

# Experiences and Lessons Learned Using UML-RT to Develop Embedded Printer Software

L.A.J. Dohmen<sup>1</sup> and L.J. Somers<sup>1,2</sup>

<sup>1</sup> Océ Technologies BV, P.O. Box 101  
NL-5900 MA Venlo, The Netherlands  
{lajd, lsom}@oce.nl

<sup>2</sup> Eindhoven University of Technology, Dept. Math. & Comp.Sc.  
P.O. Box 513, NL-5600 MB Eindhoven, The Netherlands  
wsinlou@win.tue.nl

**Abstract.** From 1997 on Océ has used UML-RT (formerly ROOM) to develop its embedded printer software. This paper highlights our experiences in relation to a number of development process issues. Our conclusion is that UML-RT is well suited for developing embedded printer software, but still has a few shortcomings, mainly with respect to the specification and verification of hard real-time requirements.

## 1 Introduction

Embedded real-time software forms a continually increasing part of the development effort of a device like a printer or scanner.

Such a device has many actuators and sensors (like heating or engine control) that need to be controlled synchronously or asynchronously with the speed of the printing process. A sheet of paper travels with a speed of 0.5 meter per second through a printer and at the same time a toner image is produced through a photo-electric process. Both sheet and image should be synchronized when fusing the toner image on the sheet, requiring registration accuracy of tens of millimeters. At one moment in time more than 10 sheets may be traveling through the printer, each demanding attention from the embedded control software.

Among the typical characteristics of the embedded software is the fact that it has to control a physical environment that has non-deterministic and concurrent behavior. The software has to respond in time and is difficult to test because the target environment differs from the development environment. Last but not least, requirements are not specified at one moment in time, but gathered during the whole engineering phase. Often they are incomplete, ambiguous and subject to change.

Océ has set two main goals for the development of embedded software: the development should become more efficient (products should ship earlier and with less effort) and the quality level should be raised. As a consequence, the underlying technology basis for the development of embedded software should give ample and adequate support for issues like requirement specification, architecture description, reuse of

components, and testing. Some important aspects to consider are concurrency, timeliness, and understandability.

In the remainder of this paper we outline the history of our use of methods for embedded software development (section 2) and focus on our experiences with UML-RT with respect to the criteria mentioned above (section 3 and 4). Section 3 deals with process-related issues, whereas section 4 treats the modeling aspects of UML-RT. Finally, section 5 summarizes our main conclusions.

## 2 Evolution

Before going into detail about UML-RT, we first give a short overview of the history of our use of software development methods for embedded printer software.

### 2.1 Early Experiences

In the past we have used Ward/Mellor [1] to model the behavior of the real-time embedded control software. This method uses a number of different views to describe a system: entity-relationship diagrams model the data of a system, data flow diagrams show a.o. the control flow, and state transition diagrams show the state changes. Although Ward/Mellor uses environmental modeling (mapping elements from the system's environment into the model by means of event-lists) instead of functional decomposition, the resulting model is still presented as a functional model.

In our case, the real-time embedded software of a typical printer has to cope with a very large number of internal and external events, which turn out to be difficult to manage in single level state transition diagrams. The number of diagrams soon explodes. Component reuse becomes difficult as soon as small changes have to be applied to a new instance. The main problem is the fact that the consistency between models and actual code has to be kept manually. Generated code amounts to at most 20% of the total code size.

### 2.2 Transition to UML-RT

Confronted with the problems sketched above and the promising emergence of object oriented methodologies, we decided in 1997 to use the ROOM modeling language [2] supported by the ObjecTime Developer tool for the development of embedded software. Currently, the ROOM diagrams have been merged into UML, giving rise to the UML-RT extensions [3],[4] supported by the Rational Rose RealTime tool (RoseRT). In the future, UML 2.0 will encompass these extensions.

Because the diagrams in UML-RT are related explicitly, the tool is able to generate fully executable code. If one also models parts of the environment, the model can be simulated.

The primary modeling element in UML-RT is a capsule (called actor in ROOM). A capsule is a concurrent, active class with a precisely defined responsibility and interface. The dynamic behavior of a capsule is expressed by a state-chart (based on the state-chart formalism of Harel [5]). The interfaces of a capsule are specified by proto-

col classes that define exactly the valid messages and message sequences. Sequence diagrams are used to visualize the message flows between objects.

### 2.3 Current Status

We have used UML-RT initially during the period 1997 – 2001 for the development of a print-engine (DPS400) able to print up to 100 pages per minute. This product has been released and shipped in May 2001.

The print-engine contains approximately 120 sensors and 100 actuators. The developed software contains 350 capsule (actor) classes and 170 protocol classes. The total size of the code is 520 Kloc (executable code), from which the main part (490 Kloc) has been generated from UML-RT. Code reused from previous projects involves only a small part of the total code base. All engineers have attended UML-RT, C++, and operating system training courses.

Motivated by the success of this project, UML-RT is currently being used for the software development in four other projects.

The following sections show our experiences based on the development of the DPS400 print-engine mentioned above.

## 3 Process Related Experiences

Together with the modeling language UML-RT we have used a development process based on three approaches: architectural driven, requirements driven and iterative / incremental. In the early phases of the software development the architecture of the software system is defined. This architecture guides the software engineers through the succeeding phases by defining the major structure, behavior and interfaces of the system.

The software requirement specifications are the basis for all other development activities. They are defined in the requirements analysis phase, they drive the model construction in the design and implementation phase, and finally they are used to verify („Have we build the system right?“) and validate („Have we build the right system?“) the system in the testing phase.

To better control the development process we apply iterative (not everything perfect at once, paying attention to learning) and incremental development (not all requirements at once).

Applying a new technology like UML-RT for software development also has its consequences for the development process that is being used. The issues that are most important in our opinion are listed below.

**No Silver Bullet.** A good modeling language like UML-RT contributes to a more efficient software development, but yet alone does not guarantee success. Most engineering practices learned in the past (like requirements engineering, testing, quality assurance, project management, etc.) are still absolutely necessary for success.

A lot of software developers may feel attracted to a new technology and may tend to forget about the most suitable way it has to be applied in the development process.

**Development Effort Not Decreased.** Mainly because of the learning curves of the software engineers (new modeling language, new programming paradigm, changed development process, new case-tool) the development effort has not been reduced compared to previous projects.

Due to such initial effects it may be difficult to justify the introduction of a new methodology. Anyhow, it is very important to make a planning that explicitly takes into account the learning curve. Also, if other projects are going to adopt the new methodology, one should try to „reuse“ the experiences of the first project. Although very straightforward in theory, in practice a vast number of organizational problems may arise.

Because the introduction of new technologies is expensive, it is important to synchronize among the many projects that exist in a large R&D organization. An innovation management group should chart clear roadmaps for future technologies.

Another issue is whether one should use UML-RT also during prototyping activities. In our development, a lot of prototyping code has been written outside the case-tool domain. The use of two paradigms (plain C for prototyping code versus UML-RT for final code) had a negative effect on the ability of the software engineers to use UML-RT properly during the engineering phase.

**Improved Understandability.** The use of a highly visual modeling language greatly improves the understandability, resulting in a higher quality of the software. Peer reviews (which are commonly considered to be one of the most important practices to enhance the product quality) become much more efficient, because the software engineers quickly understand each other's models.

**Importance of System Requirements Engineering.** In our development process, requirements were written for all software modules, but no or poor requirements were written at system level. Because of the absence of such multi-disciplinary requirements, the software requirements lacked a stable basis and a lot of changes were caused by unclear system requirements.

## 4 UML-RT Modeling Experiences

In the previous section we concentrated on the implications of the introduction of UML-RT for the development process. This section shows the technical aspects we experienced with the usage of UML-RT in combination with the RoseRT development environment. Note that part of the actual experience already originates from the predecessors ROOM and ObjecTime Developer.

### 4.1 Strengths

First we highlight the positive points we experienced with the usage of UML-RT together with the RoseRT development environment. For each aspect, some concrete examples within the development of the DPS400 print-engine are given.

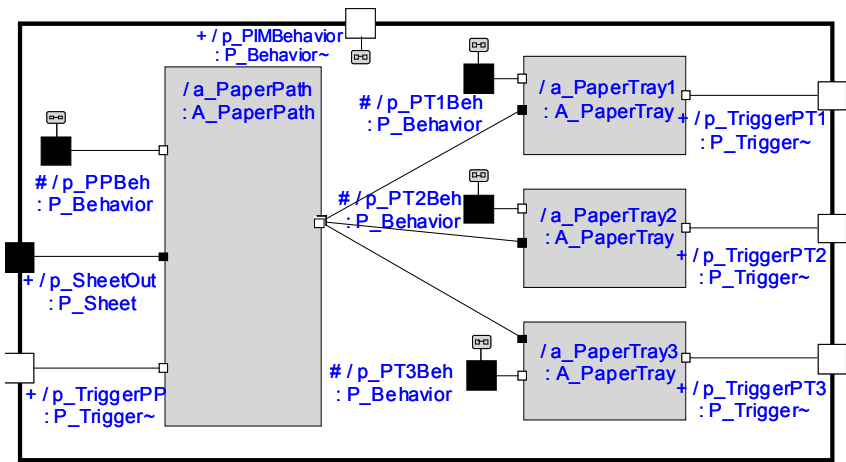


Fig. 1. Capsule of the Paper Input Module (PIM) with three paper trays

**Modelling the Physical Environment with Capsules.** The primary modeling element in UML-RT is a capsule (see for example Fig. 1). All capsules in a model will run concurrently and can respond to events in the physical environment.

Thanks to the use of parallel capsules, no complex software was necessary for task synchronization, e.g. no semaphores, message queues, task creation/deletion, etc. Because capsules are classes, physically identical functions can be controlled by just instantiating the particular capsule class twice.

**Hierarchical State-Charts.** With multi-level state-charts it is possible to describe complex behavior within one class. It is not necessary to decompose the static structure by creating more and more classes. Multi-level state-charts enable the reuse of top-level behavior (e.g. abstract behavior).

The higher level states (idle, standby, running, low power, etc.) are implemented in an abstract superclass. Each module subclasses from this superclass and automatically inherits all status behavior. Fig. 2 shows this behavior. This abstract behavior is also mentioned in [2] and is called „internal control“.

**Communication Based Approach.** UML-RT uses protocol classes for specifying the inter-capsule communication. The definition of protocol classes is vital for reusing components. Furthermore, it is a basis for polymorphism (different components can share the same interface), and necessary for separating the interface and the implementation.

The capsules within the DPS400 model can easily be replaced by new reusable components from the „reuse group“. As long as the interface is identical, replacing a component is just plug-and-play (black-box approach).

A powerful usage of polymorphism is the use of the status protocol. Each module has the same status protocol and reacts to the same messages. Only the internal behavior with respect to these messages is module-specific. For example, all modules receive the „initiate“ command, will do their specific initialization and will reply with „initiatedDone“.

**Sequence Diagrams.** The protocol classes define the valid messages, whereas the sequence diagrams specify the valid sequences of messages between several objects. Scenarios described in sequence diagrams provide an excellent communication mechanism among team members. Fig. 4 shows a representative sequence diagram.

In the architectural document sequence diagrams are used to describe the dynamic behavior of the system.

The test specifications use sequence diagrams to describe the messages sent to the „module under test“ and the expected response from the module.

Each service test initiated by a service technician has an associated sequence diagram describing the corresponding valid message sequence between the system and the environment.

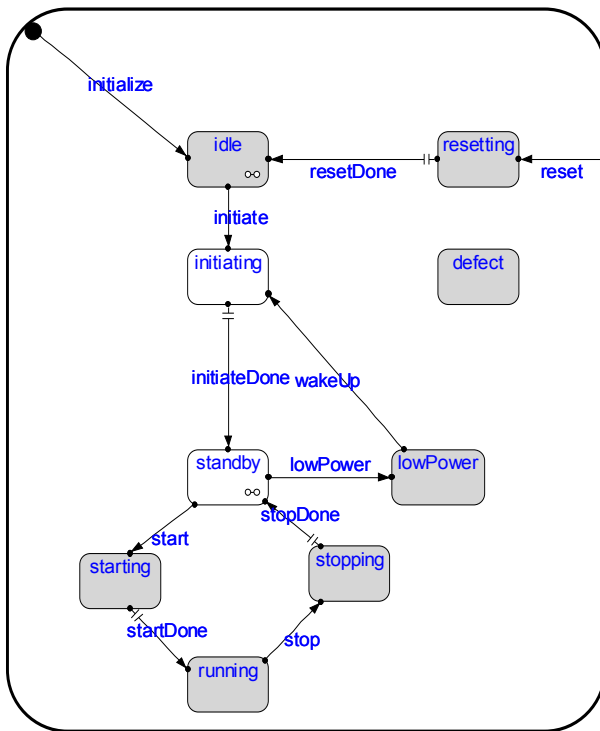


Fig. 2. State chart showing the abstract behavior

**Inheritance.** If a software component does not completely implement the desired behavior, inheritance allows the addition of the missing behavior in a derived subclass. Copy/paste actions (and thereby losing the relation with the original component) can be replaced by inheritance.

The executors in the DPS400 model (responsible for generating the action triggers) are all derived from one superclass. This superclass specifies 70-80% of the functionality and the derived executor only adds the module specific behavior.

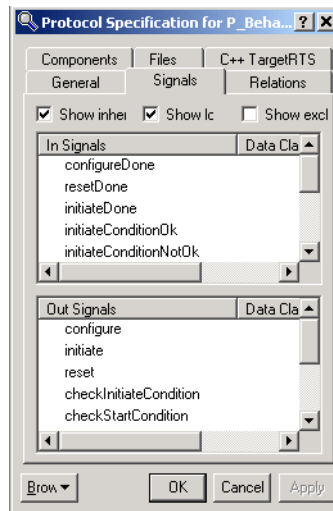


Fig. 3. Protocol class for controlling behavior

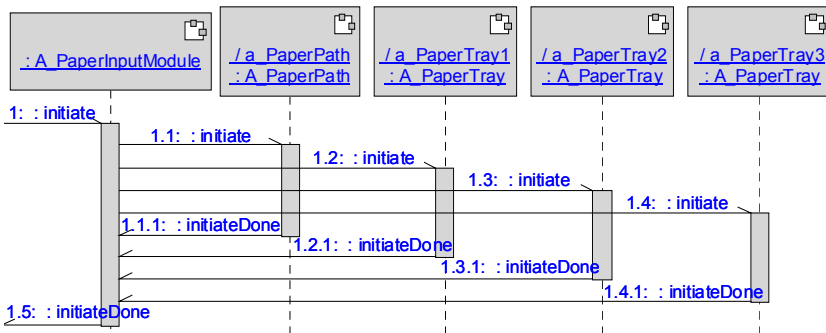


Fig. 4. Sequence diagram showing the initialization of the Paper Input Module

**Class Diagrams.** In ROOM it was difficult to see the relations between classes. However, UML-RT supports class diagrams to visualize class relationships.

**The Model Is the Implementation.** The typical separation between the design and implementation is eliminated. The model is always consistent with the implementation. The developers use the graphical interface to build the model and the tool set captures the structure, behavior, data and communication paths of the application. These models are automatically converted to C++ code.

All application code for the DPS400 has been generated from the graphical models. There is no translation from design to implementation. This means that all graphical models are consistent with the implementation.

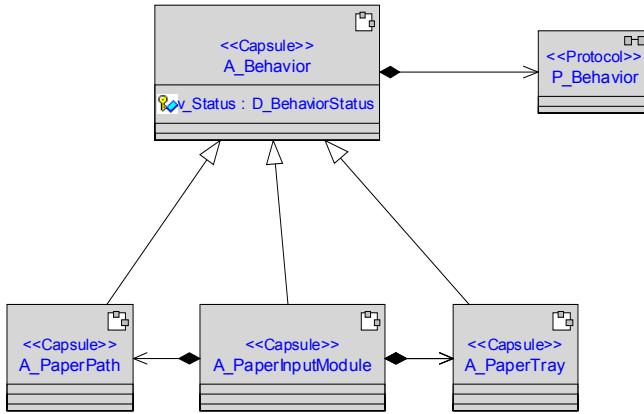


Fig. 5. Class diagram showing the relations between capsules for the Paper Input Module

**Executable Models.** The case-tool supports technology to execute the model and observe the running behavior. A popular „agile“ work-style approach of the developers (build a little, test, build a little more, and test again, etc.) is strongly supported.

Models can be validated on the host development environment using a simulation environment. Because an actual target is not needed, components can be tested independent from the target environment.

The software engineers use the simulation environment (on the host development system) to execute their models very early in the development cycle. This gives them feedback on the dynamic behavior of the software very early. Many bugs are found early and solved.

The first running model (prototype) of the service tests was executed on the host development system. All constructions were simulated and tested before going to the target hardware. The confrontation on the target worked the first time.

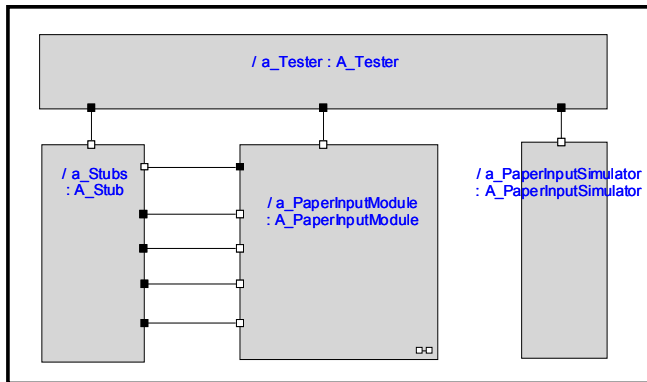


Fig. 6. Typical configuration of the simulation model

## 4.2 Weaknesses

Although our overall opinion about the usage of UML-RT is positive, some aspects still need improvement. This section gives an overview of the main problems we experienced.

**Difficult to Realize Real-Time Requirements.** UML-RT does not support the specification, implementation and verification of hard real-time constraints.

Due to this limitation of UML-RT, we had to write our own set of rules for dealing with real-time requirements. To facilitate the verification of the real-time behavior we have changed the RoseRT run time environment and added the logging of timestamps. The adapted run time system logs all messages in a file (including timestamps) and off-line this log file is translated into a sequence diagram (see Fig. 7). With this procedure we can verify the real-time behavior. But we cannot analyze the real-time behavior before implementing the complete software, nor can we prove the real-time correctness of the software.

**No Formal Analysis.** Although the tool-set supports early executing of models, it is not possible to perform any formal analysis like protocol deadlock or timing analysis. The latter is necessary to guarantee correctness, because a simulation or test will never execute all possible execution paths.

Timing analysis very early in the development phase can also guide the design choices where hard-real time requirements have to be met.

**Weak Support for the Testing Phase.** No tooling support is available for coverage testing like code-coverage or path-coverage. Tracing requirements and test case execution has to be done manually by the software engineer.

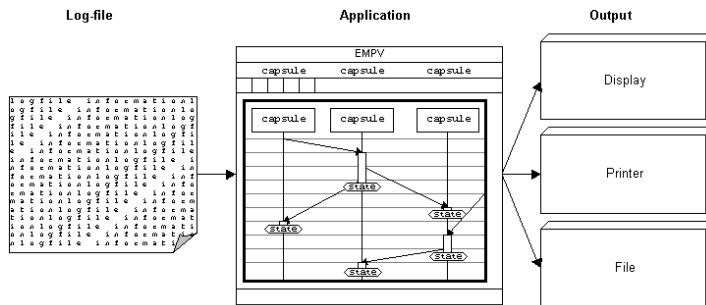


Fig. 7. Logging real-time behavior for verification purposes

Note that we did not use Quality Architect, which should be able to generate a test harness from a defined message sequence chart.

**Bad Target Observability.** It is very difficult to debug the software as soon as it has been downloaded to the target hardware. The offered tooling slows down the software execution and is therefore not usable.

It is hardly possible to view the performance and latency of the software running on the target hardware. Tuning the real-time behavior to meet all timing deadlines takes a lot of time.

## 5 Conclusions and Future Developments

Our two main goals for the development of embedded software, more efficient development and higher software quality, have only been met partially.

Although we have not seen a reduction in development effort, we believe that subsequent projects will benefit from the work achieved in this first project with UML-RT. The quality of the software increased due to the use of a visual modeling language and code-generation („the model is the code“).

The two main issues we still have to tackle are the timely availability of system level requirements (a software process issue) and a solution for protocol analysis together with the specification and verification of real-time constraints (a technical issue). For the second point, a model checking approach might help, but it is unclear whether it will scale sufficiently.

In retrospect, we first migrated from assembly to high-level languages to develop software. Now we experienced the transition to graphically oriented languages (like UML-RT). A graphical model is many times more expressive than any number of source files. This enables a higher level of abstraction while thinking about the „real“ problems.

To benefit even more from the chosen technology, we have started a „reuse group“ responsible for delivering software components to the projects using UML-RT. The strong encapsulation and the exact definition of interfaces in UML-RT potentially enable an easy reuse of components. The challenges of this group are more organizationally oriented and less technological.

A second initiative for the future is the definition of an Embedded Software Reference Architecture (ESRA). ESRA defines the architecture for all new projects. This reference architecture will serve as a stable base for the software components developed by the „reuse group“.

## References

1. Ward, P.T, Mellor, S.J.: Structured Development for Real-Time Systems. Prentice-Hall / Yourdon Press (1985)
2. Selic, B., Gullekson, G., Ward, P.T.: Real-Time Object-Oriented Modeling. Wiley, New York (1994)
3. Selic, B.: Using UML for Modeling Complex Real-Time Systems. In: Mueller, F., Bestavros, A. (eds.): Languages, Compilers, and Tools for Embedded Systems. Lecture Notes in Computer Science, Vol. 1474. Springer-Verlag, Berlin Heidelberg New York (1998) 250-260
4. Lyons, A.: UML for Real-Time Overview. Rational Software Corporation (1998). [http://www.rational.com/media/whitepapers/umlrt\\_overview.pdf](http://www.rational.com/media/whitepapers/umlrt_overview.pdf)
5. Harel, D.: Statecharts: A Visual Formalism for Complex Systems. Science of Computer Programming 8 (1987) 231-274