised for the target platform by means of a mapping between the SDL architecture and the physical architecture defined by the user. Any change in the application target it only requires a change in the mapping, so the system specification and its logical architecture remain the same. The user only has to supply the code which is target dependent.
Because SDL is a formal language it can be used to define rules in the partition and synthesis of a system specification into hardware and software, as is the case of a methodology presented in [1]. Furthermore, the implementation can be automatic, thus limiting the manual coding to the non real-time operations. The generated application is scalable, because the logical architecture is independent of the physical architecture. The mapping between objects and hardware is define in the implementation phase only.
The SDL specification, being a model expressed in a formal language, permits the automatic simulation of the system [3], to make early validations, and the automatic code generation. The simulation of a formal language is trustable since it is defined by a clear set of mathematical rules. Therefore, comparing to the non formalised development, the applications are better in terms of efficiency, less errors, flexibility and easy of maintenance.

REFERENCES

[1] J.M. Daveau, G.F. Marchioro, T. Ben-Ismail and A.A. Jerraya, “Cosmos: An SDL Based Hardware/Software Codesign Environment”, in: *Hardware/Software Co-design and Co-Verification*, eds. Bergé, J-M, Levia, O. and Rouillard, J., Kluwer Academic Publishers, 1997, 59-87.

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

[3] V. Encontre, How to Use Modeling to Implement Verifiable, Scalable, and Efficient Real-Time Application Programs, *Real-Time Engineering*, Fall 1997.

[4] O. Faergemand and A. Olsen, Introduction to SDL-92, *Computer Networks and ISDN Systems*, 26(9), 1994.

[5] Cris Kobryn, UML 2001: A Standardization Odyssey, *Communications of the ACM*, 42 (10), 1991, 29-37.

[6] ITU-T Recommendation Z.100, Specification and Description Languge (SDL), March 1993.

[7] ITU-T Recommendation Z.100 Appendix 1, SDL Methodology Guidelines, March 1994.

[8] ITU-T Recommendation Z.120, Message Sequence Chart (MSC), ITU, October 1996.

[9] ITU-T Recommendation Z.100 Addendum 1, Specification and Description Languge (SDL) Addendum 1, October 1996.

[10] ITU-T Recommendation Z.100 Supplement 1, SDL + Methodology: Use of MSC and SDL (with ASN.1), May 1997.

[11] A. Olsen, O. Faergemand, B. Moller-Pedersen, R. Reed, J.R.W. Smith, Systems Engineering Using SDL-92 (North Holland, 1994).

[12] OMG Unified Modeling Language Specification, Version 1.3, June 1999.

[13] E. Rudolph, P. Graubmann, J. Grabowski, Tutorial on Message Sequence Charts, *Computer Networks and ISDN Systems*, 28(12), 1996.

[14] Verilog, ObjectGEODE Method Guidelines (Verilog SA, 1996).

[15] Verilog, ObjectGEODE SDL Simulator Reference Manual (Verilog SA, 1996).

[16] E. Yourdon, Object-Oriented Systems Design: An Integrated Aproach (Prentice Hall, 1994).

[i] *Object*GEODE is a registered trademark by Verilog.