ISSN 0280-5316
ISRN LUTFD2/TFRT--5641-SE

# Design, Implementation and Verification using UML-RT in GSM Radio Base Station 2000

Deb Ghatek
Johan Olofsson

Departmen of Automatic Control
Lund Institute of Technology
May 2000

**Department of Automatic Control**
**Lund Institute of Technology**
**Box 118**
**SE-221 00 Lund Sweden**

*Document name*
MASTER THESIS

*Date of issue*
May 2000

*Document Number*
ISRN LUTFD/2TFRT—5641--SE

*Author(s)*
Deb Ghatek, Johan Olofsson

*Supervisor*
Martin Addibpour Ericsson Radio Systems
Karl-Erik Årzén LTH

*Sponsoring organization*

*Title and subtitle*
Design, Implementation and Verification using UML-RT in GSM Radio Base Station 2000

*Abstract*
This work deals with the issue of implementing a UML-RT standard, one of the latest notations for object oriented specification and design, in the developing-process of new real-time software for Radio Base Stations in the 2000 series at Ericsson. UML-RT is the real-time extension of the Unified Modeling Language (UML).
The thesis investigates the design-, implementation- and verification-problems that exist when combining the current RT functions, Multi Platform Support, with the UML-RT tool, ObjecTime Developer.
We describe UML, look at the advantages and disadvantages of using UML-RT tools and investigate current and future possibilities in ObjecTime Developer.

*Keywords*

*Classification system and/or index terms (if any)*

*Supplementary bibliographical information*

*ISSN and key title*
0280-5316

*ISBN*

*Language*
English

*Number of pages*
48

*Recipient's notes*

*Security classification*

# Preface

This report is a Master's Thesis in automatic control performed at Ericsson Radio Systems in Kista. We would specially like to thank Mattin Addibpour and Leif Andersson among others for their support during this work.

Ericsson Radio Systems, Stockholm, February 2000

Deb Ghatak,
Johan Olofsson

# Abbreviations

| | |
|---|---|
| ERA | Ericsson Radio Systems |
| FSM | Finite State Machine |
| GUI | Graphical User Interface |
| GSM | Global System for Mobile communication |
| MPS | Multiple Platform Support |
| MSC | Master Sequence Chart |
| OS | Operating System |
| PLS-Sim | PLatform Subsystem Simulator |
| RBS | Radio Base Station |
| ROOM | Real-time Object Oriented Modeling |
| RTS | Run Time System |
| SK | Soft Kernel |
| SU | Software Unit |
| UML | Unified Modeling Language |
| UML-RT | Unified Modeling Language for Real-Time |
| WCDMA | Wideband Code Division Multiple Access |

# Contents

# List of Figures

# 1. Introduction

As telecommunications evolve and spread, the business gets more and more competitive. The real-time developing process has to be faster, more complex and cheaper in order to expand the market share. The complexity in real-time systems arises from aspects like [1]:

- *Concurrency*. In a concurrent system, at any given time, multiple simultaneous activities can take place. Therefore, a concurrent system needs to support many processes depending on each other to be run simultaneously.

- *Dynamic behavior*. Real-time systems are often unpredictable. It is hard to predict when events will occur.

- *Variable loading*. The external environment using the real-time system uses the system quite variable. Sometimes the system has little to and sometimes not.

- *Memory and Processing Limitations on target platforms*. The hardware on the target platform is often limited in terms of processor- and memory-capacity.

This work will, with the use of the UML, act as a guideline for how to implement the UML-tool, ObjecTime Developer, in the development-process of new radio bases. As Object Oriented software development tends to be more popular the need for a single, common and widely usable modeling language is arising. In this report we will handle the complexity with the use of models, modeling language and the object paradigm.

- *Model*. A model is an abstraction that shows the important parts of a complex problem. Complex system demands different models to show different aspects of the sample problem.

- *Modeling language*. A modeling language consists of a notation (symbols that is used in the model) and a number of rules that defines how the model is used.

- *Object paradigm*. The object paradigm explains what is meant by an object. It is a combination of different techniques, like encapsulation, inheritance, polymorphism explained later.

Objects, according to definition are "entities that model some physical or conceptual entity" [2]. An object has some unique identities, a public interface (attributes and operation) and a hidden implementation.

## 1.1 Background

This work has been carried out at the Control and Transmission department, a department of Ericsson Radio Systems working with the development of transmission software in GSM Radio Base Stations (RBS). The complexity described earlier implies that the Control and Transmission department continuously has to evaluate different design alternatives in their effort to stay at the cutting edge of software development. To fulfill this goal the Control and Transmission department hopes that using a software development toolset, built specifically for the real-time domain, will secure future success.

Due to the rapid development of hardware platforms Ericsson Radio continuously investigate how to use these for their new Radio Bases. In the RBS system software design changes are needed because of these new hardware platforms, which implies new product configurations and limitations in the current software design. In the RBS system there are a lot of software that is hardware dependent and the software development has grown bigger and more complex. The goals are long term cost efficient solutions, possibilities to reuse software and supporting many products.

## 1.2 Problem Specification

The core problem in this thesis was to implement a UML tool, ObjecTime Developer, in the design and implementation phases

## 1.3 Objectives

The main objective of this thesis, is to enable external execution of existing real-time functions within ObjecTime, so that future development can be done in Objectime. One part objective was to create a communication link from ObjecTime to a simulator, PlsSim (HOST). The second part-objective was to implement the solution on an AMD target processesor. Future developments of radio bases at Ericsson will be made on PowerPC's, so after developing this Ericsson wanted to implement the same solutions on a PowerPC.

## 1.4 Limitations

This thesis will not try to explain the ObjecTime Developer environment, as this is better done working through a tutorial.

Neither will it describe the TRU model, described in Appendix A, since it is of specific interest only to those who works at Ericsson Radio, and of little interest to others.

The development environment on Power PC?s hasn?t been finalisedyet, sowe nevewr implemented the solution on a Power PC, however itwould havebeen done in the same manner as the AMD.

# 2. UML for Modeling Real-Time Systems

## 2.1 Introduction

“Developing a model for an industrial strength software system prior to its construction or renovation is as essential as having a blueprint for a large building” [4]. In large and complex real-time software systems it’s crucial to design the software with a sound architecture.

A good architecture simplifies the construction of the system and accommodates changes during the development process. A good modeling language includes fundamental modeling concepts/semantics and visual rendering of model elements. In figure 2.1 is UML used to form the requirements model and the design model. To translate the design model into ObjecTime code the ROOM notation is used. ObjecTime deploys an executable model that can be run in the Real-Time Services packages included in ObjecTime.

“.. UML is hot. People new to object and component development want an overview and that’s exactly what UML provides. It provides diagrams that describe the basic perspectives that OO and component designers routinely create to capture the important elements of the application they create”[3]. The Unified Modeling Language (UML) is a language for specifying, constructing, visualizing and documenting the artifacts of a software-intensive system. UML is appropriate for object-oriented software development because it provides support for modeling classes, objects and the many relationships among them, including association, aggregation, inheritance, dependency and instantiation.

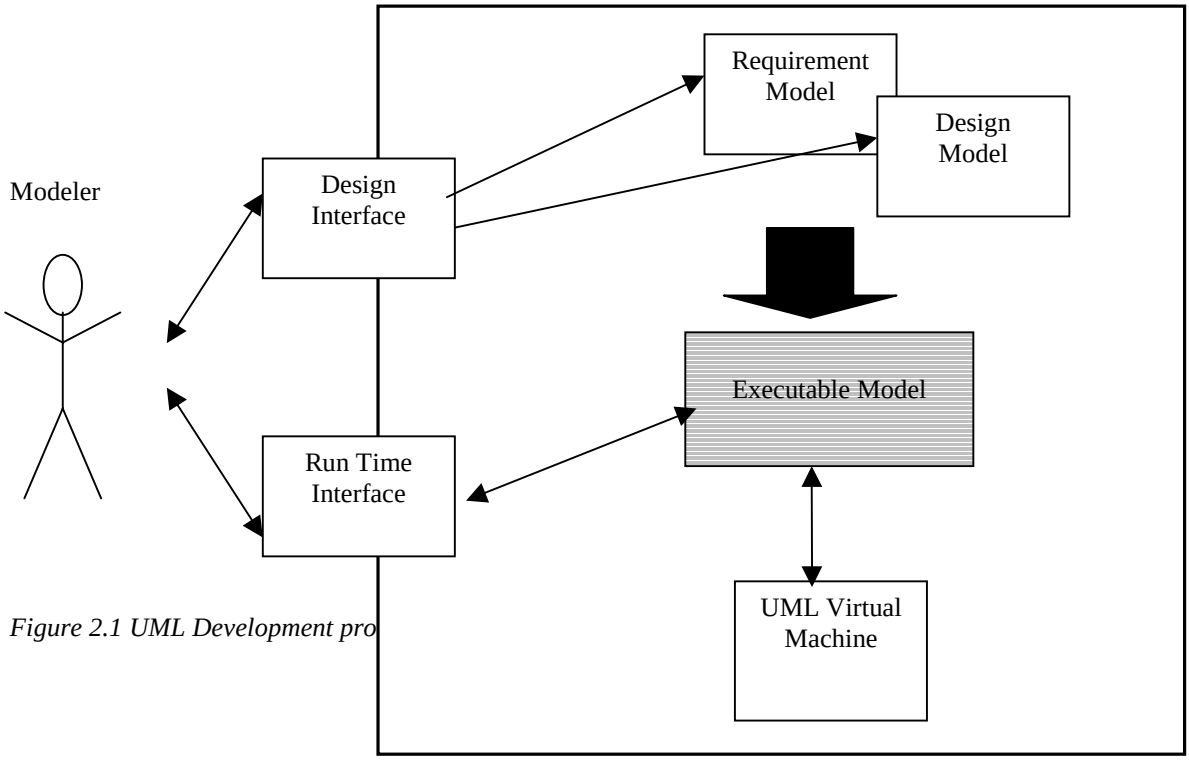


*Figure 2.1 UML Development pro*

## 2.2 The Unified Modeling Language

### 2.2.1 Introduction

The UML is recognized as a modeling language and not a methodology or method. The difference is that a methodology contains recommendations on object-oriented notation and design, while a modeling language is a vocabulary or notation on how to express the design.

### 2.2.2 UML Diagrams

When designing and developing software systems in UML nine different diagrams can be used. Different projects needs different diagrams. The diagrams form a skeleton of the complete design. They describe different aspects of the system as well as different steps in the development process. To get the full picture one must know what the diagrams shows and how to combine them. In Chapter 2.2.3 to 2.2.8 the different types of diagrams will be explained.

### 2.2.3 Use-Case Diagrams

A use case diagram provide a way of describing an external view of a system and its interaction with the outside world, it documents the behaviour of a system from the user's point of view. Figure 2.2 describes the outside world as actors, an user/actor can be a person or an another information system or a hardware device. A user/actor can have more than one role, and there may be many roles playing in it. The use-case diagram describes the interaction between the actor and the described system.

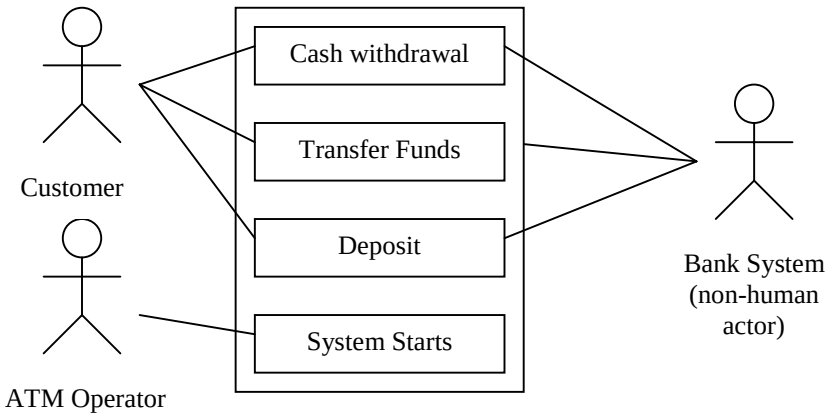


*Figure 2.2 Use-Case Diagram; Library*

Scenarios are instances of a use case, just as objects are instances of classes, In this way, use case diagrams are like class-diagrams they show the logical static structure of scenarios .

### 2.2.4 Class Diagrams

The class diagram is a central modeling technique that is used in most object-oriented methods. A class diagram shows the classes (sometimes objects) and relationships between classes and between objects. It is easy to follow the relationship between a book at a library and a staff member in the class diagram illustrated in Figure 2.3.

The class diagram is directly related to the source code, since all methods and attributes are listed, this makes it easy for the developer to transform the information into source code, and the developer can concentrate on implementing the methods.

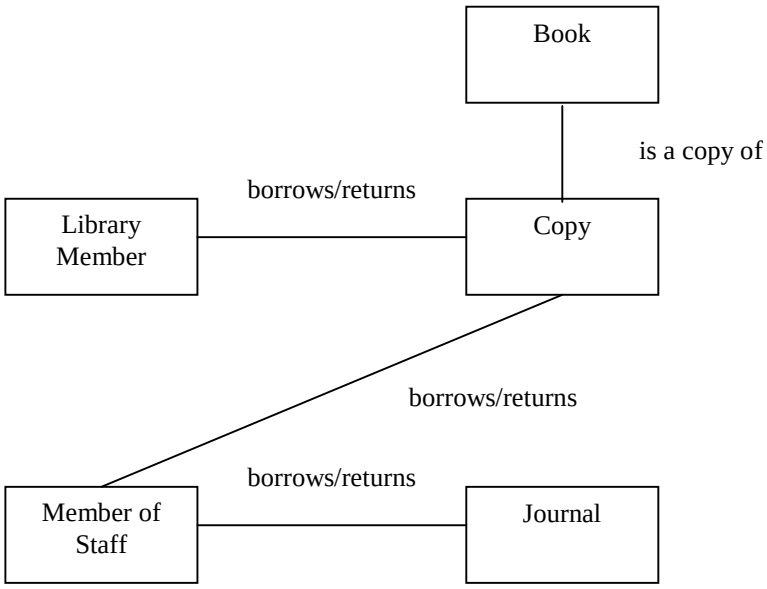


*Figure 2.3 Class Diagram; Library*

## 2.2.5 State Transition Diagrams

The basic idea is to define a machine that has a number of states, the state machine receives events and the events cause a transition from one state to another. In Figure 2.4 we describe how to set present time in a watch with two buttons, mode and inc(-rease), and three states.

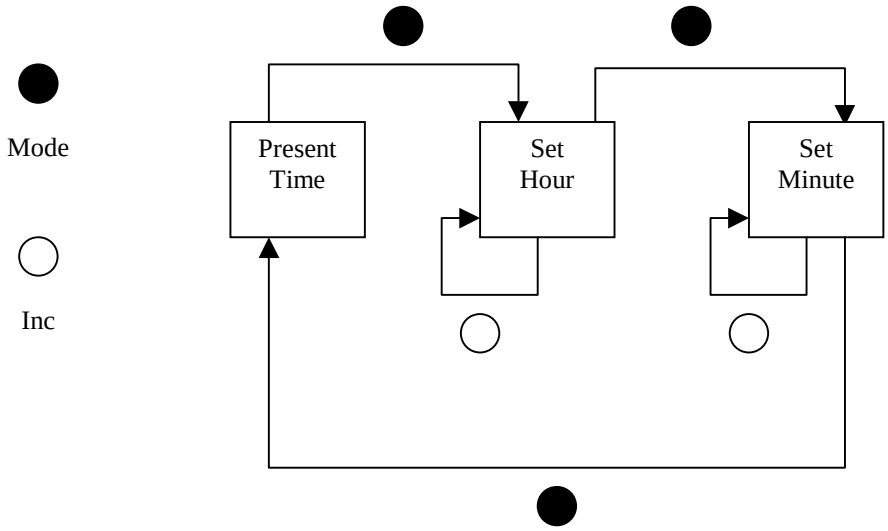


*Figure 2.4 State Transition Diagram; Watch*

The machine begins in a state when it displays the current time. If the user pushes the mode button, the machine jumps to the state where it displays the hour. Here the user can set the

hour by pushing the inc button. Another push on the mode button makes it possible to set the minutes. A final push on the mode button and the machine jumps back to the initial state.

The big disadvantage with State Transition Diagrams, is that one has to define all the possible states of a system. In small systems this is not a problem, but in larger systems the State Transition Diagrams become far too complex. Many object-oriented methods (e.g. ObjecTime) define separate state transitions for each class. State models are excellent for describing the behaviour of a single object, but not to describe whole systems.

## 2.2.6 Interaction Diagrams

Interaction Diagrams can be divided into three different forms of diagrams: sequence diagrams, collaboration diagrams, and activity diagrams.

A typical interaction diagram describes how a group of objects collaborate in some behavior. The diagram shows objects and the messages that are passed between them. Interaction diagrams are best used when you want to look at the behaviour in a single use case, but not so good in precise definition of the behaviour.

### 2.2.6.1 Sequence Diagrams

In this diagram the objects are shown as vertical lines (Figure 2.5) with the messages as horizontal lines between them. The messages can be an operation, a signal, a procedure call, or anything that starts an activity at the reception.

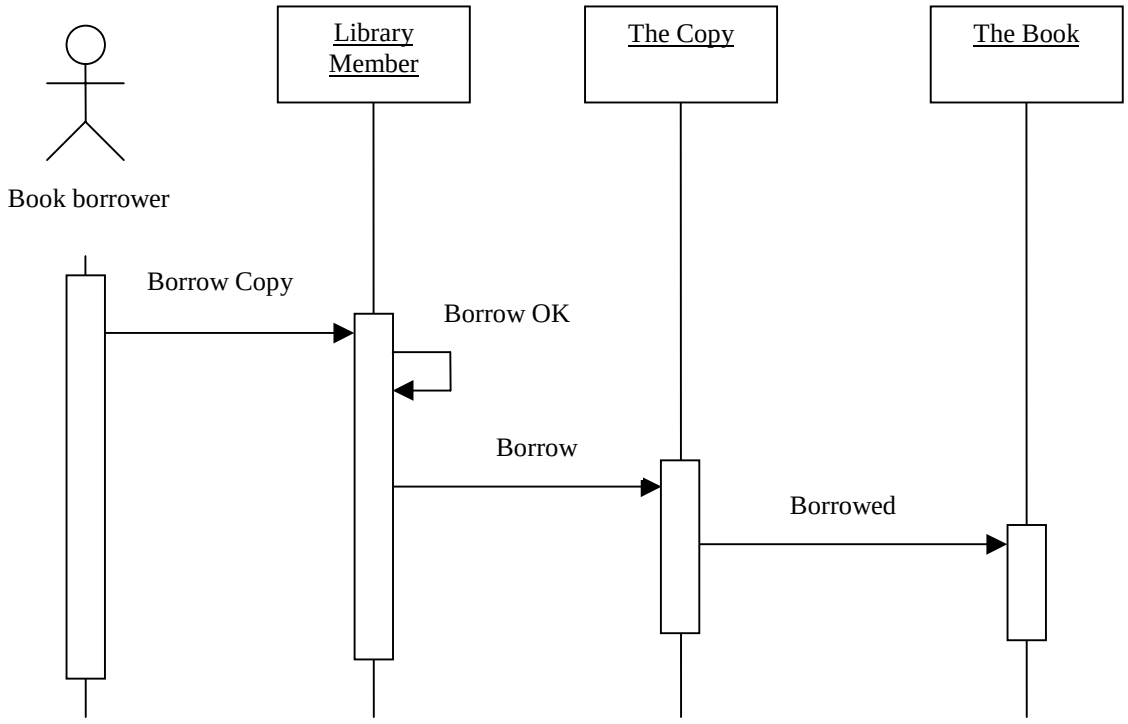


*Figure 2.5 Sequence Diagram; Library*

The sequence diagram is a description for a single use case scenario. This is helpful when we want to understand the logic of the operations during the design phase. As in the library example, the last transition registers that a copy of a certain book is borrowed.

### 2.2.6.2 Collaboration Diagrams

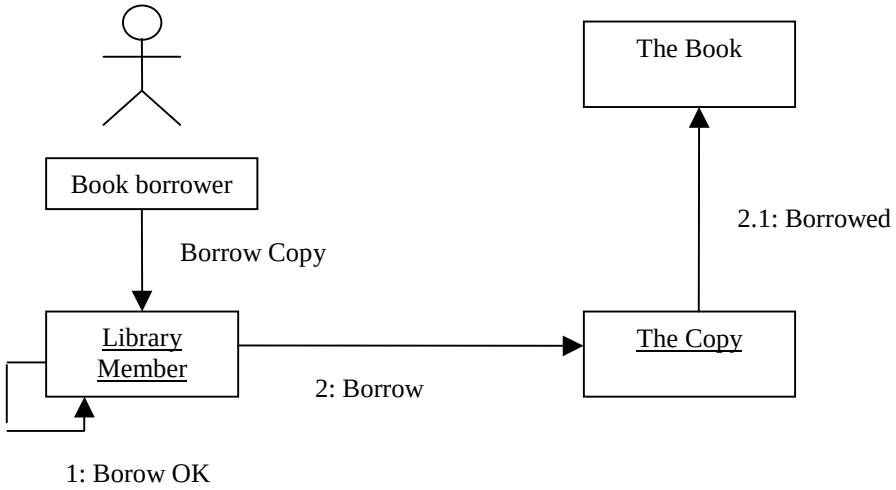


*Figure 2.6 Collaboration Diagram; Library*

The collaboration diagram shows the message flow between the classes and it numbers the messages in sequence.

A collaboration diagram, Figure 2.6, is often used as a complement to the class diagram, since they don't show the message flow between the classes. The rectangles represent the various objects in your application, and the arrow represents the flow of the message. The numbering of the messages makes it easy for the viewer/developer to see the many different usecase scenarios of the system.

### 2.2.6.3 Activity Diagrams

The Activity Diagram focuses on activities and the coordination of those activities, it is quite similar to the state chart diagram. It is a form of flowchart, but the difference is that the activity diagram supports parallel activities and their synchronization.

In the figure a stock-ordering procedure is illustrated. Parallelism is illustrated as a payment process paralell to the stock assignment process. Both processes have to be completed in order to settle an affair.

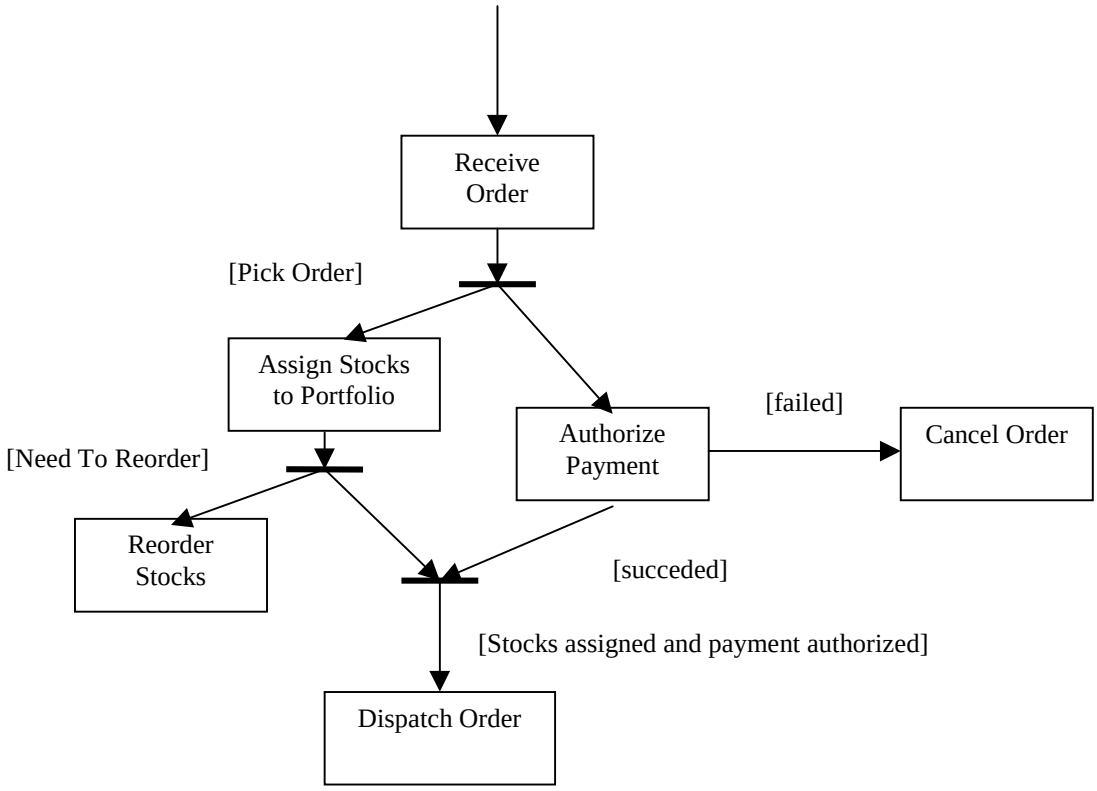


*Figure 2.7 Activity Diagram; Stock Trade*

## 2.2.7 Package Diagram

The package diagram has to do with the implementation of the system. In this technique we put each class in a single package. If a class uses another class in a different package, we have to draw a dependency line to that package. As seen in Figure 2.8 the ’Stock Pricer UI’ depend both on the ’GUI Library’ and the ’Stock Pricer’.

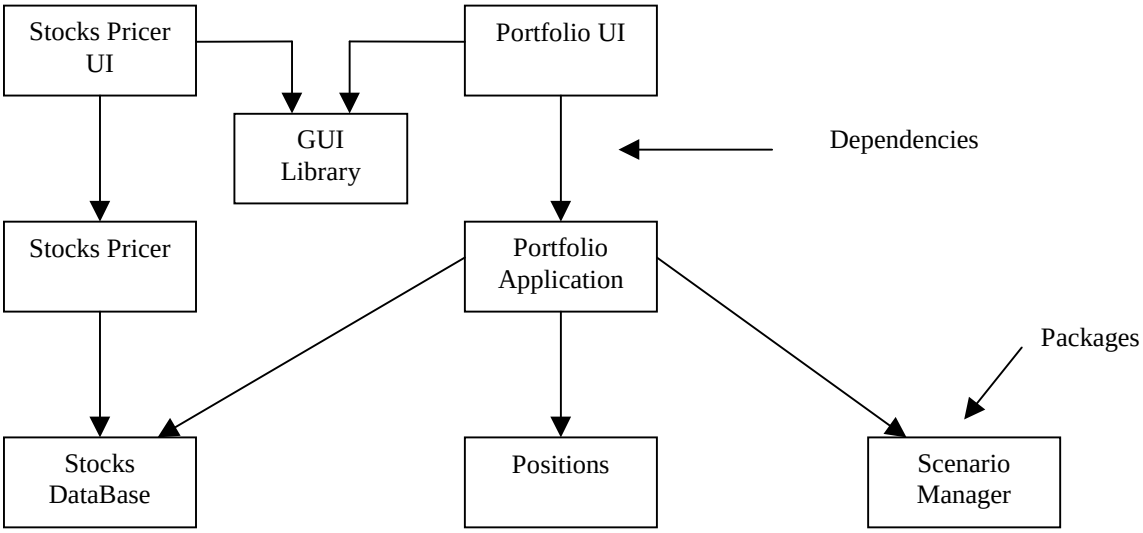


*Figure 2.8 Package Diagram; Stocks Data Base example*

UML does not treat package diagrams as a separate technique, it rather treats them as icons in a class diagram.

## 2.2.8 Deployment Diagrams

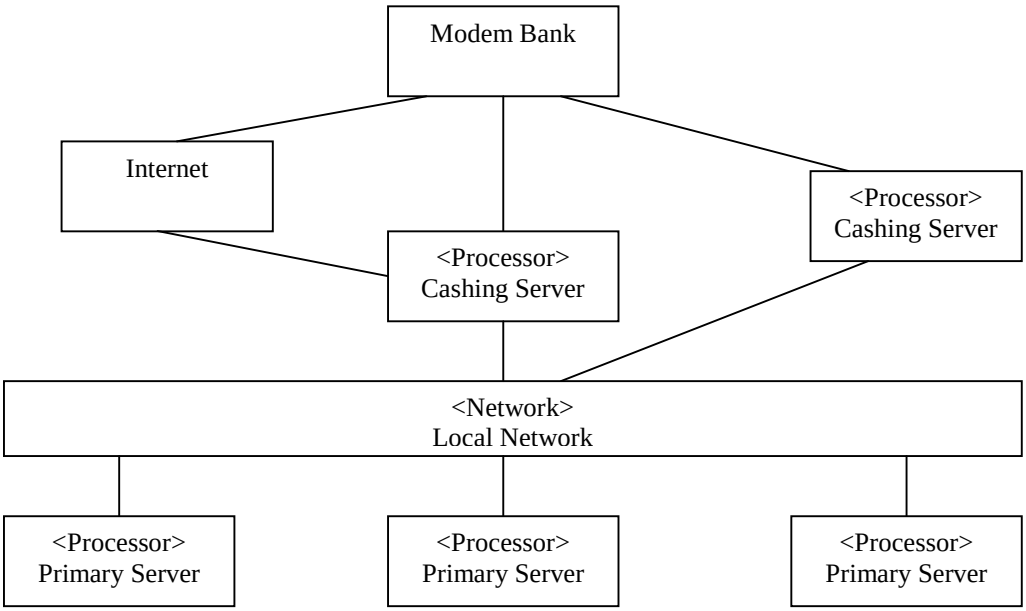


*Figure 2.9 Deployment Diagram; Internet*

Deployment diagrams have to do with the implementation of the system and it only shows the configuration of the run-time units. Components that doesn't exist in run-time are not shown in this diagram. The deployment model shows physical communication links between hardware items. Figure 2.9 shows all physical components needed for internet traffic with three servers. For each component in the deployment diagram, you have to document issues like transaction volume, network traffic, and the required response time. The components in the deployment diagram will then be described by other appropriate models.

## 2.3 How UML Diagrams Fit Together

The boxes in Figure 2.10 show the diagram that we have described below and the relationship between them. The arrows indicate an 'input into' relationship.

Figure 2.11 shows how the UML diagrams are used at different times of the design and development process. The arrows between the boxes represents that the previous diagram is documented by the following, e.g., a State Transition Diagram is documented by a Class Diagram.

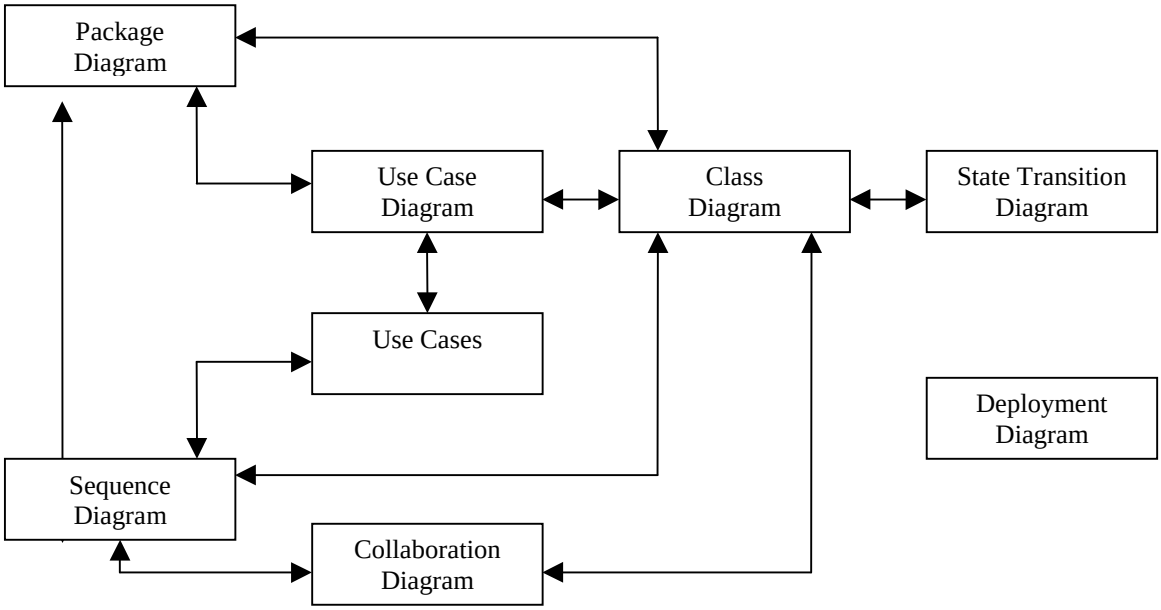


*Figure 2.10 Iterative Development with UML modeling techniques*

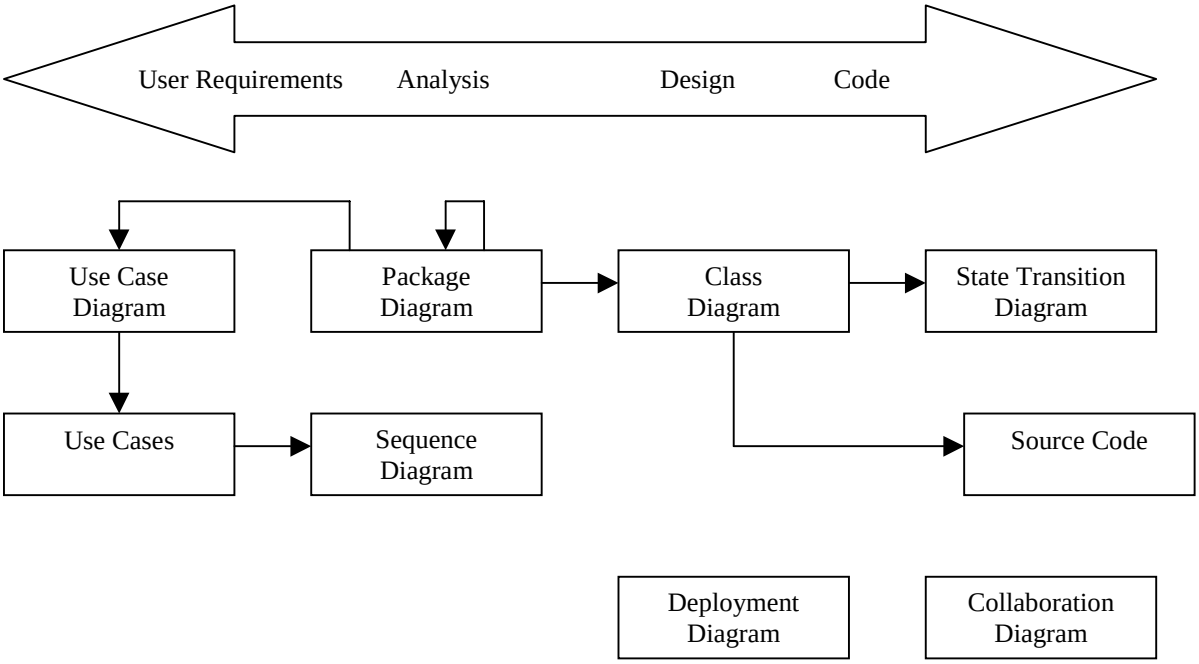


*Figure 2.11 Serial Development with UML modeling techniques*

## 2.4 Real-Time Object Modeling: ROOM

### 2.4.1 Introduction

In ROOM, there is a mix of advanced object oriented concepts and well-tested methods for real-time development. In ROOM you work with models (UML diagrams described earlier) that are executable and used to transform the requirements to an implementation. Models are translated to programs that can be executed in simulators on workstations or on target

machines with a real-time operating systems. ROOM is the base in UML-RT, the real-time extension of UML. The development process is often done in an iterative manner. You work with component-based models that are successively refined through a couple of iterations. In each step integration and verification of the models are done to ensure stability with the specification validated in an earlier stage. In this manner the system is tested thoroughly in each step of the development process

## 2.4.2 The Method

In ROOM you build a model with development tools like ObjecTime. ROOM has like traditional programming languages a syntax to express domain specific knowledge and design ideas. ROOM is object oriented with concepts chosen for event driven real time applications. As mentioned in 2.4.1 one can throughout the development process generate executable models.

The fundamental concepts of ROOM are objects, called actors that communicate with each other sending messages solely through interface ports defined by some protocol. To manage complexity, a structure can be hierarchically created, i.e. actors may contain other actors. Actors can also be created dynamically, i.e. actors are created and destroyed at run-time. This implies a system structure in constant change. In addition to structure an actor has a state-machine expressing its behaviour, i.e. its reactions to events. In addition to this, ROOM uses inheritance to describe variations of objects. Variations can be changes on structure, behaviour, as well as different protocols and data. To these, all graphical, concepts, ROOM uses traditional programming languages to explain detailed structure and behaviour. For example, high-level languages such as C or C++ are used to describe actions taken on events in a state-machine. This is powerful due to the use of a combination of graphical concepts and traditional high-level languages. Developers can use the same concepts for all activities, from analysis to implementation. The semantic gap common in traditional analysis implementation methods is reduced. From analysis and design to implementation there are only iterative refinements in contrast to the mappings between different representations used in traditional analysis-design-implementation methods.

## 2.4.3 Modeling Structure

A ROOM structure describes communication relationships between actors in a system. An actor is the primary concept in a structure. Actors are parallel objects that can exist and execute independently of other actors in the same environment. Actor implementations are encapsulated. Actors can be of two types: static or dynamic. In a static actor there is a one-to-one relationship between objects in the design and objects that actually execute at run-time. Dynamic actors are objects created and destroyed dynamically at run-time. Actors communicate with messages sent solely through ports. A port can be described as an opening in the encapsulation allowing the actor to communicate with other actors. A message contains a signal and a message body, i.e. an instance of a data class. A port is a specialized interface for an actor using a protocol. A protocol is a set of messages allowed to be sent through the port. Ports are instances of a protocol class allowing the reuse of interfaces or refinement through the inheritance mechanism.

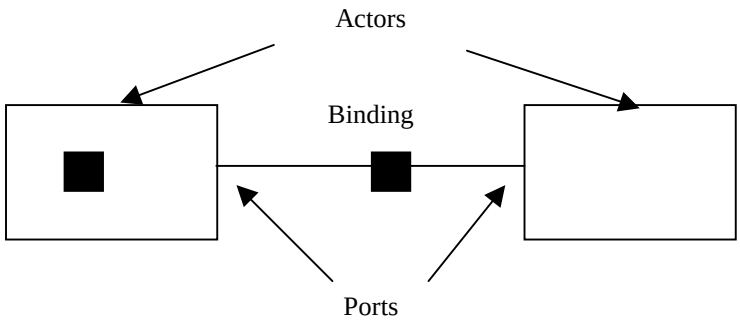


*Figure 2.12 Basic entities in the ROOM notation*

To mediate communication among actors ports are connected to each other with a construction called binding. A binding is like a communication channel that mediates the message sent by actors through its ports. A binding shows explicit communication paths between actors. Note that two actors can only communicate with each other if there exists a binding between them. The fact that actors communicate solely through ports instead of directly to eachother, makes it possible to view the actor as a black box independent of it's outside environment. This leads to actors that are inherently distributable and highly reusable in different design situations. Actors also take part in the concept of aggregation. Aggregation makes it possible to create arbitrary complex objects. In Figure 2.13, actor 4 and actor5 are leave actors, i.e. they do not contain any other actor. On the other hand instances of actor4 and actor5 are contained in the higher level actor3. This is displayed with a little symbol in the lower left of actor3. Actor3 and actor2 are contained in the even higher level actor actor1. ROOM allows the creation of arbitrary number of levels in the structure hierarchy. This is quite powerful when complex systems are to be modeled: it is easy to give an overall view of the system on a top-level structure, hiding the details in lower level hierarchies.

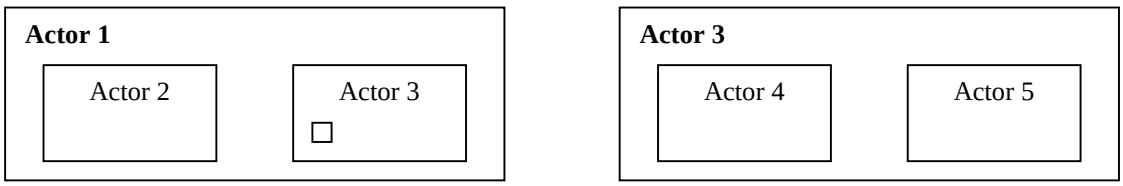


*Figure 2.13 Hierarchical Design with actors and subactors*

## 2.4.4 Modeling Behaviour

Behaviour expresses what actions an actor should take when messages arrive on its ports. ROOM uses extended hierarchical state machines, modified to manage object orientation and effective real-time implementations. In Figure 2.14 the principle of a state machine is explained. The state machine has three transitions: initialize, request and timeout, and two states. Idle and Handle. The initialize transition is fired when the actor starts up and are used to bring the actor to its operational state. The Watch example in Figure 2.4 would have been implemented in ROOM as seen in Figure 2.15.

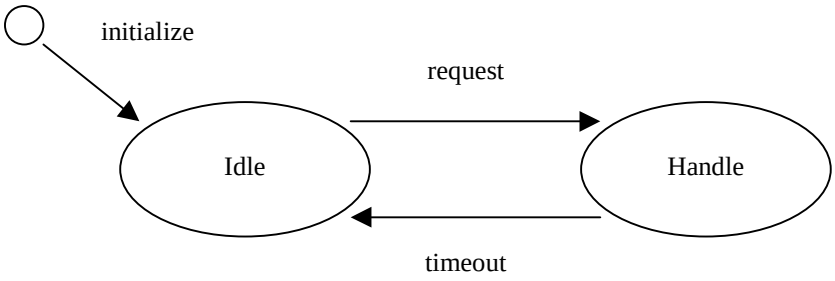


*Figure 2.14 Finite State Machine*

In this manner arbitrary complex behaviors can easily be hierarchically decomposed making it easy to express complexity and making state-machines easy to read and maintain.

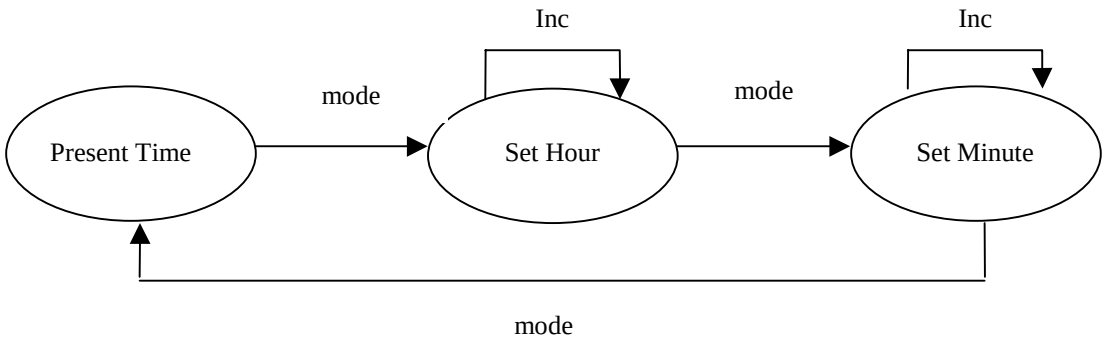


*Figure 2.15 Finite State Machine; Set Time*

As mentioned earlier the detailed behavior are the actions to be taken in the transitions between states. This is not expressed graphically.

Tools supporting the ROOM method, such as the ObjecTime toolset does combine the graphical representation with the text-based code in a graphical environment. The developer just click on a transition to bring up a text-editor where code easily can be written.

## 2.5 Designing UML Systems

“Designing a notation for use in object-oriented analysis and designs is not unlike designing a programming language”[8]. When designing with UML we divide the process into three different categories: architectural, mechanistic, and detailed design (Figure 2.16). The architectural design, designs software structures such as subsystems, packages and tasks, while the mechanistic design includes the design of mechanisms composed of classes working together to achieve common goals. Detailed design specifies the internal primitive data structures and algorithms within individual classes.

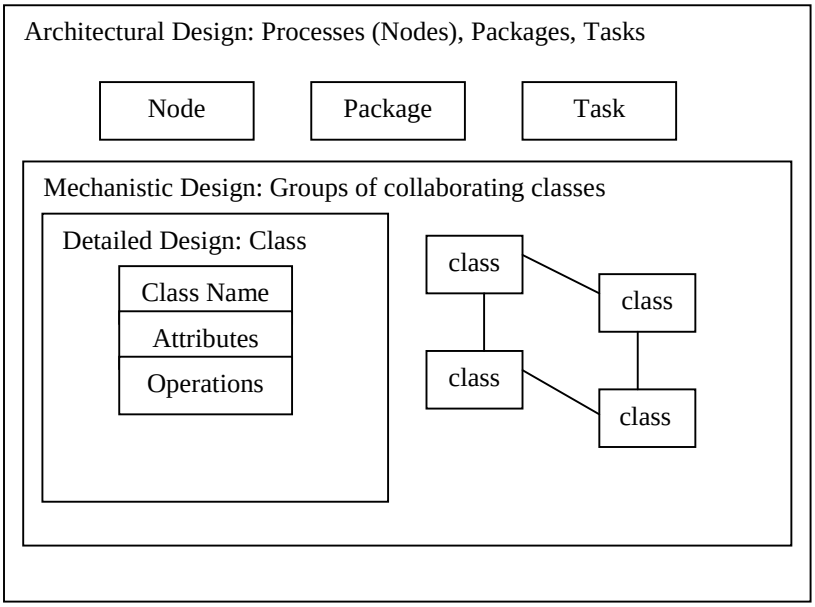


*Figure 2.16 Designing UML Systems*

## 2.5.1 Representing Physical Architecture in UML

In UML we represent physical architectures with deployment diagrams. The most important object in deployment diagrams is the node. A node can represent processors, sensor, and displays as seen in the ObjecTime implementation in Figure 4.7. Interconnections in the diagram represent physical interconnections were information (signals) is sent. Processor nodes may contain classes and objects, but can also be broken down into to subpackages and tasks. These subpackages ultimately contain objects and classes.

### 2.5.1.1 Distribution of Control in Systems

The complexity of a multinode system is the co-ordination of the separate subsystems. The two possible strategies are centralized and decentralized control. In the centralized control one node passes on signals (orders) to the other nodes, this is commonly known as the master-slave architecture.

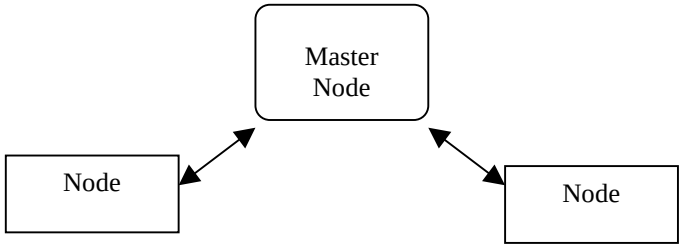


*Figure 2.17 Centralized control*

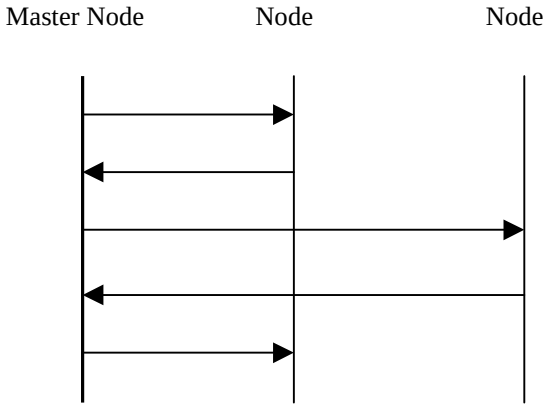


*Figure 2.18 Message Sequence Chart, Centralized control*

Centralized control assures simple modification and maintenance, up to a point. In a complex system it is sometimes wise to distribute the processing more evenly between the nodes. It is often easier to build a complex of less smarter systems combined with each other, instead of figuring out the complexity within one node. When the complexity gets to high, the workload may sometimes become to high for the single master node, and then there is no other option, than decentralizing the control.

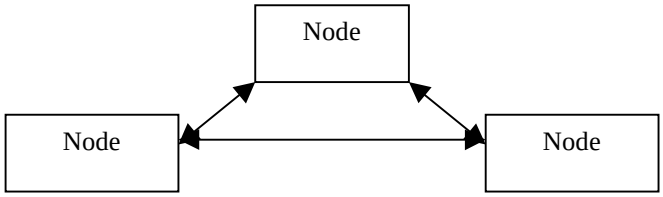


*Figure 2.19 Decentralized control*

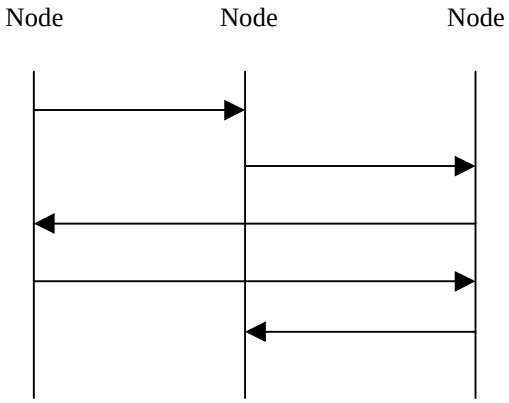


*Figure 2.20 Message Sequence Chart, Decentralized control*

### 2.5.1.2 Communication Infrastructure

When we use a multiprocessor (multinode) system, we require a communication path linking the processors to each other. Two separate but related issues are the bus topology and the protocol operating over the bus. The protocol defines what rules, formats (of signals) and procedures that the communicating objects have agreed on. There are different types of protocols, *Data driven* protocols are highly specialized and concentrate on the communicated information, this makes it simple and efficient; *Grammar driven* protocols concentrate more on what is said than the specific things themselves, the generality of the latter protocol makes then flexible but not as efficient as the previous.

## 2.5.2 Mechanistic design

Mechanistic design deals with how small sets of classes and objects collaborate to achieve common goals. A real-time system has multiple mechanisms operating concurrently. When the models and classes of the system have been defined, we use mechanistic design to add objects to facilitate their collaboration. An autopilot may use many sensors and actuators, if we have a common one-to-one association, the common solution is to have a pointer or reference in the client that sends a message to the server.

```
class Actuator {
   int value;
public:
   int gimme(void) {return value; };
   void set(int v)    { value = v; }; };
```

```
class Autopilot {
   Actuator *s;
public:
   void AutoPilot(Server *YourActuator) : s (YourActuator)
   {};
   // send message to Server object s
   void setIt(int a)    { s->set(a); }
   void IncIt(void)  {
      int a = s->gimme();
      s->set(++a);
   };
};
```

*Figure 2.21 Class example*

The class Autopilot uses pointers to locate its Actuator object and it sends the object messages like s-> set(a) and s->gimme() as above. Here we use a simple pointer since this is a one-to-one association, but the situation changes when the association is one to several. It is fully possible to collect the Actuator objects directly in the Autopilot class, with linked-lists or binary trees. But the drawbacks of such a solution are that the Autopilot class has to handle a task that is totally unrelated to the Autopilot task. If the handling of the collection is complex we will make the Autopilot class complex as well. [10] If the system has several classes with multivalued roles, we have to rewrite the behavior for each class. However during mechanistic design we can add a new class, a collection class. This solution enables us to change the type of collection easily. Remember that the analysis model identified the association and it's multiplicity, but the use of a separate collection class is a design decision.

## 2.5.3 Detailed Design

The fundamental unit of decomposition in object-oriented systems is the object. Some objects are fairly simple, while other objects needs detailed design to be implemented correctly or optimize the performance. The decisions in detailed design will concern, data structures, implementation of associations, algorithms, etc. This is nothing that is specific to an UML Real-Time system, but appears in all sorts of software development, so we won't go further into this subject.

# 3. ObjecTime- A brief overview (of the C-version)

## 3.1 Introduction

ObjecTime, which is developed by Rational, and is a software development tool based on UML design and the ROOM notation. It is well suited for designing and implementing real-time systems. A defining property of real-time systems is that they require responding to certain inputs within a predefined time interval. In a real-time system, one has to monitor and control the time resources very closely. Given the criticality of time in such systems, it is unfortunate that there are very few facilities available to improve the predictability of time utilization during software design and development. This situation is particularly problematic for concurrent systems in witch it is very difficult to rely on intuition about temporal properties.

There are two basic sets of techniques that address this issue:

- *Analytical techniques* are based on the construction of a formal mathematical model of the system and the use of mathematical algorithms to determine a system's performance characteristics.

- *Simulation techniques* are based on the construction of a computer-executable model of a system from which the performance characteristics are extracted by measurement.

Analytical techniques tend to be more rigorous but are usually quite complex and require mathematical expertise beyond the training of most software designers. Also, most of these techniques do not scale up to large systems so that they can only be applied either at a very high level of abstraction or they are applied at subsets of the system.

At present, the use of ObjecTime for performance modeling is based purely on simulation techniques. Several specialized features have been provided in ObjecTime especially for this purpose.

It is important to notice that ObjecTime is not intended to be used as a primary performance-modeling tool. Its current capabilities in that regard is generally not as extensive as found in most professional tools created expressly for that purpose. Instead, they provide a low-overhead facility for quick (but not necessarily extensive) insight into the performance properties of the system under construction. The overhead is low because, instead of building a separate model of the system, the ObjecTime design itself can be used for this purpose with little or no extra work required. This enables performance analysis to be tightly integrated with the analysis and design activities.

## 3.2 Why use ObjecTime?

ObjecTime itself is a powerful graphical modeling environment for the object-oriented design and simulation of real-time systems. It is easy to get to know and has a smooth user-interface that makes it ideal for rapid prototyping of distributed, event-driven systems using synchronous or asynchronous communication, and the development of efficient implementations for execution on the real-time platform. The Graphical User Interface, GUI, enables developers to work fast and with a good overhead of the system. "Good programmers

that understand ObjecTime are able to work three or four times faster than they could in any similar environment,.." [6]. The implementation is generated directly from the toolset, so that the implementation always stays in sync with the model.

ObjecTime Developer's graphical models provide a high-level, easy- to-understand environment for communicating requirements and design among developers, managers and customers. New team members can become familiar with designs faster by examining them at the higher levels then by dealing with details. [7]

Since ObjecTime is based on the UML/ROOM notation and thereby supports object oriented implementation, designing, both structure and behavior, is done graphically in a fashion that closely corresponds to the implementation in ObjecTime. (The GUI, graphical user interface, can be seen in Figure 3.2). ".. a well-designed architecture is not only one that simplifies construction of the initial system, but more importantly, one that easily accommodates changes forced by new system requirements"[9]. A comparison of ObjecTime and UML-RT notation is included in the table below. (Since there are no major differences between the UML and ROOM notations, and UML is the method that are to be used in forthcoming versions of ObjecTime/RationalRose the table shows the UML notations.)

| **ObjecTime term** | **Equivalent UML-RT term** |
|---|---|
| Actor | Capsule |
| Actor | Reference SubCapsule |
| Optional Actor | Optional SubCapsule |
| Imported Actor | Plug-in SubCapsule |
| Actor Structure | Capsule Structure |
| Actor Behavior | Capsule Behavior |
| Replication Factor | Multiplicity |
| Port | Port Role |
| Port Type | Protocol Role |
| Unconjugated Protocol | Protocol Base Role |
| Conjugated Protocol | Protocol Conjugated Role |
| Binding | Connector |
| Choice Point | Branch Point |
| Join Point | Chain State |
| Initial Point | Initial State |
| Data Class | Class |
| External Class | Class |
| MSC | Sequence Diagram |

*Figure 3.1 Comparison of ObjecTime and UML-RT notation*

The structure and behavior of a model in ObjecTime are described as a Finite State Machine (FSM) that shows in which order events are handled and what to do when different events occur. The structure and behavior editors are illustrated in Figure 3.2 and 3.3. Unfortunately, since the function call implementation only needed one actor in the structure editor and one FSM box in the behavior editor, there are no internal bindings between different actors or states. A larger example is illustrated in Appendix A.

The behavior of the states in the FSM is coded in detail in for example C. Most often this behavior is reactions to events that occur in the real-time environment, such as time-outs, in signals or the completion of other events. Those are fired in the transition between two states in the FSM. The three transition editors in Figure 3.2 shows some examples. To simplify coding ObjecTime has included a macro template from which macros can be dragged and

dropped into the developers code. The same goes for the ports and binding that are to be defined for each actor and transition.

The use of FSMs is especially well suited when dealing with real-time systems. ObjecTime takes care of the real-time entities such as semaphores and the real-time kernel automatically. The issue of setting priorities of events is preferably taken care of when designing the system since the FSM executes in a predefined order. Still, prioritized events can be modeled as special events in the actual states of the FSM.

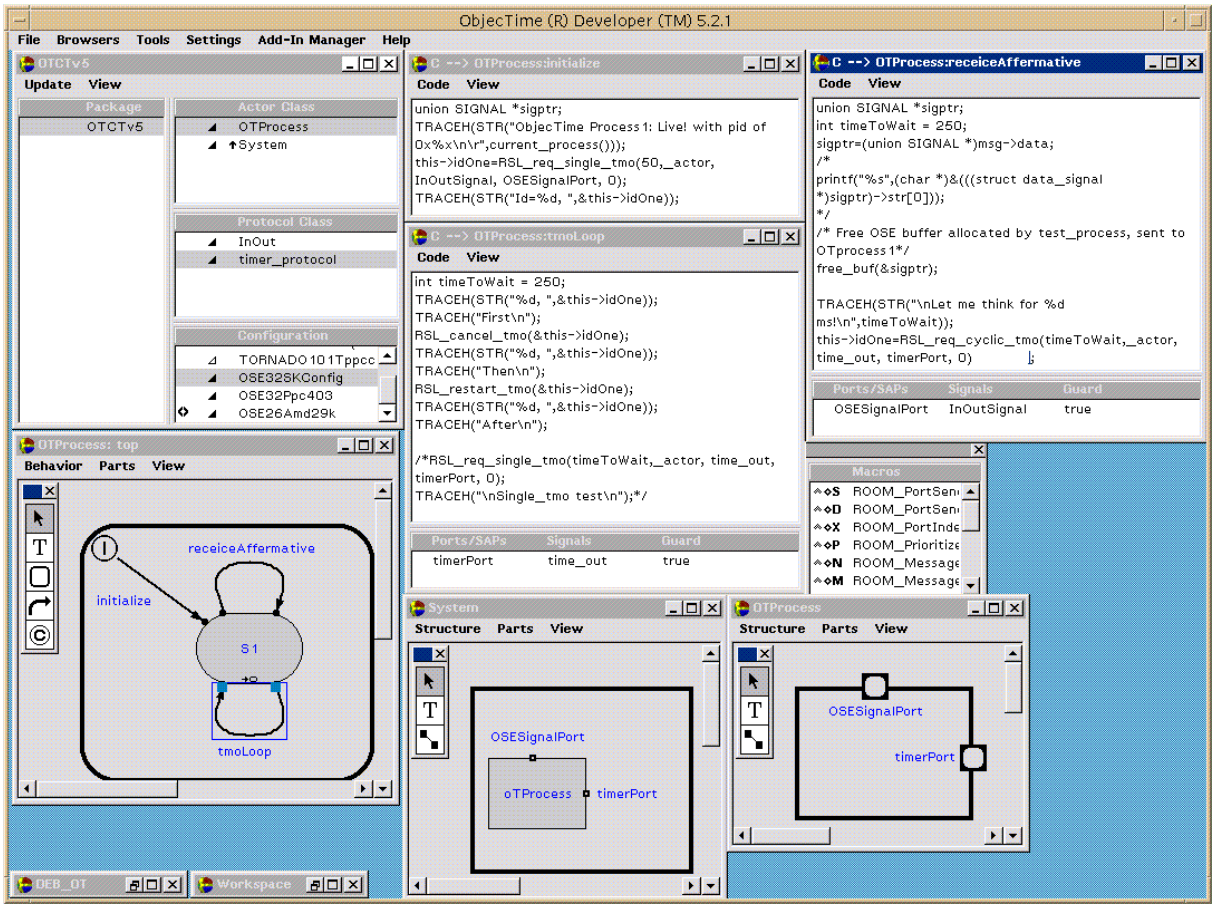


*Figure 3.2 ObjecTimes GUI*

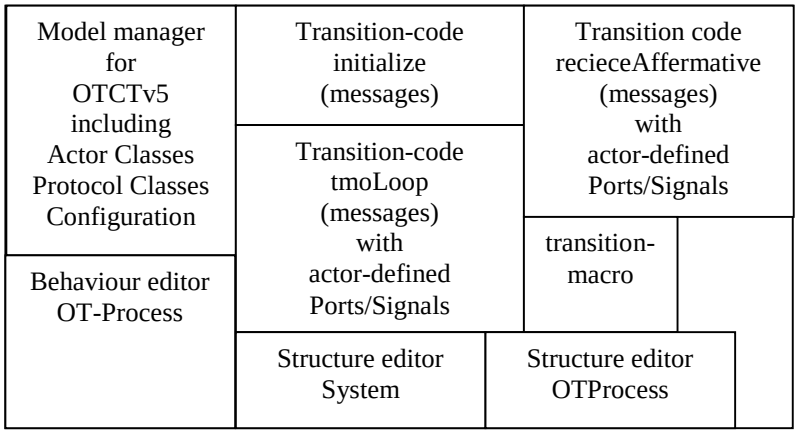


*Figure 3.3 ObjecTimes GUI explanation*

ObjecTime is a set of tools that spans critical sections of the software development procedure. Currently, ObjecTime provides tools for capturing requirements as well as designing, executing, debugging, and documenting designs. The tools are all integrated within the development environment. The debugging tool is especially user-friendly. The possibility to create Message Sequence Charts (MSCs) in the Simulation Services Library makes debugging easy and convenient. (The debug environment is illustrated in Figure 4.3).

Besides what has been mentioned above Rational, the company that provides ObjecTime, also supplies aid for documenting the work done in a stylish way. Documenting changes and development is necessary when working in projects, if software units are to be reused or changed by other team members. Rational also offers a version control tool, ClearCase. This tool is used to label different progress steps, record and report actions, history and milestones and assure integrity of software elements.

## 3.3 How to use ObjecTime?

When the design is done in UML, it is fairly simple to translate the design into an ObjecTime model. This is done via a ROOM translation as mentioned before. “Many, if not most, of the design errors in projects occur because the design was ambiguous or not properly translated into the software architecture and eventual code. ObjecTime eliminates these translation steps by creating a methodology and a tool that let a team seamlessly move from high-level design all the way down to code, with no translation steps." [7] Until now ObjecTime has been based mostly upon the ROOM notation, though in the coming version of ObjecTime (v 6.1) a change towards direct implementation of an UML (v 1.4) design will be carried out. This will not result in any major changes in notation since the ROOM notation well corresponds to the different components in an UML design.

To learn ObjecTime, working through a tutorial is recommended. This gives you a rapid, still thorough, insight in the basic ObjecTime concepts.

A software unit in ObjecTime is composed of two main parts: [7]

- The generated files and directories that represent the ROOM/UML model. This includes the structural components (actors, and so forth), as well as the behavioral detail of the model (the C code for transition actions, choice points,...)

- The ObjecTime Run-Time Service (RTS), which provides an abstract interface to the underlying OS services. The RTS provides the support for basic ROOM/UML concepts such as the message-based communication service. The generated code structure of the user model includes the hooks into the underlying ROOM/UML RTS.

### 3.3.1 Run-Time Services (RTS)

The RTS is simply the simulation tool in ObjecTime Developer. It is possible to run ObjecTime in two different versions of the RTS. The Simulation Services Library (Simulation RTS) and the C Target Services Library (C Target RTS) respectively. The Simulation RTS can be run on a variety of workstation-platforms while the C Target RTS runs on many operating systems and on workstations. It is also designed to be easily user-ported to additional target environments. The Simulation RTS is inherently C++ based and uses a C++ compiler to generate an executable, while the Target RTS is strictly C-based (in the C version

of ObjecTime). C++ is used because of the close connection between the Service RTS and the ObjecTime graphical toolset, which is C++, based.

The workflow that is recommended in taking a model from early prototyping to final production is shown below. The first step enables a full use of the simulation tools and debugging capabilities. The second step, which is reached after some necessary adjustments, is used to check the compilation. Here one can use C-source debuggers and C analysis tools to secure that the model runs as wanted. The ObjecTime feature Target Observability is a well-suited tool for verification of the behavior of the model.

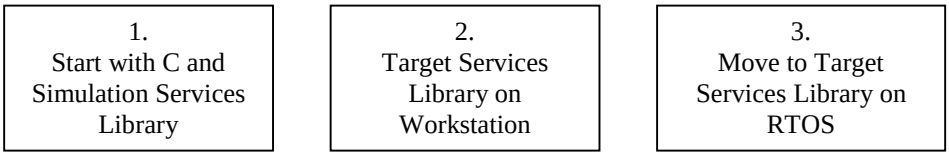


*Figure 3.4 Workflow in ObjecTime*

The final step is to compile the model for your platform and download and run the model on the target. Of course there are no systems that can be verified without feedback, and this process has to be run numerous of times with incremental improvements.

## 3.4 What does ObjecTime consist of?

### 3.4.1 Software Components

The software components in ObjecTime act as black boxes, similar to a Finite State Machine (FSM), against each other and have a protocol for interacting with other software components. This means that components can be reused in different models as long as the same interface is used.

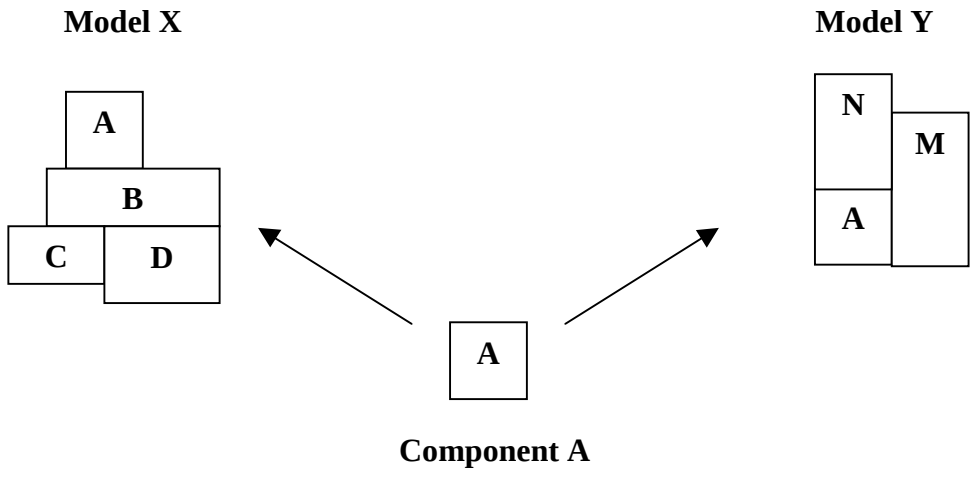


*Figure 3.5 Reusable software components*

Changes in one component of a design can be localized, taken care of and fitted into the environment among other components as long as the same interface is preserved. Using encapsulated components helps manage the complexity of a design, and increases the overall reliability of the software.

## 3.4.2 Real-world and virtual models

When designing it is often convenient to model real-world entities as software components. They model concrete entities that exist in the application domain. ObjecTime allows designers to create components that correspond directly to things in the real-world application domain as well as virtual or abstract entities. Virtual or abstract entities might be such as an algorithm or data structure (for example a queue), or a piece of software that performs a special function that doesn't exist in real life. An example of a real-world entity is a telephone while a phone-call can be modeled as a virtual entity.

To simplify reuse and protection of software components are stored in libraries. This reduces the overall effort required to develop new and maintain old software, and makes it easy to work in parallel in teams.

ObjecTime is made up of some corner stones. They define what can be done and how it is implemented. Since the core in ObjecTime is UML/ROOM, which originally was developed as aids for object-oriented software development, the components are much similar to the different parts of any object-oriented development language. (All entities named below, except data classes, are shown in Figure 3.2 as they appear in ObjecTime Developer.)

### 3.4.2.1 Actors

The basic structural components in ObjecTime are called actors and are the main units of design. In UML/ROOM there are standards specified for how the definitions of actors are made. This simplifies the reading of the generated code that ObjecTime evolves. A telephone example will be used to illustrate the different entities in ObjecTime. To start with an example of how actors form a model is shown.

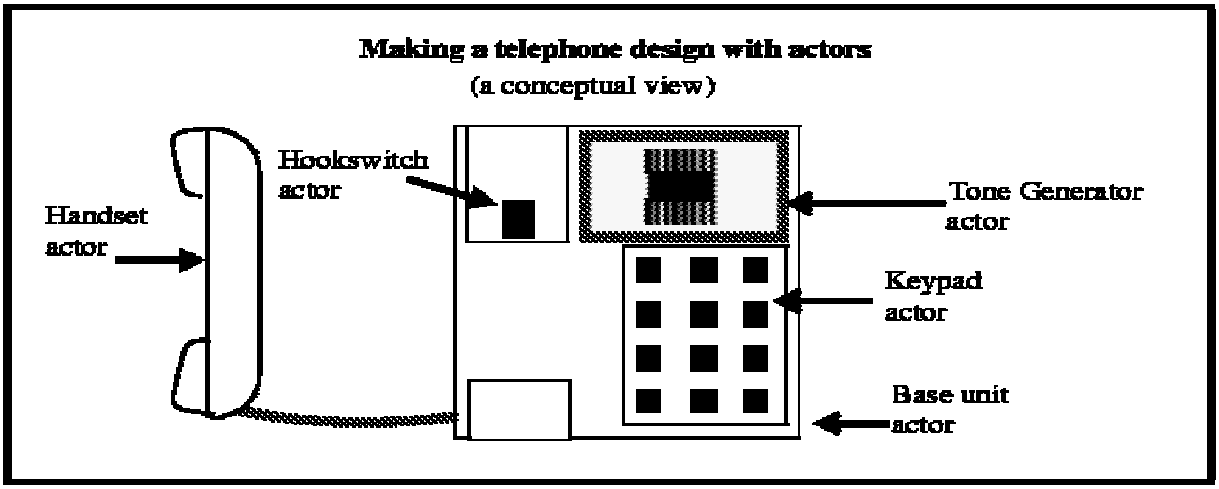


*Figure 3.6 Actors*

Actors have some parts that are specific, and five points can be listed namely: [5]

- Actors are potentially concurrent.
- Actors communicate by sending messages.
- Actors are encapsulated.

- Actors have a structure.
- Actors can have a behavior.

In the telephone example the points above can be:

- When a phone is in use, one can not use it for other calls.
- A handset actor communicates with the base unit actor.
- The base unit does not see what states the handset actor contains and vice versa.
- A telephone contains a handset, tone generator, keypad, hook-switch and base unit.
- The actions that have to take place to process an outgoing call.

If ObjecTime is going to be used in an appropriate way, designing multi-layered systems is mandatory. With layers containing more than six states, where a state is an instance of an actor, it is fairly hard to get a clear overview of the system. The example in Appendix A contains six actors in the first layer, with five of them containing subactors. This is indicated by the small rectangles in the lower left corner of these actors.

### 3.4.2.2 Messages

The actors communicate and interact with each other by sending messages from ports via bindings to other ports on other actors or to subactors within it. Those bindings are edited in the model manager and make the system easy to understand. For each actor you have to designate a set of messages to which it will respond. A complete design represents an overall system, and each actor is a part of the system. The overall behavior of the system is the sum of the behavior of all its interacting components.

It is not possible for an actor in the system to look inside other actors. By encapsulation, actors are not directly aware of other actors in the design: they see only their own interface through which they may communicate. Other actors send messages to request that an actor perform the functions for which it is responsible.

Message processing can be illustrated as the signaling between a keypad actor and a tone generator in the telephone example.

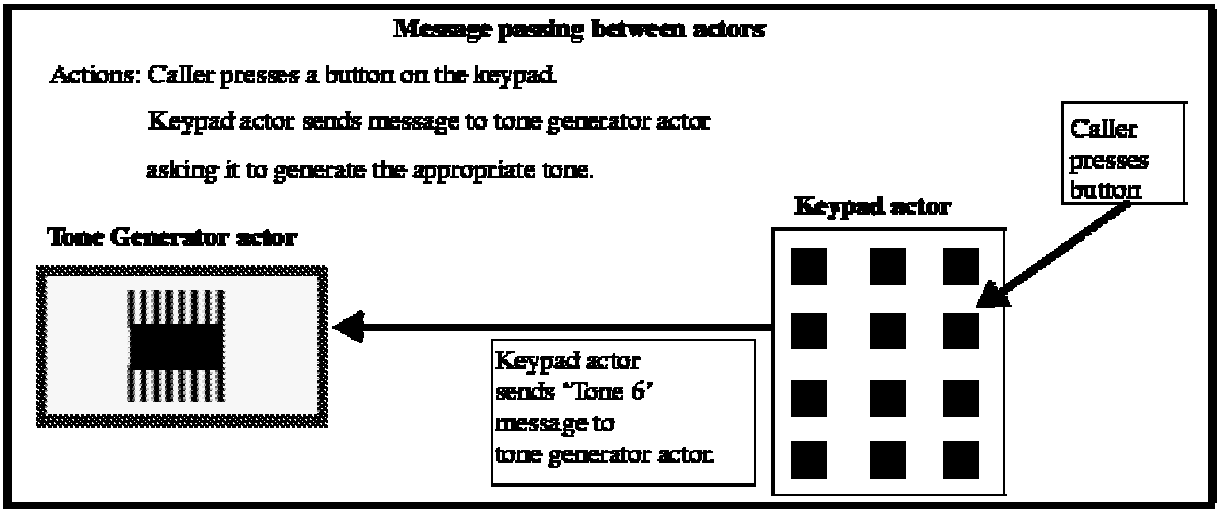


*Figure 3.7 Messages*

### 3.4.2.3 Actor Classes

Actor classes are the most basic component in an ObjecTime design. These specify the fundamentals for actors of a special type. All actors have a class specification witch serves as a skeleton for actors of the same type.

Two or more actors belonging to the same type used in a design are said to be references of the same actor class. Each actor class specifies the actor's structure and behavior, as well as the messages it can send and receive. Classes can also be stored in a library where they can be reused in other designs.

To communicate in a telephone network one has to have at least two telephone references to the telephone actor class.

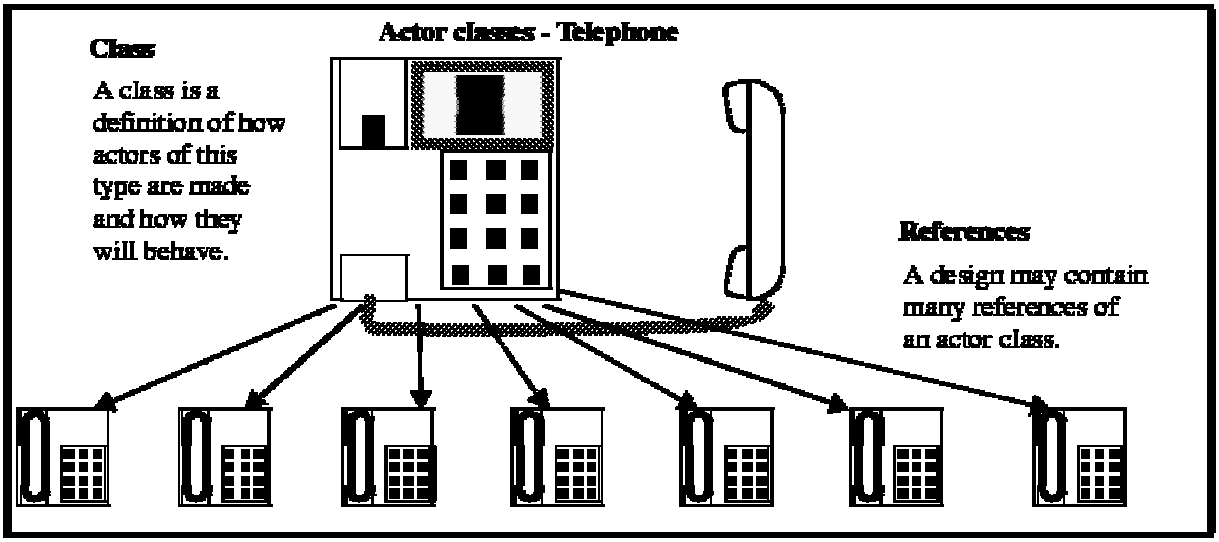


*Figure 3.8 Actor Classes*

### 3.4.2.4 Inheritance

Inheritance is the core of object oriented programming and is the biggest difference to procedure oriented programming, such as C or Pascal. C-code can be used to specify the behavior that the actors inherit.

Classes are organized in an inheritance hierarchy. Subclasses, not to be mixed up with substates, inherit various attributes from superclasses such as structure, behavior, and design documentation among others. Inheritance is an abstraction and reuse mechanism for system components. Different software designs often have parts in common. The telephone example is continued below with a picture of an inheritance situation.

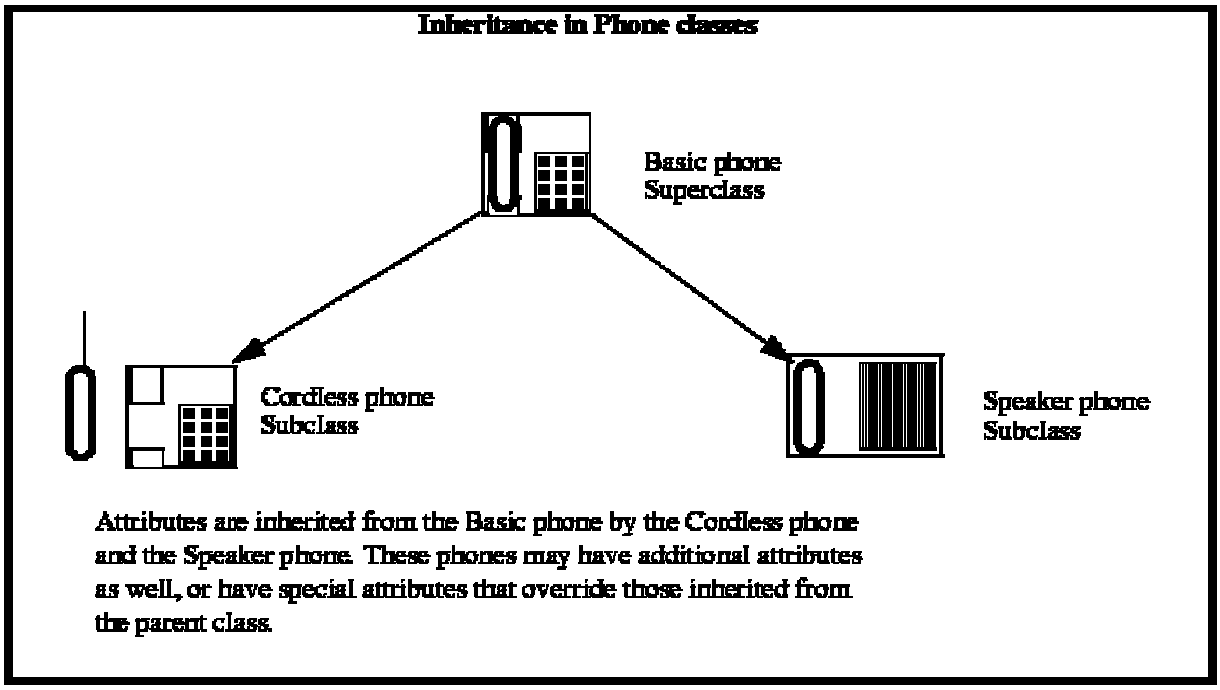


*Figure 3.9 Inheritance*

Inheritance allows you to factor the common areas of a component into a superclass, and then simply add specific behavior to subclasses that are specialized for the different designs.
This has two benefits:

- A common design is reused, instead of being copied and modified, or even re-invented.

- When bugs are discovered in the design, a single change fixes all affected designs. This makes software more reliable in the long run.

### 3.4.2.5 Actor Structure

To keep the complexity of software at an as low level as possible it is strongly recommended that you try to simplify your design. Actor structure is preferably organized in a manner where actor classes contain references to other actor classes.

This is a way of simplifying designs by allowing complex actors to be decomposed into simpler actors. Decomposition is an important principle in ObjecTime. It is a very useful way of dealing with software complexity. The decomposition is described by an actor’s structure. Structure is defined in the actor's class specification. The structure captures the communication and containment relationship among system components.

The actor structure is illustrated and further explained in Figure 3.9.

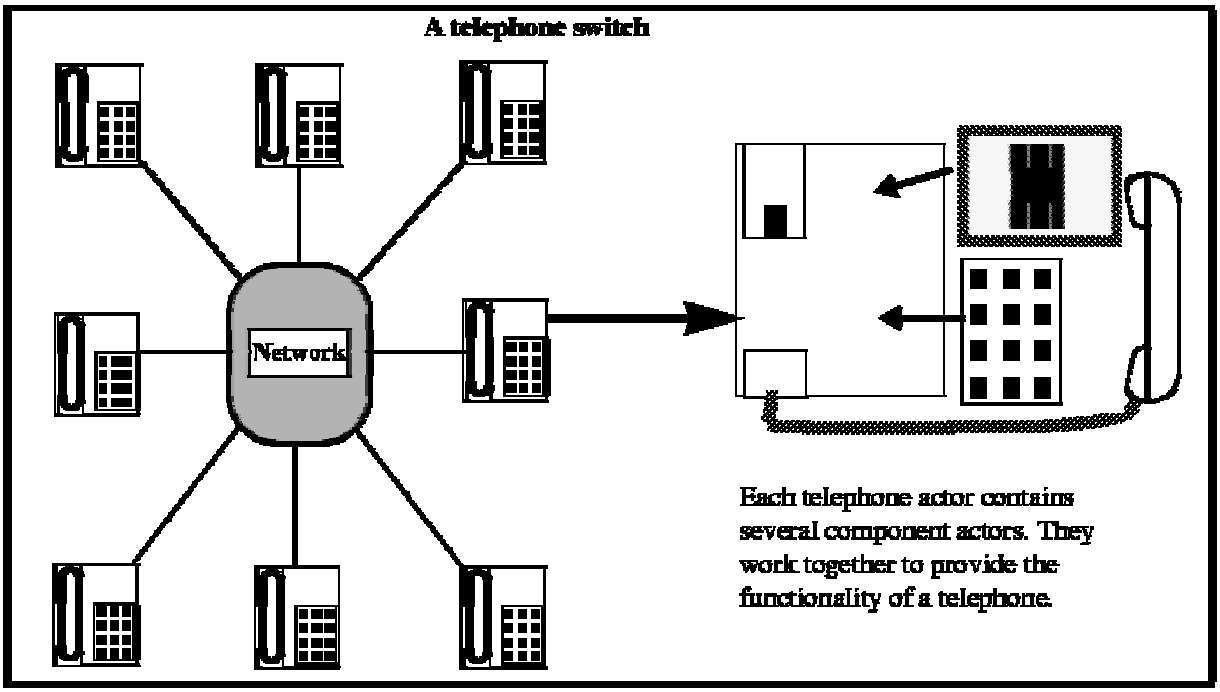


*Figure 3.10 Actor Structure*

### 3.4.2.6 Actor Behavior

In an FSM, components must have a way of reacting to system events. They must also have a way of communicating with other components in or outside the system.

Most actors have a behavior, which defines how the actor responds to different events. The behavior is specified via the FSM. When an actor receives a message, a transition may occur causing the FSM to perform a specific action and, possibly, move to a new state.

To make ObjecTime easy to read the number of states in each layer should be kept at a fairly low order. Up to six states gives developers, and other users, a fair chance to catch the full picture of the FSM. We, for example, tried to limit ourselves to a maximum of four states in each layer and there was never a problem to group states into one superstate with two or more sublayers when requested.

The actor’s behavior specifies actions to be performed when an action occurs. Actions can include sending message, performing computations, changing the value of a local variable, and accessing lower-layer services. The behavior in the different states is coded in C or C++. These behaviors should be fairly short, at maximum thirty rows since we still want the system to fully use the benefits of ObjecTime.

As described below an outgoing call demands, at a high level, at least a pickup of the handset, dialing a number, placing the call and a hang-up.

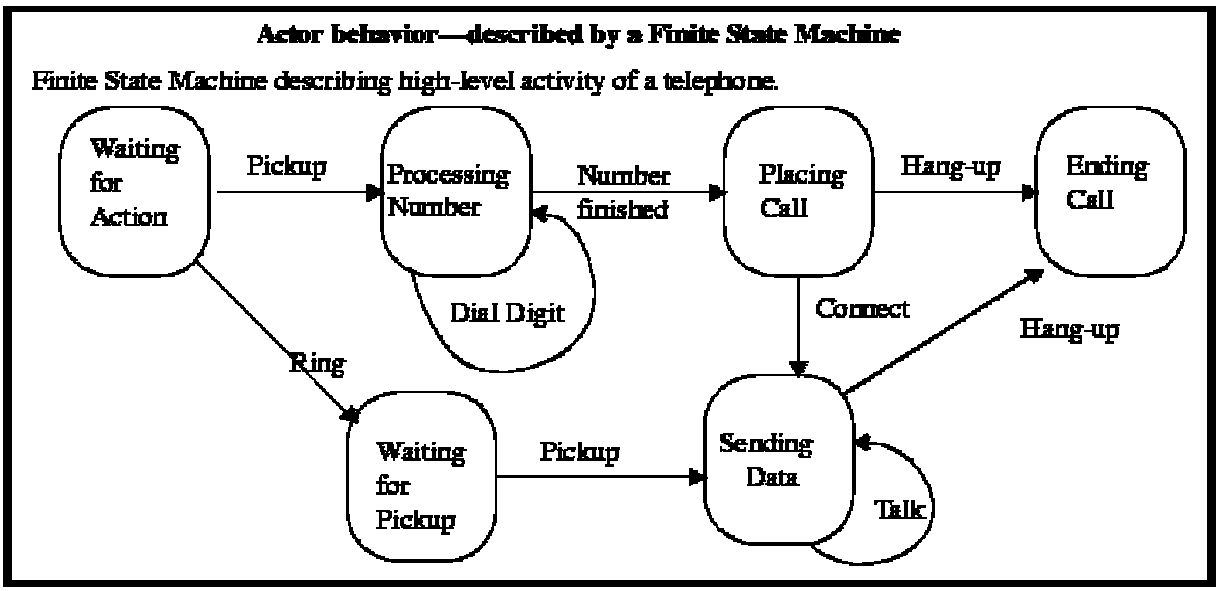


*Figure 3.11 Actor behavior*

### 3.4.2.7 Ports and Bindings

The interaction between actors is managed through ports and bindings. A port is a reference to a protocol class which defines the set of messages that a port is permitted to send and receive. The binding is a connection between the ports that channels the communication or interaction. Ports are attached externally to the interface of an actor or internally for communication within the actor. A binding has to be placed between the keypad actor and the tone generator. The ports, in this example, are only permitted to send respectively receive digits.

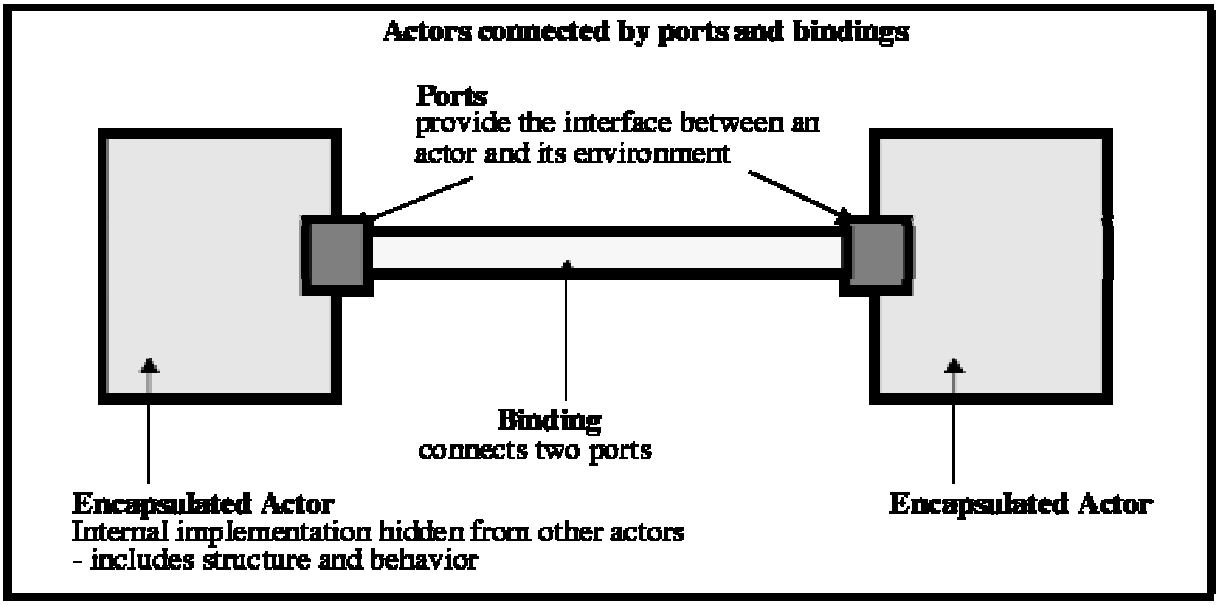


*Figure 3.12 Ports and Bindings*

### 3.4.2.8 Data Objects

Data objects are used in the actor's behavior. A data object is in some way similar to an actor. Considered to have a single, implicit port on its interface, it is passive and always executes within the thread of control of an actor.

A data class defines a data type and the valid operations on it. A variety of different base data types are supported which are based on language independent types. Data objects can also be sent and received by actors using messages. The data in a message is processed by the behavior of the receiving actor. Like actors and protocols, data objects are specified by classes.

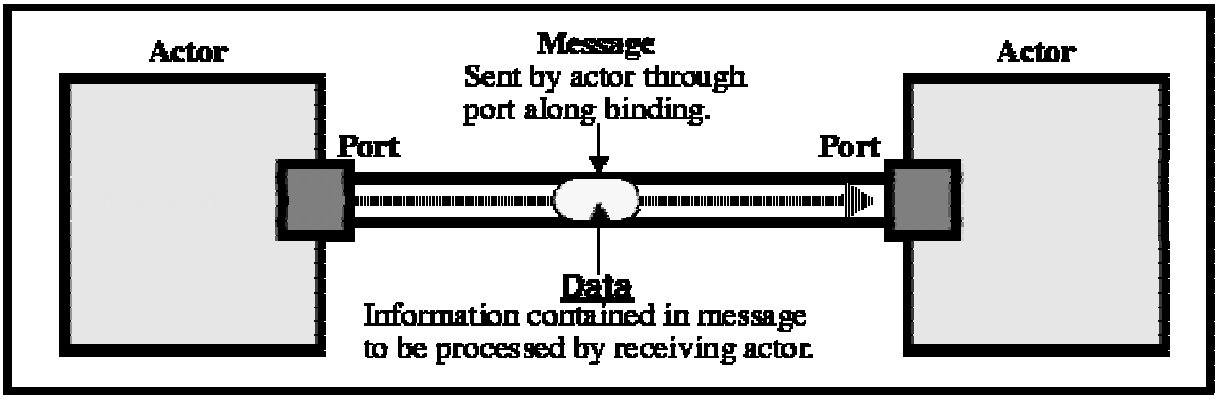


*Figure 3.13 Data Objects*

Note: Data classes are not supported with the C Target Service Library but are included to get the full picture of the possibilities in ObjecTime.

# 4. External communication in ObjecTime

## 4.1 Making ObjecTime communicate with external systems.

To meet the main objective of these thesis which, i.e. to enable external execution of existing real-time functions (a package called MPS at Ericsson Radio Systems, developed in the RTOS OSE-delta at ENEA) from within ObjecTime, part objectives were established. Those were first to manage the communication with test programs running on a simulator, PLS-Sim (HOST-testing), and then implement our solution, with code-generation, on a target processor AMD (TARGET-testing), as Figure 4.1 illustrates. As the next generation of radio-bases will be developed on PowerPCs, there was a possibility to implement our solution on a new PowerPC as well. This would have been very similar to the AMD-case, but as we reached this point the department's progress in developing an environment for the PowerPC wasn't completely finalized.

C was the main developing language at our department. This because C is a fairly low-level language, close to assembler-code. The C version of ObjecTime doesn’t include all the functionality that the C++ version does. One drawback of using the C version is that at simulation C-models are actually transformed into C++ executables running on top of a Simulation Services library. This is though, for most parts, invisible to users running models in the Simulation Services Library, but could be seen as a simulation of C++ code and not C as intended. The different MPS-functions are written in C on an OSE-delta platform.

OSE-delta is a real-time operating platform developed at ENEA. Our aim was to develop an interface in ObjecTime that managed the communication with HOST/TARGET so that the MPS-functions could be viewed as being actors within the ObjecTime model. ObjecTime was chosen because the UML-core and the possibility to evaluate the system through Target Observability. To accomplish our task a number of obstacles had to be solved.

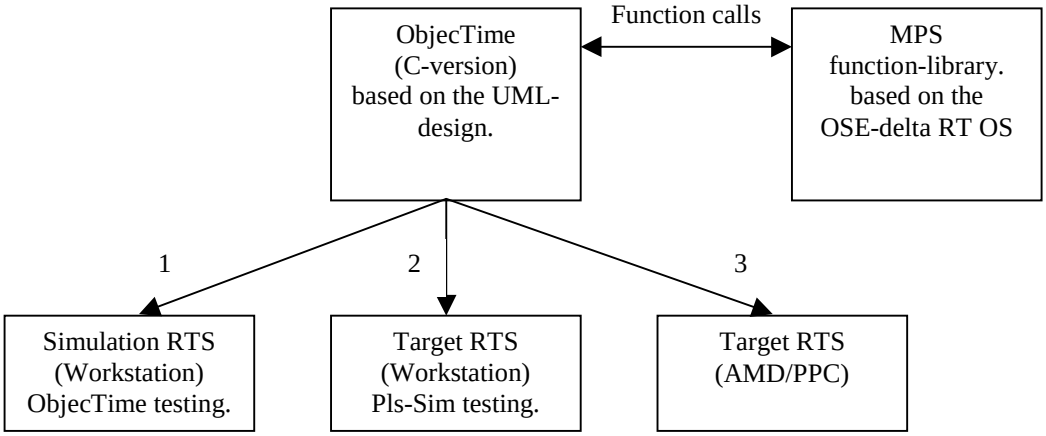


*Figure 4.1 System parts and relations*

## 4.1.1 Communication

To choose, and choose wisely, how to manage the communication to and from ObjecTime we looked at some different aspects beside the sole RT-aspect. Four major areas were discussed.

1. Compilation
2. Synchronization of processes (hunting)
3. Addressing (linking) and
4. Initiation.

### 4.1.1.1 Compilation

To be able to use the MPS functions, they had to be included in ObjecTime. Our aim was to unify the compilation and MPS function inclusion process. This was at first done in ObjecTime through writing the full function path in the configuration manager (can be seen in Figures 3.2 and 3.3) as inclusions for each ObjecTime-system. Since there often are up to twenty different functions that needs to be included and those search paths contain many layers we grouped all the functions needed in an archive including the MPS functions. This archive was stored in the ClearCase environment and included into ObjecTime as one link only. Still we had to make an extra compilation of the archive.

The link-solution was improved as the thesis progressed and now we have entirely skipped the archive for the MPS functions. Even the external compilation can now be skipped. This was one objective for choosing the way of "function implementation" that we did. The different "function implementation" alternatives will be evaluated below.

We would also like to do just one compilation of the entire system (ObjecTime, the HOST/TARGET program(s) and the external functions). Since it takes some time to compile ObjecTime as well as the external units we would like to see that parts of the compilation is optional, so that time is not wasted on compilation of already compiled processes. The compilation can be implemented in the ObjecTime make file, by making a load pearl script, ld.pl, file that "overrides" the old linking.

### 4.1.1.2 Synchronization of ObjecTime and MPS processes

Problems can occur with the priority of the processes as well as the with the port addresses, i.e. ObjecTime doesn't know what address to send and receive signals on. This can be a dilemma when we want the processes to automatically initiate the communication between each other. We had to solve this problem, with the processes starting in a proper order. As a first naive solution we wrote a macro in an init transition in ObjecTime that used a "while-loop" to wait for the external initialization-process to start. When started this process sent a message with it's priority and address to ObjecTime indicating that it now was all right for ObjecTime to start running. This solution was time-consuming and looked bad for a real-time solution. Finally, with some assistance from Tom Moore at Rational, a compilation pearl script was created that initiated the synchronization and set ObjecTime in it's first ready state, waiting for the external process or an event (internal or external) to occur firing a transition. Priority and address were stored in a linked list as mentioned below.

#### 4.1.1.3 Addressing

Since we were using processes running at different platforms, it happened to be that the addresses for ObjecTime and MPS had different sizes. Incoming messages from MPS to were not recognized in ObjecTime and vice versa. This problem was solved with two new queues that was created in ObjecTime, one for incoming messages and one for outgoing messages. The queues, two linked lists, inserted the function-calls in priority-order and modified the adresses to fit the recievers standard. Linked lists are by far the least time-consuming and flexible way of keeping track of those lists.

#### 4.1.1.4 Initiation

OSE-delta's and ObjecTime's default installations, had to be examined. As we have seen above, problems can occur if they have different default port-addresses, address-size etc. The solution was to write a batch-file for ObjecTime that initiated the receive-/send-queues, managed the compilation, and set the standards for those ports and address sizes. The file starts the HOST/TEST compilation, executes it, when done starts ObjecTime, makes a compilation of ObjecTime from within and finally closes it down.

### 4.1.2 Function implementation

Run-Time service systems can be equated to the so-called "system" services of traditional operating systems. For example, inter-process communication, file system access, or runtime exception handling are standard services provided by most operating systems. In ObjecTime such facilities are folded into the more general concepts of layering and services.

Layering handles the communication with other processes as well as internal communication while services takes care of function calls and the different operations executed during run-time (internally) [5]. In the "function implementation" phase the major issue was to ensure a sound real-time solution. The telecommunication area is an area that mostly runs under hard real-time constraints. Therefore we chose to implement every function as if it had to meet the hardest time constraints. This made this to be the largest problem to be solved. We had to investigate, test, implement and verify the different ways of solving the communication and function implementation of MPS functions into ObjecTime. Several alternatives were found possible. Some of them were fairly rapidly rejected while others were found to be almost as appropriate as our final solution. The different alternatives were as follow.

#### 4.1.2.1 Proxy

This is a simple communication method to implement. It is also a reasonably effective solution when there are no hard real-time restrictions to meet. Experts at Rational told us that this method was the most widely used way of dealing with the external communication. The time-aspect is difficult to fulfil because of the proxy, which acts as a bounce-point, that receives the signal, manages it and finally forwards it.

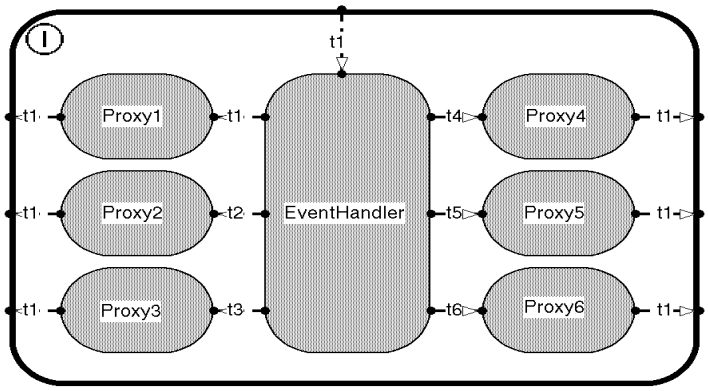


*Figure 4.2 Proxy Solution*

Since telecommunication features extremely hard real-time objectives this solution couldn't be used but are recommended for applications with soft time-constraints.

To implement the proxy-solution one has to create a signal-processing actor class, proxy, and then specify an instance of this class for each function call inside ObjecTime. In Figure 4.2 the EventHandler uses six instances of function calls that can be sent and handeled via six different proxies. The proxy communicates, with its MPS function, in the OSE-delta environment. The signals are sent to and from ObjecTime/OSE-delta via the proxy where they are modified to fit the receivers signal structure. This solution can be seen as two black boxes communicating with each other, with no possibility for Target Observability, that is, it is not possible for ObjecTime to see what signals that are sent in the OSE-delta environment and vice versa. Target Observability will be described in detail below. The debugging ca not be done in ObjecTime in any other way than with "printf". The Target Observability problem will be solved in the new version of ObjecTime (6.1).

#### 4.1.2.2 SAP/SPP

Service Access Point/ Service Provision Point can be seen as ports that send messages through different layers instead of ordinary ports that send within specific layers. The SAP's/SPP's are not graphically represented in the model as ordinary ports are. This solution might look as the natural communication alternative at a glance. ObjecTime appears as one layer and the external layer as an other. It is also the method recommended in the ObjecTime manual, but when evaluated it appears to be both complicated to implement, and fairly time-consuming to run. ObjecTime functions are used, but it is the adress string for these functions that has to be sent to OSE. In every call from ObjecTime through SAP/SPA, ObjecTime has to search for an address string in a fairly huge address register. If we have many calls/sec seeking for addresses, the solution becomes even more time consuming than the "proxy" solution.

#### 4.1.2.3 Inline coding

This is the straightforward "low-level" programming solution. One simply recodes all MPS-functions into replicas in ObjecTime. This is of course a very time-consuming method and demands a great deal of work. Every function has to be exactly imitated and with a fairly small insight in the different functions besides the amount of functions to be rewritten this is an alternative that is hard to implement.

#### 4.1.2.4 Port send (unbound)

This alternative looked almost similar to the SAP/SPP solution, with the drawback of finding the function adresses The difference is that the developer defines what the signal should contain, and that this solution is unbound. Unbound indicates that there are no threads drawn between ObjecTime and the external system. Instead port send uses a signal register when sending and receiving signals. Still ports are modeled in ObjecTime but there are no bindings connected to them. For example it might look as follows in a port that sends and recieves messages (from an ObjecTime point of view):

Incoming Register (Signal, Port, OTSignal, Actor)
Outgoing Register (OTSignal, Actor, SignalNo, Size, Address)

The hardest thing to manage is what size and address the outgoing register is supposed to use. The size of the signal might have different physical size in OSE-delta and ObjecTime and it appeared that OSE-delta and ObjecTime did not use the same format for similar definitions. (32 respectively 16 bytes for unsigned char). As this problem was harder to solve than first expected we did not continue on this track although it looked quite promising.

#### 4.1.2.5 Function call

The most flexible solution, which this chapter will focus on is the function call solution. All communication to OSE-delta is wrapped up as function calls. In these project the function names in MPS was used, since the developers at Ericsson are familiar with those. To ensure that trere are no possibilities for mix up RSL was added before the MPS call, i.e. “req_single_tmo” in MPS is referenced to as “RSL_req_single_tmo” The signals are sent out and received on an unbound port in ObjecTime. In the “RSL_...” definitions the MPS functions has to be called and the recieving, or sending, actors as well as ports and signals in ObjecTime has to be determined. At compilation ObjecTime will give an error message because it can not find a recipient. As a first solution we simply specified the external target (HOST or TARGET) the first thing at run time and wrote a macro to initialize the communication in the initialization file in ObjecTime. The final solution was to modify the make-files in ObjecTime. Now the error messages are not showed anymore and the communication initialization is performed automatically at compilation. An other problem with this solution is that one must know what actor and which port on that actual actor you want to send your signals to. The only ways to manage this is either to always know what actor and port adress you are sending to, which is most often solved with having one specific communication port, or to look for the address number in an Object database. Finally the addressing problem is solved in an elegant way where one specifies the port with name and actor and then ObjecTime finds the address number in the Object database automatically. If the signal is to be sent to the same port as it was sent from an &-pointer can be used for convenience.

This solution is more complicated to implement than the Proxy solution. Since the functions are called from ObjecTime in the same manner as it would have been in any other process the functions do not need to be included in any archive. Function calling is far more effective than most of the other alternatives when the call is time dependent, as for timer-functions calls. An other necessity is that ObjecTime uses the same timer as OSE-delta does, and this was achieved in this solution as well.

## 4.2 Target Observability

One of the major reasons for using ObjecTime as UML tool is that it features a Simulation tool that enables Target Observability. This tools enables debugging functions such as state-machine animation and break points, message tracing, message inclusions, single stepping and MSC plotting. [12] As to date PLS-Sim has been used for simulating the SW models.

During the progress of this thesis ENEA developed a new version if OSE-delta. This version better supported Target Observability. The problem had been that a function had been used with the same name in OSE-delta as in ObjecTime. With this feature one are able to see what signals are sent, when they are sent and from and to what ports the signaling occurs.
This is seen in the RTS control panel as in Figure 4.3.

As to date developers are able to plot Master Sequence Charts (MSC, exemplified in Figure 4.3 below), when running the model on a HOST target, to get a clear view of the Finite State Machine (FSM) signaling inside ObjecTime, but not for actors or procedures acting from outside ObjecTime. MSC’s are great to use when pinpointing race conditions that have occurred because of flaws in the design process or alternative execution scenarios.

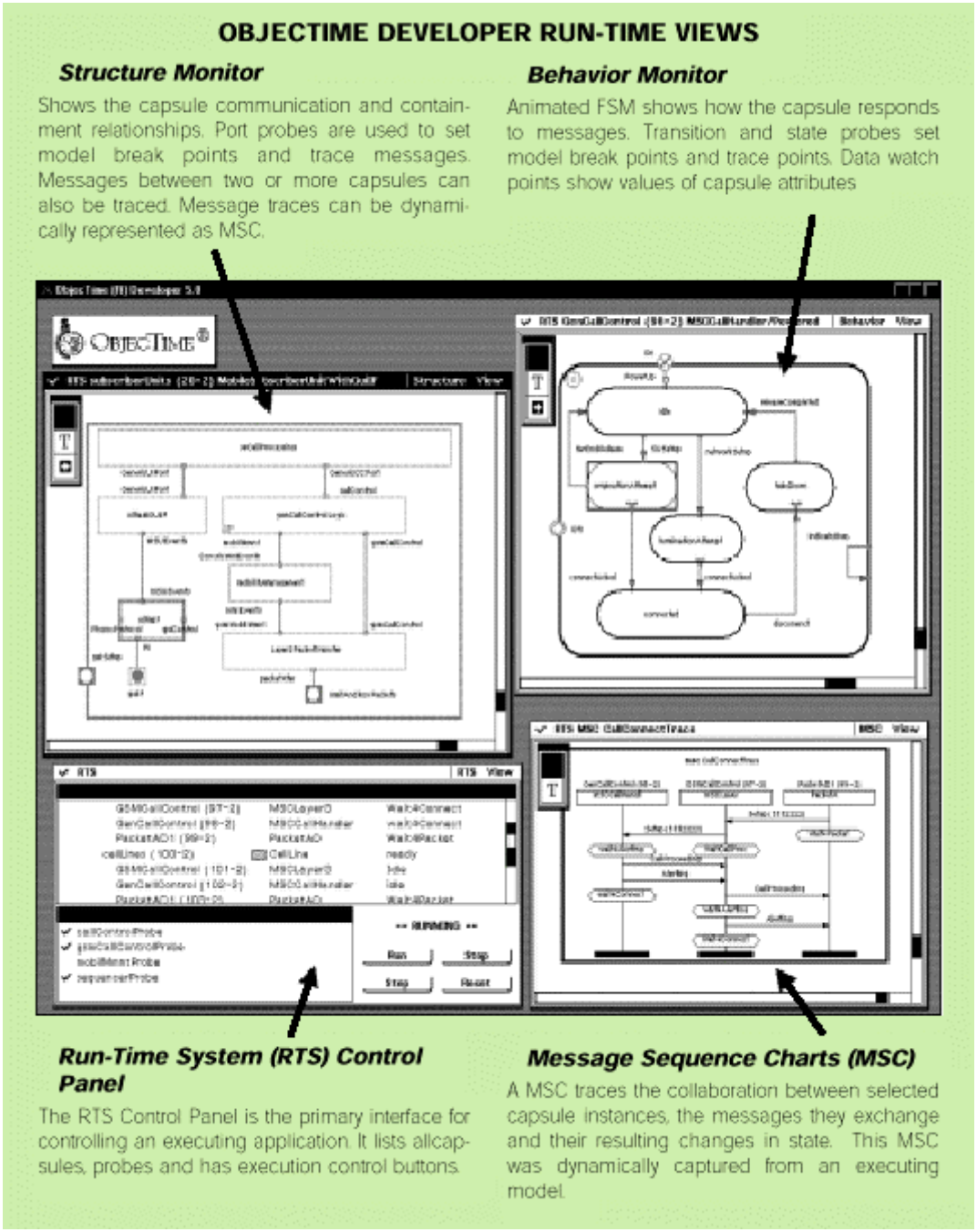


*Figure 4.3 ObjecTime Views*

In forthcoming versions of ObjecTime you should be able to automatically create FSMs from your MSC.

Problems can occur when porting ObjecTime to the target since one has to specify the physical port number (master port) for the communication. To date you are only able to use the Target Observability feature for simulated processes. Your system can be tested and verified, but still you can not use this nice feature on your target platform if you run the C version of ObjecTime. In the C++ version this is solved and in version 6.1 of ObjecTime this will be implemented in the C version as well.

## 4.3 Results

This thesis resulted in a model, illustrated in Figure 3.2 and 3.3, which can be run on an AMD Target. The model runs, is developed and can be evaluated in an ObjecTime environment. After the necessary start-up procedure a ”Request Single Time Out” is requested in the initialize-transition from the MPS function library. MPS returns a call when the time-out has run out. This triggers the recieceAffermative transition and the ”Request Single Time Out” is followed by a ”Request Cyclic Time Out”. One time-out is executed and afterwards the ObjecTime model triggers the tmoLoop. Here it signals ”Cancel Time Out” with simple call that includes the time out identity. In this transition the multiple time out is restarted as well. The function calls are described in detail below. This banal model runs on the Target processor and dumps a “printf” each time a signal is received from the OSE-delta environment.

In ObjecTime we had to implement a function call interface to the MPS functions that was to be used. This interface can be used as a template for further function implementations.

As the ObjecTime model runs smoothly on Target as well as in Simulation the probability for a full implementation of UML-design using ObjecTime (v 6.1, Rational Rose-RT) has hopefully increased.

As mentioned above four MPS time-out functions were implemented in ObjecTime, namely: RSL_req_single_tmo, RSL_req_cyclic_tmo, RSL_cancel_tmo and RSL_restart_tmo.

The function-calls looks like:

- RSL_req_single_tmo (int timeOutValue, RSLActorIndex _actor, RSLSignalIndex _signal RSLPortIndex _port, void* _data);

- RSL_req_cyclic_tmo (int timeOutvalue, RSLActorIndex _actor, RSLSignalIndex _signal, RSLPortIndex _port, void* _data);

- RSL_req_cyclic_tmo (mpsk_tmo_id *timeOutId);

- RSL_restart_tmo (mpsk_tmo_id *timeOutId);

Here ”timeOutValue” is the length of the time-out in milliseconds, ”_actor”, ”_signal” and ”_port” are the adress, signal and port to witch the time_out call shall send it’s respond, ”_data” is an extra void-pointer for future undefined purposes and ”*timeOutId” is a pointer to the actual timeout.

A header-file containing the definitions of the MPS-inclusions is stored in the ObjecTime-file: …/objectime/TargetRTS/include/ crsl_if.h. This header-file is target-independent, that is it goes for every instance of OSE-platform (HOST, AMD and PPC).

The function-calls are specified in a *.c file in ObjecTime. Here it is necessary to rewrite and store the *.c files locally for every OSE-target. The name of this file is RTThrSig.c and it is stored in …/objectime/TargetRTS/src/target/OSEXX/CRSL/. XX is the OSE-target and equals: 32SK for HOST, 26 for AMD respectively 32 for PPC. (SK stands for Soft Kernal).

Problems can occur when compiling ObjecTime if the pathes to the external root-libraries not are specified. It is recommended that at least the OSE, PLS and MPS root-pathes are defined in the …/objectime/TargetRTS/target/OSEXX/target.mk file specific for every OSE-target.

To verify the functionallity of the time-out function calls a small ObjecTime program was developed. It contained three transitions, as seen in Figure 3.2.

The first transition, initialize, calls a ”RSL_req_single_tmo” in 50 milliseconds. The pointer, ”this->idOne”, stores the timeout id. Figure 3.2 shows that the ”RSL_req_single_tmo” responds to the same actor as from it was sent. It sends a ”InOutSignal” on port ”OSESignalPort”. In the bottom of the transition code editor of the ”recieceAffermative” (!) it is indicated that this transition is triggered by a ”InOutSignal” on port ”OSESignalPort”. As this is received the ”recieveAffermative” transiton is fired. Now a ”RSL_req_cyclic_tmo” call is done in the same manner as described in the ”RSL_req_single_tmo” case. The respond from this call triggers the third transition, ”tmoLoop”. In this transition ”RSL_cancel_tmo” cancels the ”RSL_req_cyclic_tmo” and the ”RSL_restart_tmo” call restarts it again.

(The TRACEH calls are OSE calls aswell, and are used for debugging purposes.)

To think of, when developing in the ObjecTime/OSE environment, is that compiling has to be done in two, or optionally three, steps. First the ObjecTime TargetRTS environment has to be compiled. This is done in …/objectime/TargetRTS/src> with:

clearmake CONFIG=OSE32SKT.sparc-gnu-2.8.1 (HOST)
clearmake CONFIG=OSE26T.amd29k-gcc-2.5.2-921031 (AMD)
clearmake CONFIG=OSE32T.ppc403-Diab-4.0b.3 (PPC)

Here gnu, gcc and Diab are the compilers used for each TargetRTS.

If a SU is developed and stored in ClearCase this can be compiled seperately or togheter with the ObjecTime SU. If choosen to be compiled togheter one has to choose which SU to be compiled first. The easiest way is to make a patch in the mk.rules files for the ClearCase SU that starts the ObjecTime compilation. To do the other way around one has to make the patch in the ObjecTime-generated ld.pl file (pearl-script).

Two library inclusions have to be made in ObjecTime Developer configuration is now the following directories: …/pls2/pls_sw/ MPS/MPSK/EXPORT
and EXPORT

Besides the functions stdio.h, stdlib.h, string.h, mpsk_timeout.h, mpsk_traceh.h inclusions that includes MPS functionallity.

# 5. UML and ObjecTime at Ericsson Radio Systems

The work done in this report served as a test for ERA to see what possibilities UML design and ObjecTime implementation and verification could have in developing new software for radio bases in the new 2000 family and forthcoming generations.

At present the UML/ROOM notation and ObjecTime are being introduced at the department where this thesis was carried out (Control and Transmission, SW). Some pilot studies are done and developers are offered courses both internally at Ericsson (small tutorials) and externally at Rational. The courses focus on the developing process in ObjecTime and surprisingly not on the far more general methods of UML/ROOM design. Still, when using ObjecTime, it is inevitably to use the UML/ROOM notation. The pilots studies are done in case of time and serves as an introduction and evaluation on personal basis over what can be done in ObjecTime, and for what use it can be integrated in the personal development process of new software. As to date all pilot studies are restricted **not** to use the code generated in ObjecTime.

The single pilot study done completely in ObjecTime Developer, for the department, is a transmission model that simulates the signal flow in a TRU unit. The TRU is a unit that handles the interface between the Radio Base and the antenna. This model can be seen in Appendix A. The TRU model is far too complex to be described in this thesis and the Appendix is to be seen as an example of how a larger SU, Software Unit, might look. The example includes a screen dump A.1 of the TRU model and the system manager that shows all actor classes and data packages. Two of the actors are shown in detail in Figure A.2. The upper actor, the Lapd actor, has three ports on its fringe. All of them are end-ports since there are no subactors inside the Lapd actor. The behavior of the Lapd actor is shown in the upper right of Figure A.2. It contains two states, five bindings and a choice point, where the choice point simply acts as an if-statement. The boxes on the bottom of Figure A.2 show the DataLink actor. This actor contains two subactors, one of that contains subactors (illustrated by the small rectangles in the lower left corner of the data_link actor). The ports on the fringe of the DataLink actor are all relay ports, since they only distribute the signals down to the receiving subactor. There are no behavior needed for this actor and the behavior is as seen empty. The MSC of six selected actors are included as well.

The TRU model was not intended to be run on a target processor. The reason for the model was to look at what, where and when signals were sent. This was plotted in the MSC, Figure A.3.

(As a remark can be mentioned that the developer by his own words meant that he was at the beginning on of the most skeptic developers at the department but now one of the most enthusiastic.)

Other departments at ERA, e.g. WCDMA (Wideband Code Division Multiple Access), have ObjecTime running at full scale.

ObjecTime developing will probably be integrated in a broader scale when the new version of ObjecTime (v 6.1) becomes available. This version is a mixture of the existing versions of ObjecTime (v 5.2) and Rational Rose (v 98) and is also named Rational Rose-RT. It contains a more convenient window handling system, ability to extract your system from MSC’s,

better documentation tools etc. Ericsson holds a license of Rational new products and free support can be used and received continuously.

# 6. Summary

The object of this thesis is to describe UML-RT and evaluate to what degree UML-RT based design, verification and implementation (with ObjecTime) can be implemented into the existing development process at Ericsson Radio System. ObjecTime was chosen because of recommendations from a previous thesis. Ericsson Radio has previously used non-graphical methods during the testing, implementing and verification of Radio Base Stations. The new graphical programming languages has integrated the design and development process, making it possible to develop the system more accurate from the initial design, Ericsson Radio is interested in applying ObjecTime to the software development of Radio Bases. This thesis will describe both UML-RT and how ObjecTime could be integrated with the development of Radio Base systems.

We found that when integrating two different systems with each other, problems in the fringes between, in this example, ObjecTime and MPS were common, i.e. during compilation or in special cases when a certain kind of information is sent from one system to the other. In this particular case problems during compilation appeared due to overlapping names on different process calls, and information was interpreted wrong because the two systems used different standards on dataclasses. This was solved with changes in a pearl script and with changes with help from ENEA, the developer of OSE delta real-time OS. What may be annoying for the developer is that one still has to include, by hand, several functions from the old development tool, MPS, this means that when files are moved or renamed in MPS, changes also has to be done in ObjecTime. A future goal of the implementation will be to make MPS and ObjecTime independent from each other, with out destroying the compatibility of the two programs. The advantage of the current implementation, is that other systems may easily be connected with each other by using the same solution that we have developed. Even the common compilation that we implemented can be extended to include other systems that need to be compiled.

There are some minor disadvantages with the C-version of ObjecTime. One is that dynamic structures are not supported. This is an obstacle when adding new hardware components that must be represented as new actors in the software. Another problem is that inexperienced users easily tend to spread out small chunks of C-code on different places, such as in transitions, enter- and exit-code for states. This makes the generated code somewhat messy.

We conclude that UML-RT has strengths for developing real-time systems. Mostly because of the use of a uniform notation throughout the entire development process, from a design process that easily is translated into an executable model in ObjecTime where the code generation is done. The use of a standard, visual design process that is transformed directly into code means that the source code easily can be read, understood, reused and changed.

Since the UML-RT notation is based on independent components it is fairly easy to create different instances of the same component and then try the different alternatives to find an optimal solution.

The possible extensions that can be performed in the coming version of ObjecTime 6.1 are:

Include Target Visibility on the Target Platform, which as to date only can be done when simulating the system on your workstation.

The possibility to look into external systems to see what and when signals are sent. This makes it possible to draw Message Sequence Charts for the entire system and not only for the part implemented in ObjecTime.

Implementing our solution on the PPC target. This requires little work as we already has done a solution that can be run on an AMD target.

Other extensions are:

Making compilation optional. As it is now, the entire system is recompiled when changes are made. As to date one is able to compile separately but the different compilations have to be done in different libraries, which are nestled down deep in the library system.

# References


[1] *Overcoming the crisis in real-time software development*. Technical report, ObjecTime Limited, 1997. www.rational.com.

[2] *UML notation guide*. Technical report, Rational software Corporation, 1997.

[3] Paul Harmon. *UML, object modeling, and requirements specification*. Cutter Consortium, 1999. www.cutter.com.

[4] *UML summary*, Technical report, ObjecTime Limited, 1997. www.rational.com

[5] *ObjecTime Developer User Guide 5.2*. ObjecTime Limited, 1998. www.objectime.com.

[6] Mike Bienvenu. *Systems Design in ObjecTim*e. Technical report, Sparta Inc., 1995.

[7] *ObjecTime Developer C Language Guide 5.2*. ObjecTime Limited, 1998. www.objectime.com.

[8] Christian Demmer. *Unified Modeling Language vs. MWOOD-I*. Technical report, 1997.

[9] Bran Selic ObjecTime Limited, Jim Rumbaugh Rational Software Corporation. *Using UML for Modeling Complex Real-Time Systems*. Technical report, 1998.

[10] Bruce, Powel, Douglas. *Real Time UML*, Addison-Wesley, 1998.

[11] Garth Gullekson, Bran Selic, Paul T Ward. *Real-Time Object Oriented Modeling*, John Wiley and Sons. 1994.

[12] Andrew Lyons, *Developing and debugging Real-Time Software with ObjecTime Developer*. Real-Time Magazine 99-1.

# Appendix A

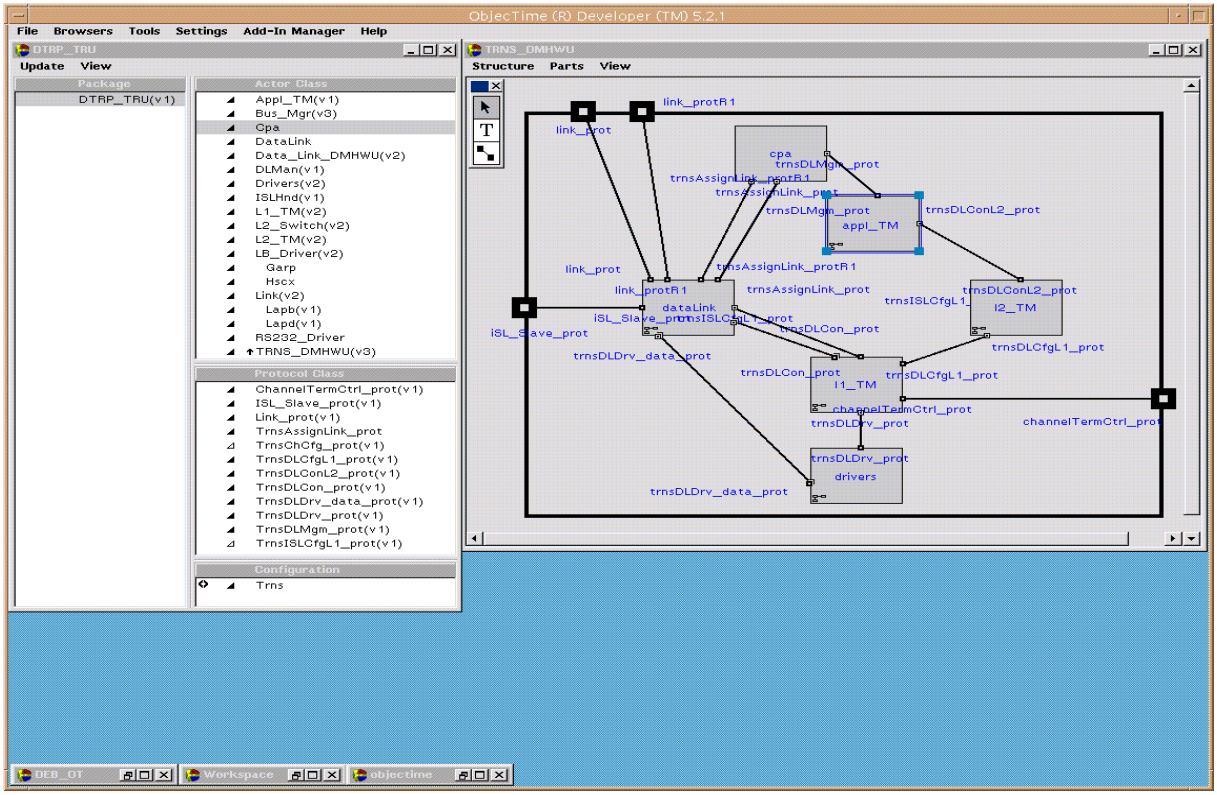


*figure A 1 TRU model*

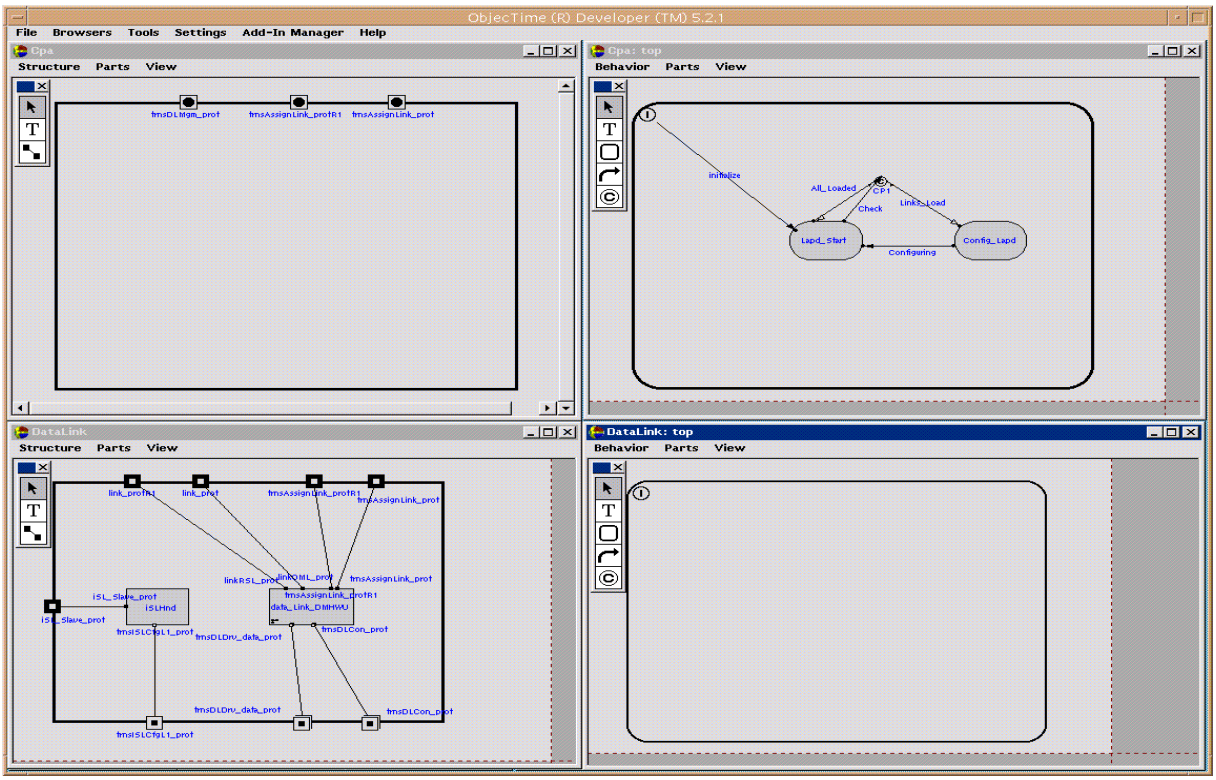


*figure A 2 Example of Actors and their Behaviuor in the TRU model*

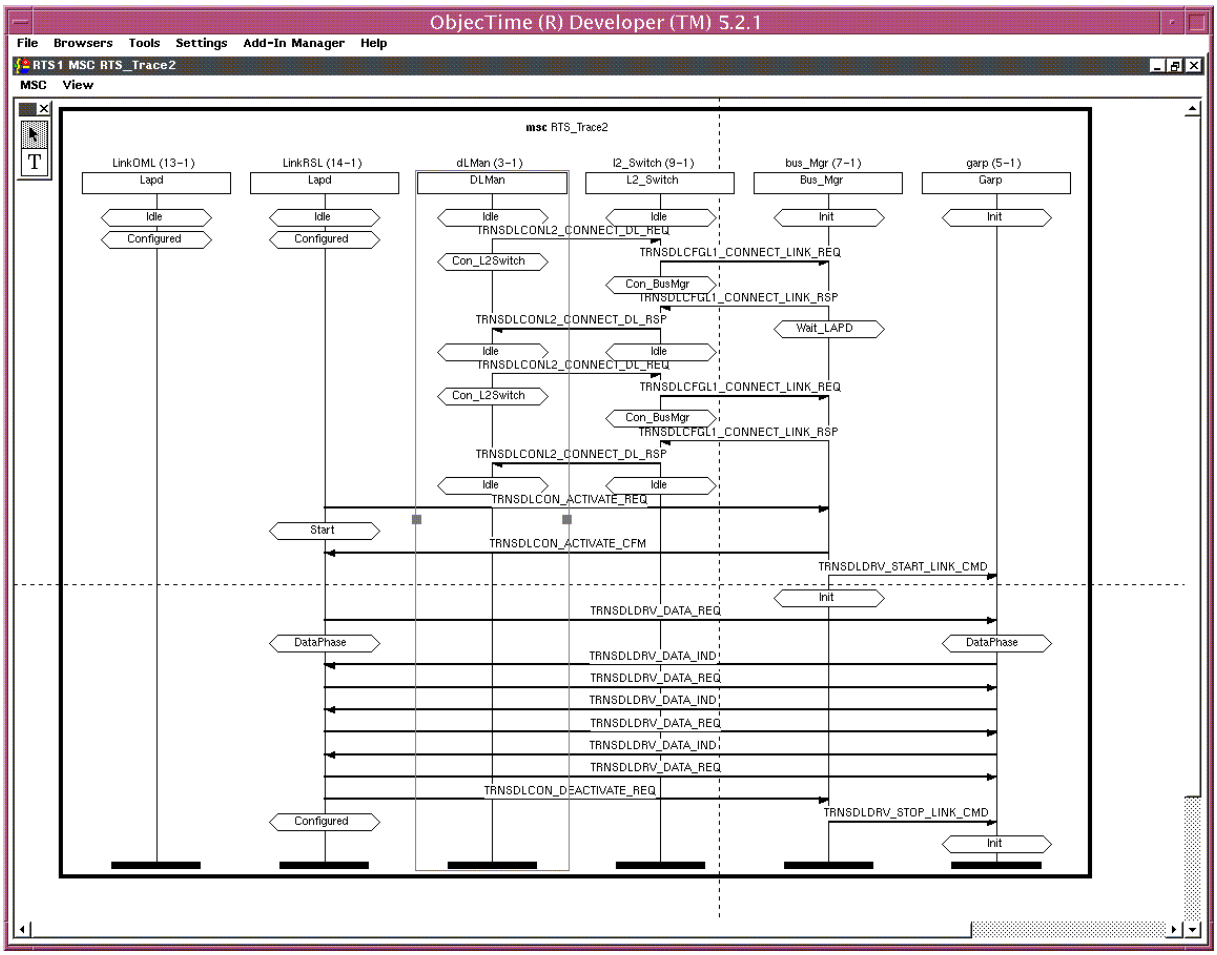


*figure A 3 MSC of selected actors of the TRU model*