Prepared (also subject responsible if other)		No.		
Magnus Antonsson, Pernilla Hansson		ERV/G-01:071 Uen		
Approved	Checked	Date	Rev	Reference
ERV/G/UE Anna Börjesson	ervrowe	2001-03-26	A	

Modeling of Real-Time Systems in UML with Rational Rose and Rose Real-Time based on RUP

Abstract

In software development, using UML for modeling of real-time systems is a fairly new area. There are a lot of different theories and reports in this area and UML is in a development phase when it comes to modeling of real-time systems.

This report provides a summary of how and why UML and UML-RT can be used for modeling of real-time systems. It also includes a summary of the advantages and disadvantages of Rational Rose and Rose-RT concerning modeling of real-time systems.

This report also provides modifications that can be done to RUP to make the process better suited for development of real-time systems. This includes among others the use of state machines to capture the concurrency within and among Use Cases.

Sammanfattning

Att använda UML för modellering av real-tidssystem är ett relativt nytt område. Det finns flera olika teorier och rapporter som tar upp detta och UML är i en utvecklingsfas när det kommer till modellering av realtids system.

Denna rapport ger en sammanfattning av hur och varför UML och UML-RT kan användas för att modellera real-tidssystem. Den inkluderar också en sammanställning av de fördelar och nackdelar som verktygen Rational Rose och Rose-RT har när det gäller modellering av real-tidssystem.

Denna rapport tillhandahåller också förändringar som kan göras på RUP för att göra processen bättre lämpad för modellering av real-tidssystem. Detta inkluderar bland annat användandet av tillståndsmaskiner för att fånga parallellitet inom och mellan Användningsfall.

Prepared (also subject responsible if other)		No.		
Magnus Antonsson, Pernilla Hansson		ERV/G-01:071 Uen		
Approved	Checked	Date	Rev	Reference
ERV/G/UE Anna Börjesson	ervrowe	2001-03-26	A	

Acknowledgements

We would like to thank all employees at Ericsson Mobile Data Design AB (ERV) for their support and help during this thesis work at their site in Gothenburg, Sweden.

We would also like to thank our supervisor Jan Jonsson, Department of Computer Engineering, Chalmers University of Technology, Gothenburg, Sweden.

Prepared (also subject responsible if other)		No.		
Magnus Antonsson, Pernilla Hansson		ERV/G-01:071 Uen		
Approved	Checked	Date	Rev	Reference
ERV/G/UE Anna Börjesson	ervrowe	2001-03-26	A	

Prepared (also subject responsible if other)		No.		
Magnus Antonsson, Pernilla Hansson		ERV/G-01:071 Uen		
Approved	Checked	Date	Rev	Reference
ERV/G/UE Anna Börjesson	ervrowe	2001-03-26	A	

Table of Contents

1 INTRODUCTION

Ericsson Mobile Data Design AB (ERV) is moving from a structured modeling[1] approach to an object-oriented modeling process when designing software systems. This includes the introduction of UML (Unified Modeling Language) and RUP (Rational Unified Process). When using RUP, software-development projects are guided through the different phases of the development process.

RUP, however, has some limitations when it comes to modeling of real-time systems. In the design phase, for example, the projects have to choose between "design of real-time components" (using Rational Rose-RT[2] or some other tool) or "design of non real-time components" (e.g. using Rational Rose). But RUP supports no guidelines on how to make this choice, nor does it state the benefits and the consequences of the two different design methods.

1.1 PURPOSE

The purpose of this thesis work is to investigate how real-time systems efficiently can be modeled using UML[3], with Rational Rose and with Rational Rose-RT based on RUP. This includes an examination of the advantages and disadvantages of Capsules and Protocols, concepts in UML-RT (introduced in chapter 2.3) used by Rational Rose-RT.

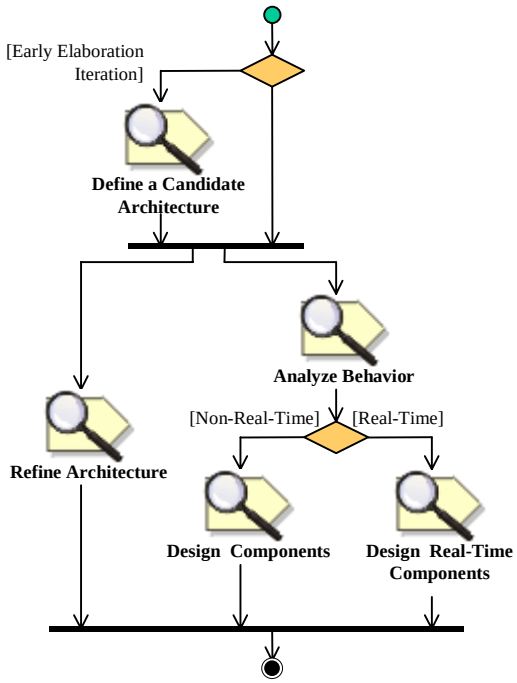


Figure 1. The analysis and design process used in the GSN-projects today.

[1] Structured modeling focuses on functions/methods instead of objects.
[2] Rational Rose for Real-Time.
[3] When referring to the UML standard in this document, we intend version 1.3.

It further includes an illustration of important issues that should be considered during modeling of real-time systems and *when*, during the development process, these issues should be taken under consideration.

The goal is to find accurate and unambiguous modeling methods that can be used at ERV when developing systems with real-time requirements and to present guidelines to help ERV in the choice between the activities "Design Components" and "Design Real-Time Components" defined in RUP (see Figure 1).

1.2 SCOPE AND LIMITATIONS

This report is not intended to serve as a tutorial for modeling in Rational Rose and Rose-RT. It will, however, show important possibilities and scarcities of the above mentioned tools when it comes to modeling of real-time requirements and systems with real-time properties.

The thesis work focuses on *modeling methodology*. We will neither consider the ability of Rational Rose or Rose-RT to support and integrate with other software development tools, nor other "technical" aspects as the support for round-trip[4], reverse engineering[5], programming languages, API's[6] and so on. Code generation and test will only be discussed superficially. We will focus on the requirement, analysis and design activities in the modeling process. No detailed description of RUP will be provided.

The system used as an evaluation and work model during this thesis work is based on a support node from the GPRS[7] standard, namely the GGSN[8]. Modifications, simplifications and additions have been made to the original GGSN to fulfill our demands on an evaluation/work model. The design of the GGSN in this thesis work is not intended to be used as a base for ERV in the development of there GGSN.

It is assumed that the reader is familiar with real-time systems and the problems involved in the design of real-time systems. The reader should also have basic knowledge of UML and concepts relating to object-oriented design.

1.3 OUTLINE

This report is the result of our studies on how real-time requirements can be modeled with UML.

[4] The capability to incorporate modifications, done in the code, to the model
[5] The capability to generate models from existing code
[6] Application Programming Interface
[7] General Packet Radio Service
[8] Gateway GPRS Support Node

The first chapter gives an introduction to the thesis work. Here the scoop and the limitations are stated.

Chapter 2, The Basics, is divided into three parts. The first part defines real-time properties that are significant to real-time systems. The second part gives a brief introduction to UML and the UML diagrams that are used in this report. Readers already familiar with the UML diagrams can skip this part. The last part introduces Capsules, Ports and Protocols, concepts added to UML to form UML-RT.

Chapter 3 presents the system that is used during our modeling. It includes a brief description of the GPRS-system (chapter 3.1) and our GGSN node (chapter 3.2).

Chapter 4, Modeling, shows how UML and UML-RT can be used to model systems with real-time requirements and how Rational Rose and Rose-RT support these methods. If you want to find ways to change your modeling guidelines and discover what opportunities UML and UML-RT can offer for modeling of real-time system, read this chapter!

Chapter 5 describes how RUP can be improved with new or modified artifacts and guards in the requirements, analysis, and design workflows. The tracks in the analysis and design workflow have been extended with new guards and guidelines on how to choose between the tracks and a new track, adjusted after the abilities and limitations of UML-RT and Rational Rose-RT, have been added. If you are interested in development processes, especially RUP, you should read this chapter!

Chapter 6 summarizes our work and conclusions and Chapter 7 presents our experiences from this work and different issues that are still unsolved and how ERV can move forward towards new knowledge.

2 THE BASICS

This chapter provides a brief introduction to real-time systems, different UML diagrams and the concepts of Capsules, Protocols and Ports introduced in UML-RT.

2.1 REAL-TIME ASPECTS

It is hard to find an unambiguous definition of real-time systems among today's technical literature. Most authors, however, seem to agree that real-time systems can be characterized by the following properties:

- Time constraints
- Concurrency
- Interaction
- Non-functional requirements
- Distribution
- Implementation constraints

2.1.1 Time constraints

Time is a critical aspect of real-time systems. The system performance must both be correct *and* timely. Time constraints arise from the laws of nature, limitations in hardware components, mathematical theory (e.g. the minimal speed of sampling) and artificial requirements (e.g. harmonicity and exclusion) [8] [9].

Time constraints are commonly divided into hard and soft time constraints. In systems with *hard* time constraints, missing a single deadline will constitute an unacceptable failure. On the other hand, in systems with *soft* time constraints, missing a deadline occasionally is acceptable.

Important modeling concerns of timeliness are modeling execution time, deadlines, arrival patterns, synchronization patterns, and time sources [19].

2.1.2 Concurrency

Concurrency is a feature of the real world. At any given time, multiple simultaneous activities can take place. Embedded in this world are real-time systems that often must react to these activities. This requires the real-time systems to be able to respond concurrently as well.

One definition of a concurrent system is that it is a system with two or more simultaneous threads of control (processes/tasks) that dynamically depend on each other in order to fulfill their individual objectives [6]. A system can be pseudo-concurrent, which means that

the different threads of control execute on the same processor. The real-time operating system schedules the threads to share the processor in a way that gives the impression of concurrency. In a "truly" concurrent system each thread of control executes on its own processor.

The definitions of tasks, threads and processes alter in different literature. When we talk about threads, or tasks, we mean threads that can coexist within an enclosing data space, but with weak encapsulation. When the thread has a stronger encapsulation and use different data address space and must resort to expensive messaging to communicate data among themselves, we will refer to them as processes.

Modeling issues concerning concurrency are scheduling of the threads of control, arrival patterns of incoming events, synchronization of threads, and the control of access to shared resources.

2.1.3 Interaction

Many difficulties in dealing with concurrent systems arise in the interaction between threads of control. The basic forms of interaction are *synchronization* and *communication*.

Synchronization does not include any exchange of information. Its purpose is, among others, to ensure proper interaction, to adjust the execution timing of threads in accordance to other threads and to avoid deadlocks, starvation and incorrect shared access to resources. Concepts used for synchronization of threads are among others semaphores, monitors and critical regions [16]. Scheduling is also a kind of means for synchronizing threads.

Communication on the other hand includes some kind of information exchange between different threads of control. The communication can be either synchronous or asynchronous.

Some of the most common communication mechanisms are [16]:

- Operation calls: An ordinary operation call is a synchronous communication technique. The object that originates the communication calls an operation on another object and then waits for the operation to finish and to return a result before it proceeds.
- Rendezvous: For this synchronous communication mechanism, a number of rendezvous points are specified in the execution of two threads. When the first thread reaches a rendezvous point it stops and waits for the second thread to reach the corresponding point. When they are both at the rendezvous point they exchange information and then continue to execute in parallel.
- Message queues/Mailboxes: An asynchronous communication technique where the sender places a message in a defined

mailbox and then continues executing. The receiver checks the mailbox when for incoming messages whenever it is ready to handle them.

- Shared memory: A technique by which a block of memory is reserved for communication, where a number of objects can write and read information. The block must be guarded by a synchronization technique so that object cannot read and/or write the information at the same time.
- Remote procedure calls (RPCs): This is both a communication and a synchronization mechanism, that allows distribution of threads to separate processors in a network and also allows threads to be written in different languages. The calling object furthers an operation call request to a RPC library that finds the remote object in the network and sends the request to it. The receiving side translates the request from the universal format that it is sent by and makes the call. The result is returned in a similar manner.

The communication between objects is modeled using events, signals and messages; the actual implementation techniques are not chosen until the design phase.

2.1.4 Non-functional requirements

Many real-time systems have high non-functional requirements (includes Quality of Service requirements) such as robustness, availability, throughput, capacity, predictability, security and safety.

A system is robust when it behaves correctly under unplanned circumstances. To achieve correctness and robustness, exceptional conditions such as deadlock and race conditions must be prevented and handling of software and hardware errors must be included.

Availability is a measure of the up time; i.e. the probability that a computation will successfully complete before the system fails. It is usually estimated with MTBF (mean time between failure).

Predictability of a system is the extent to which its response characteristics can be known in advance [19]. Aspects of predictability are for example schedulability, memory usage and memory persistence.

Security means permitting or denying system access to appropriate individuals and safety is a measure of how much risk the system incur to the environment.

With throughput we mean e.g. the speed of a packet through the node, from input to output.

Development of reliable and safe system involves architectural redundancy in some form [20]. During modeling, redundancy can be

introduced in the system through the use of architectural design patterns (see chapter 4.2.2).

2.1.5 Distribution

Large real-time systems are often distributed across many processors and across dispersed physical locations. Advantages of distributing large systems are among others, increased reliability, scalability, support for local computing, low start-up costs and resource sharing. It is also much more economical to set up a distributed system consisting of a large number of cheap CPUs than to construct a centralized system consisting of one single expensive mainframe computer.

But there are of course also "dark sides" of distribution. One lies in the increased complexity compared to centralized systems. A distributed system is highly dependent on the communication network: it can lose messages, become overloaded and may not be able to provide the bandwidth needed in an efficient way. It is harder to guarantee security, e.g. identification and verification of requests to servers. The distribution of control also makes the fault detection and the administration harder.

In general, design issues that accompany distributed systems are concurrency, unreliable communication media, prolonged and variable transmission delays, relativistic effects and the possibility of independent partial system failures [6].

2.1.6 Implementation constraints

Real-time systems are often embedded, which means that they are a part of a larger system with the purpose to help it achieve its overall responsibility. The software must often be tightly integrated with the hardware and be able to handle low-level interrupts and hardware interfaces. Embedded systems are generally very hard to validate, because of the lack of debugging tools. Most of them lack the possibility to display errors and diagnostic messages.

As many products are extremely cost-sensitive, the amount of resources in real-time systems is often strongly limited. This places high demands on the real-time developers to make even more efficient software. Instead of replicating expensive hardware, it can in some cases be more economically to replicate software to achieve increased fault-tolerance.

The usage of the services that is provided by the underlying operating system is often greater in real-time systems than in non-real-time systems. RTOS (Real-Time Operating Systems) provide, among others, services for secure resource sharing and thread synchronization. The integration with the RTOS is commonly done during the implementation, but the integration can be handled earlier, by including it in the design model.

2.2 STANDARD UML

The purpose of this chapter is to give readers, unfamiliar with UML, a brief introduction to the different UML diagrams used in the discussions later on in this report. Readers already familiar with these diagrams can skip to chapter 2.3.

Note: In the following chapters, when using the term "UML", we will be referring to Standard UML 1.3 [1], which includes a core and an extension. "UML-RT" will denote UML Real-Time.

2.2.1 Use Case diagrams

A Use Case diagram illustrates a set of *Use Cases* for a system, the *actors*, and the *relations* between the actors and the Use Cases. A Use Case is a named capability of a structural entity in a model [5]. This entity can be the entire system viewed as a black box, a subsystem or even a class. The actors are objects outside this entity that has significant interactions with it.

In RUP a Use Case is explained in the following way: A use case defines a set of use-case instances, where each instance is a sequence of actions a system performs that yields an observable result of value to a particular actor [2].

A Use Case diagram captures a broad view of the primary functionality of the system (or subsystem). Non-technical users, e.g. customers, can easily grasp how the environment can use the system (subsystem).

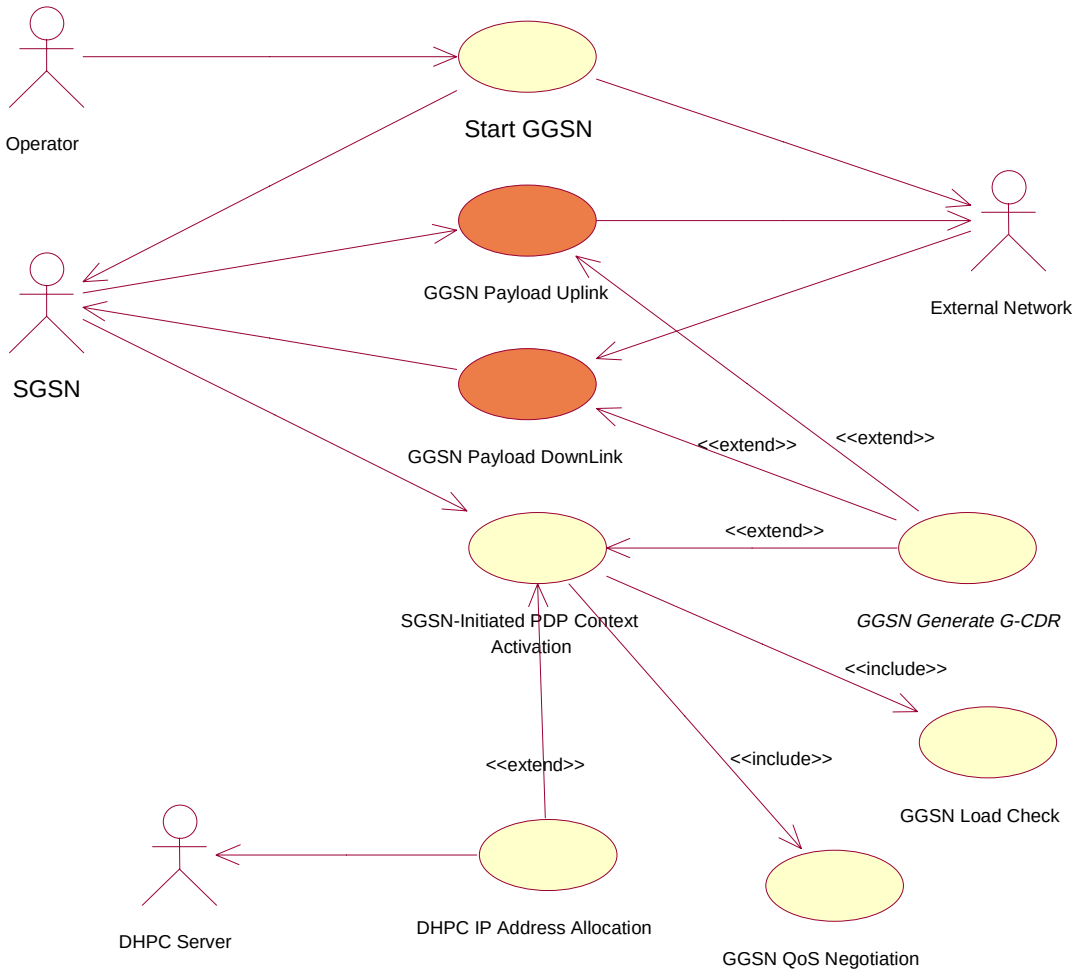


Figure 2. Four Use Cases in the GGSN-Light system with two extensions, marked with stereotype <<extend>>, and two inclusions, marked with stereotype <<include>>.

2.2.2 Sequence diagrams

UML describes interaction with two types of diagrams: Sequence diagrams and Collaboration diagrams. Both diagrams show scenarios but differ in what they emphasize.

The Sequence diagrams illustrate message interactions between instances and classes in the class model. They emphasize *time* and *sequence*. The objects are shown as vertical lines and messages/ interactions as arrows between these lines. Time passes downward in the diagram but is in most tools not to scale.

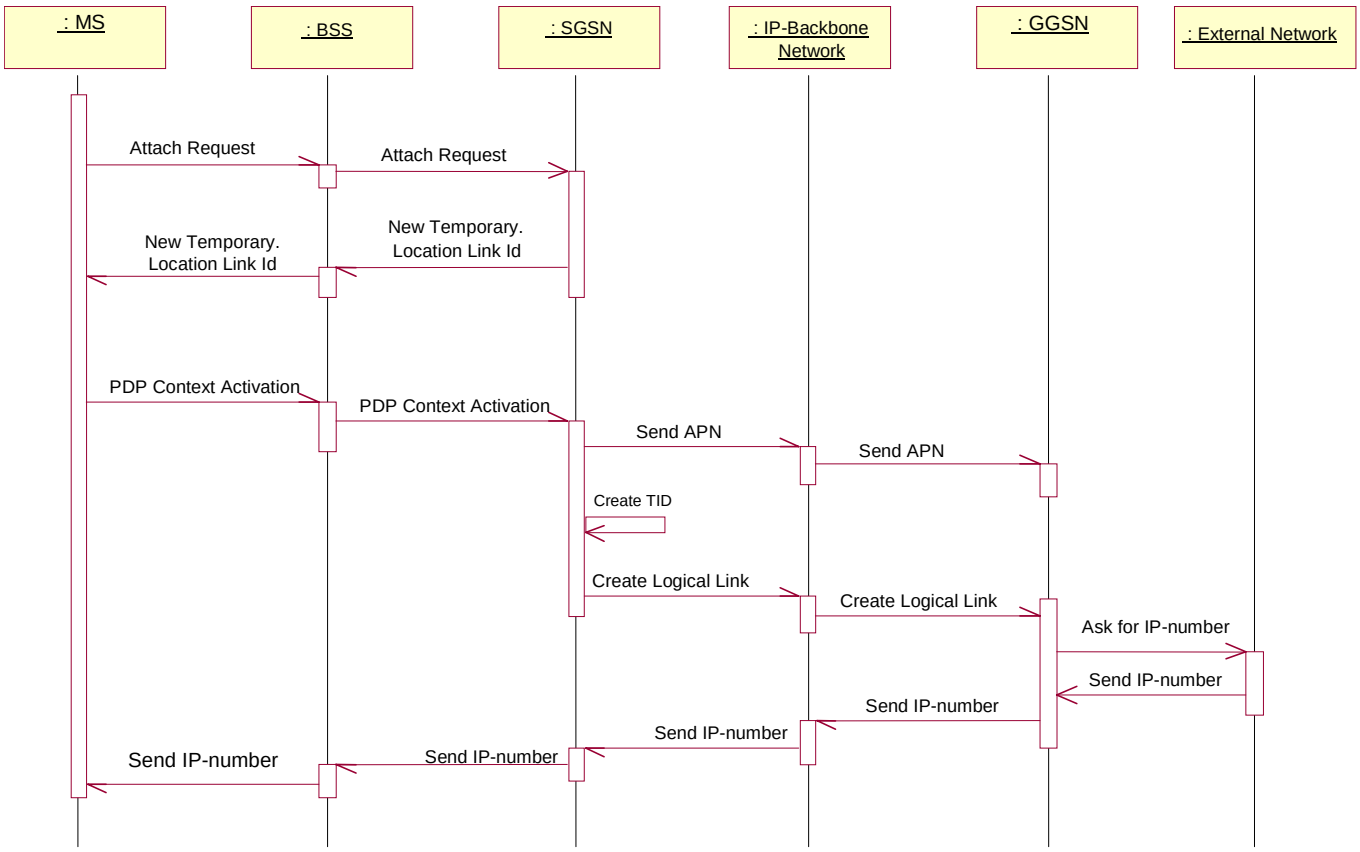


Figure 3. Sequence diagram from Rational Rose showing a sequence in the entire GPRS system during a "PDP Context Activation".

Normally, a Sequence diagram is used to show only one flow of events. Consequently, several sequence diagrams may be needed to describe e.g. one Use Case. There are ways to show alternative, exceptional and concurrent flows in interaction diagrams and we will discuss these possibilities in chapter 4.1.1.

2.2.3 Collaboration diagrams

The other type of interaction diagrams is Collaboration diagram. This too illustrates the message interactions between instances and classes in the class model. It, however, emphasize *context* by showing objects and their relationship. A Collaboration diagram is drawn as an object diagram with message arrows drawn between the objects to show the flow of messages (see Figure 4).

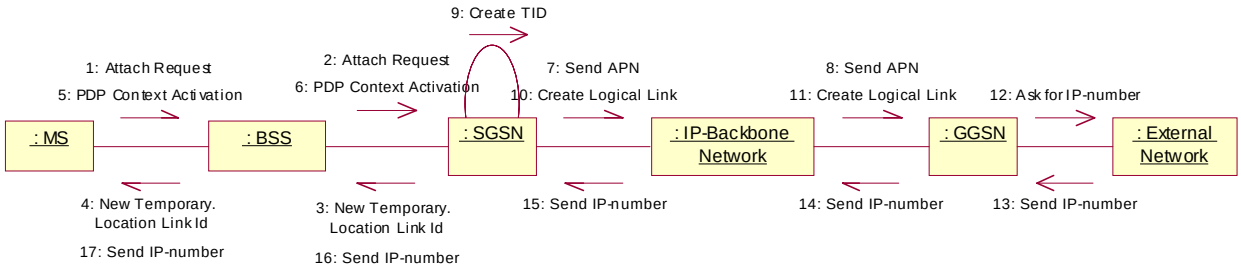


Figure 4. Collaboration diagram from Rational Rose showing the same sequence as in Figure 3.

Labels (sequence numbers) are placed on the messages to show the order, in which the messages are sent, more about this in chapter 4.2.2.

2.2.4 Class diagrams

A class diagram shows the *static structure* of the model, i.e. classes and types and their internal structure and relationship. It can also contain interfaces, packages and instances (objects and links).

The relationships that can be used are *association*, *aggregation*, *composition*, *generalization*, and *dependency* [19].

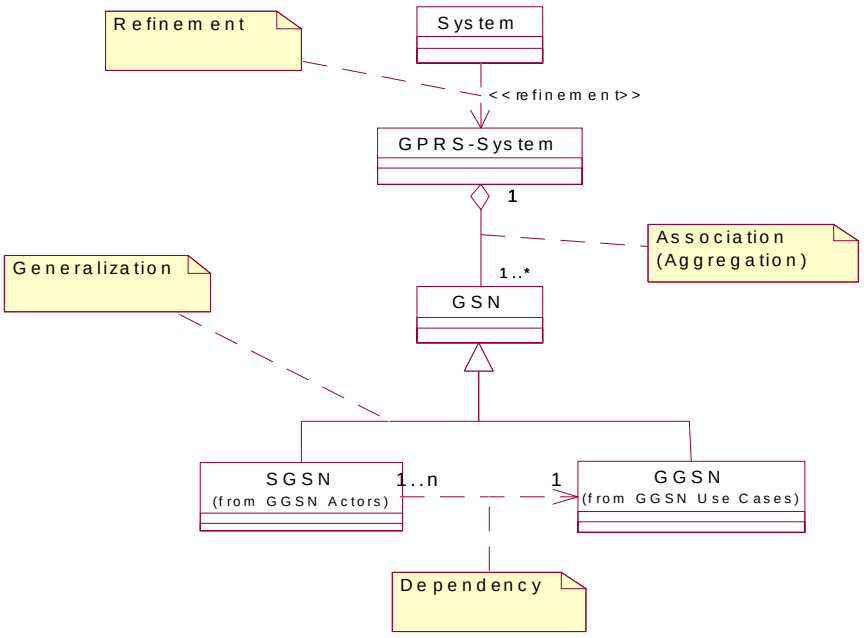


Figure 5. Class diagram from Rational Rose showing different relations among classes. A part of the GPRS system is used to illustrate this.

2.2.5 Object diagrams

Objects can be shown in an object diagram. An object diagram is a variant of a class diagram and uses the same notation and relationships. The objects are drawn as classes but have their names underlined.

An object diagram can be used to exemplify a complex class diagram by showing what the actual instances and the relationships could look like at a certain point in time [16].

2.2.6 Statechart diagrams

A Statechart diagram shows the life cycle of an object, a subsystem or the entire system. It illustrates the events and states of an object, and the behavior of an object in reaction to an event [17]. States are shown as rounded rectangles and transitions as arrows between the objects. An arrow is labeled with the event that triggers the transition along with possible guard conditions (the transition is performed only if the Boolean expression evaluates to true) and actions (that are executed when the transition fires) see also Figure 6.

Syntax: **event-name (parameters) [guard] / action list^event-list.**

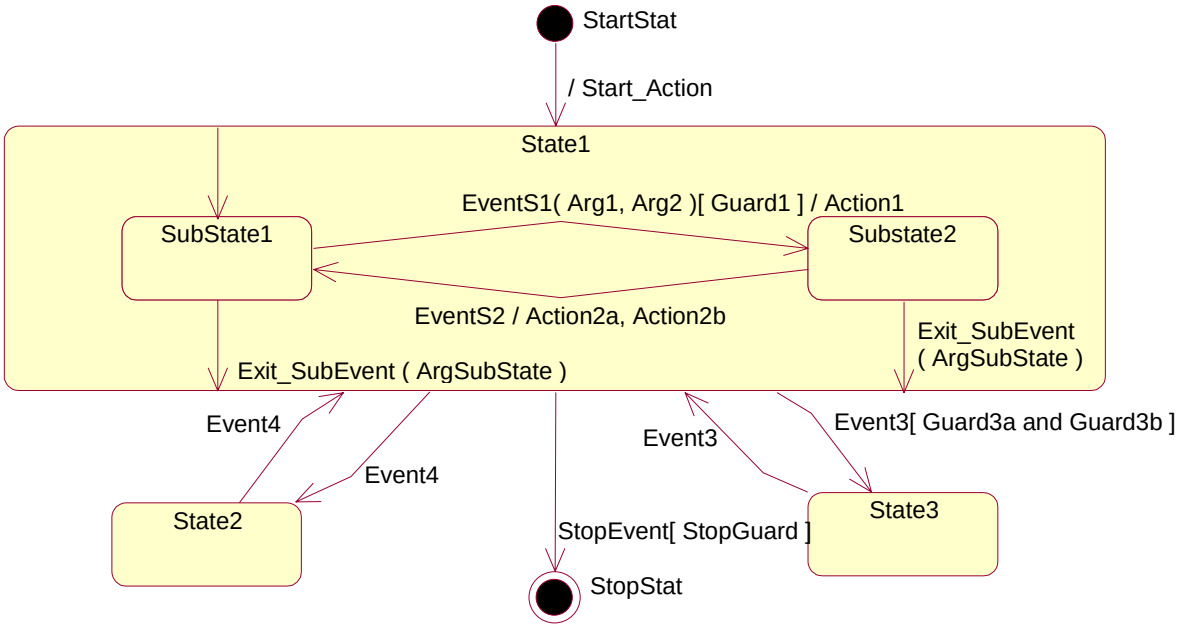


Figure 6. A Statechart diagram from Rational Rose.

Statechart diagrams are commonly attached to classes that have clearly identifiable states and complex behavior. The Statechart diagram can specify the entire behavior of a system or a class; in opposite to the interaction diagrams that only describe parts of the behavior.

Example I: **The Statechart diagram in Figure 6 shows that event "Create PDP Context Response" could not be sent until a PDP Context has been successfully created (the guard [Able to create PDP Context] guarantee this). The "Create PDP Context Request" event must also have happened and the object/system need to be in the "Activate PDP Context" state.**

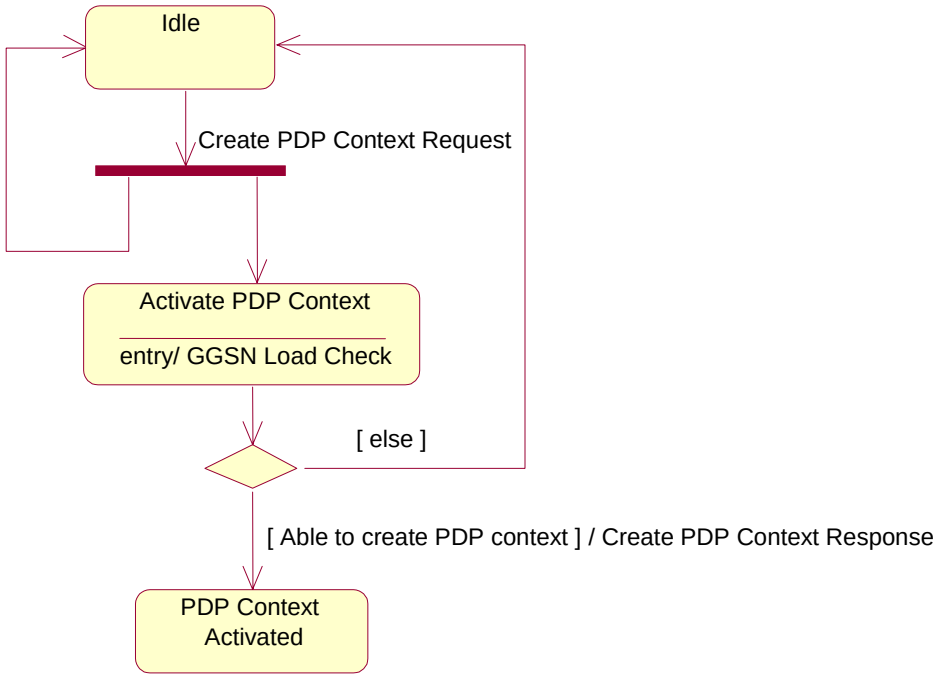


Figure 7. A Statechart diagram from Rational Rose showing the Use Case "Activate PDP Context". Statechart diagrams can be used to show the order among events. It also shows an example where one flow of control is divided into two flows that run in parallel.

As we will discuss later in this report, Statechart diagrams play an important role in the development of real-time systems, partly for their ability to show concurrency (see section 4.2.1). Figure 7 shows an example where one flow of control is "divided" into two flows that run in parallel.

2.2.7 Activity diagrams

An activity diagram shows sequential flow of activities. It is a variant of a Statechart diagram and has a slightly different purpose. It captures a*ctions* and their *results* in terms of object state changes. Activity diagrams were primary introduced into the UML to capture complex business activity models, but can also be used for requirements, analysis, and design modeling.

One difference between an activity diagram and a Statechart diagram is that the states in the activity diagram transition to the next state directly when the action in the state is performed, i.e. the transition arrows are only labeled with guard conditions, send-clauses (to send a

message on a transition) and action-expressions. Events are specified within the states. An exception is that the transition from the Start-state (illustrated by a smaller solid circle) may be labeled with an event.

Another difference is that the actions in an activity diagram may be placed in swimlanes [16]. A swimlane groups activities, with respect to who is responsible for them for them or where they reside in an organization. This could be useful when showing synchronization of threads according to Figure 8.

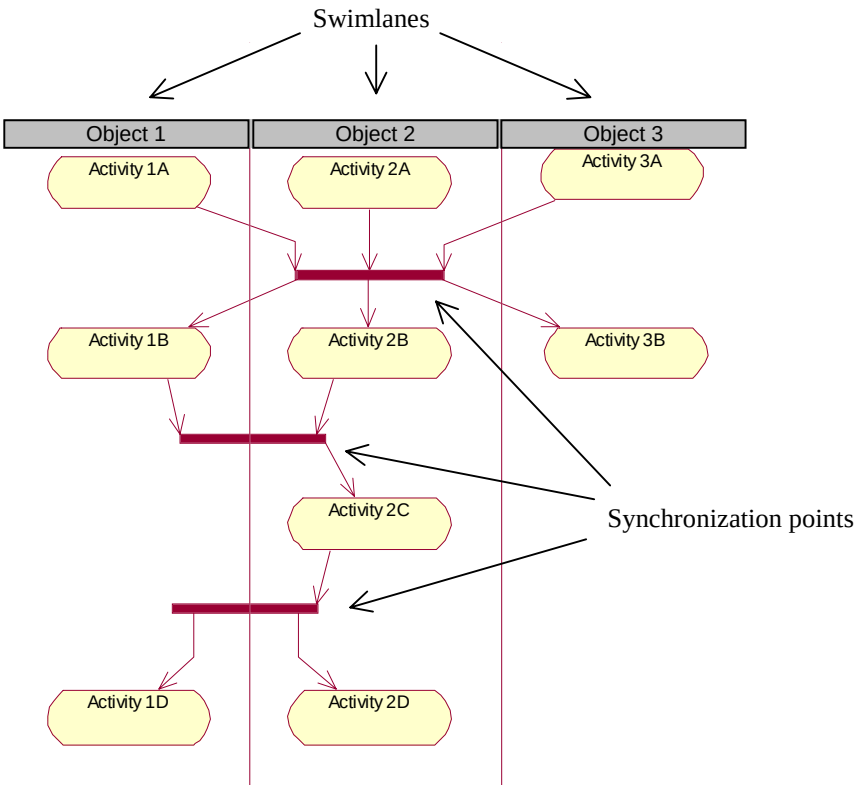


Figure 8. Activity diagram from Rational Rose with Swimlanes, showing how synchronization of threads can be described.

2.2.8 Component diagrams

Physical Architecture is in UML illustrated in two different diagrams: Component diagrams and Deployment diagrams. Component diagrams show compiler and run-time dependencies between software components, such as source code files and DLLs. A component is shown as a rectangle with an ellipse and two smaller rectangles to the left. Only component *types* are shown in a component diagram, instances are shown in deployment diagrams.

If one component needs another component to be able to have complete definition, this is shown by a dependency connection (a dashed line with an open arrow) between the components.

2.2.9 Deployment diagrams

Deployment diagrams show the distribution of processes, components and devises to processing nodes.

Nodes are physical objects such as computers with processors, printers, communication devises and so on. They are drawn as three-dimensional cubes with the name inside.

Communication associations, drawn as straight lines connect nodes to each other.

Only instances of executable components (run-time components) are shown in deployment diagrams. They are connected via dashed-arrow dependencies.

2.3 UML FOR REAL-TIME

UML for Real-Time (UML-RT), developed by ObjecTime Limited and Rational Corporation, is a combination of UML modeling concepts, and special modeling constructs and formalisms defined in the Real-Time Object-Oriented Modeling (ROOM) language [11].

ROOM was originated at Bell-Northern Research [6]. It is a visual modeling language for complex, event-driven and potentially distributed real-time systems. ROOM models are composed of actors who communicate with each other by sending messages along protocols.

Constructs that have been added to the UML modeling *structure* to form UML-RT are Capsules, ports and connectors. Protocols have been added to the modeling *behavior*. Role modeling has also been included as an extension to the standard modeling techniques. Role modeling captures the structural communication patterns between software components [11]. UML-RT focuses more on behavior than UML and as shown later in this report, it is good to focus on state machines as early as possible when designing with UML-RT.

2.3.1 Capsules

Capsules correspond to the ROOM concept *actors* [6]. They are potentially concurrent and possibly distributed architectural components that represent a logical encapsulated thread of control in the system. In UML-RT these Capsules are represented by classes, specialized by the stereotype <<Capsule>>.

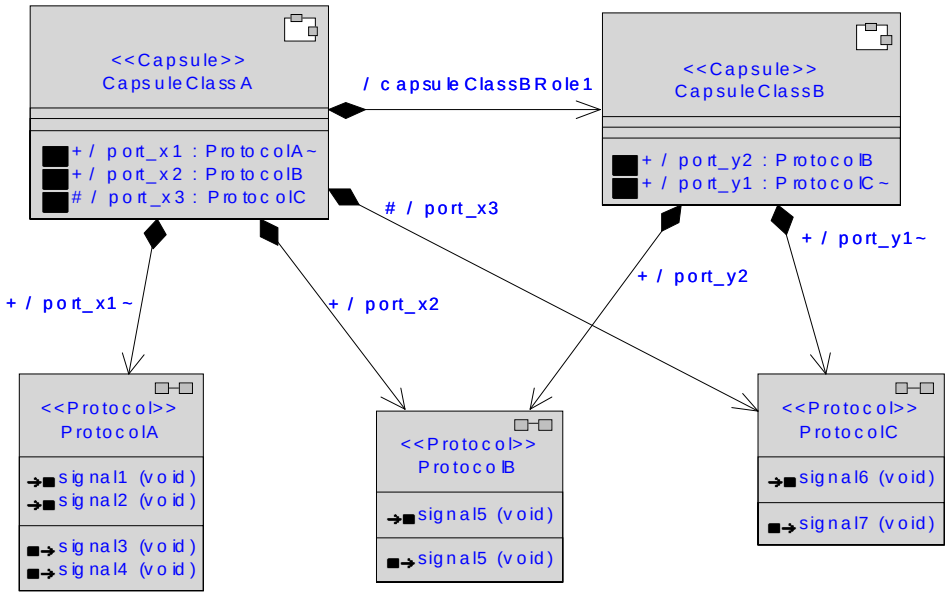


Figure 9. UML notation for Capsules: Class specialized by the stereotype <<Capsule>>.

Capsules are composite objects, i.e. they have an internal structure that can be composed by passive objects and/or other Capsules (a.k.a. Subcapsules). Figure 9 shows a Capsule "CapsuleClassA" with a Subcapsule "CaspsuleClassB". The internal structure of a Capsule is specified in a structure diagram (a kind of collaboration diagram). The structure diagram corresponding to CapsuleClassA is shown in Figure 10.

The state machine associated with the Capsule realizes the functionality. The state machines can be combined with an internal network of collaborating Subcapsules joined by connectors.

The communication between Capsules is based exclusively on message passing between ports; they cannot have public operations or other public parts. They can however have private operations. The communication with passive classes is done "as usual", i.e. the Capsules call public operations in the passive classes.

2.3.2 Ports and Connectors

Ports mediate the Capsule's interaction with its environment. They are physical parts of the implementation of a Capsule and provide a mechanism to export multiple different interfaces. Each port is tailored to a specific Capsule role.

The ports make the Capsule highly reusable by forcing them to fully de-couple their internal implementations from any direct knowledge

they have about the environment. Viewed from the outside of the Capsule, the ports cannot be differentiated except by their identity and their type, however, when viewed from the inside ports can be of two kinds: *relay ports* and *end ports*. Relay ports are connected to Subcapsules (e.g. port_x2 in Figure 9), while end ports are connected directly to the state machine of the Capsule (e.g. port_x1 and port_x3 in Figure 9). Relay ports are thus ports that only pass all signals through. They can only appear on the boundary of a Capsule and consequently always have *public* visibility. End ports may appear either on the boundary of a Capsule or completely inside a Capsule. The *protected end ports* (the ones that are on the inside, e.g. port_x3 in Figure 9) are used by the state machine to communicate with its Subcapsules or with external implementation-support layers (that represent run-time services).

Connectors correspond to the ROOM *bindings*. They are used to connect ports and are hence abstract views of signal-based communication channels. A connector can only interconnect ports with complementary types or ports of a symmetric type (see chapter 2.3.3). A port that has a complementary protocol with another port is called conjugated and is illustrated with a white square. Unconjugated ports are black.

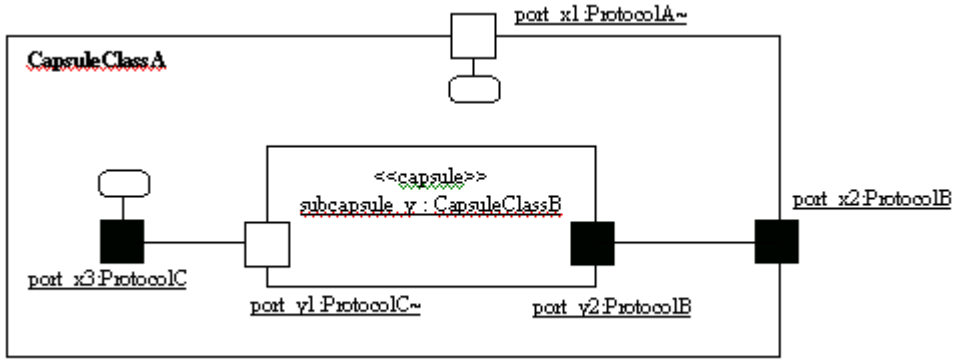


Figure 10. Structure diagram of CapsuleClassA showing the UML-RT notation of ports.

2.3.3 Protocols

Protocols define the valid flow of information (signals) between connected ports of Capsules. They are pure behavior and do not specify any structural elements. Because a protocol defines an abstract interface that is realized by a port, they too are highly reusable.

A protocol role is the specification of the set of in- and out-signals that can be received by and sent from the port [13]. The protocol defines the port type, which means that the port implements the behavior specified by that protocol. Ports must be of complementary types or of

a symmetric type to exchange messages. Two protocols are complementary if every in-signals of one protocol is defined as an out signal in the other protocol, and vice versa (see Figure 11 and Figure 12). A protocol is symmetric if every in signal is also defined as an out signal (in the same protocol) and vice versa (as ProtocolB in Figure 11). A Capsule role will typically require a protocol for each Capsule role that it is associated with.

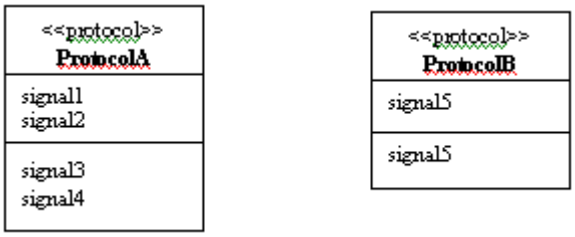


Figure 11. The UML notation of Protocol. ProtocolB is symmetric.

A protocol complementary to ProtocolA would have signals defined according to Figure 12.

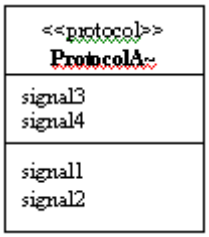


Figure 12. ProtocolA~ is complementary to ProtocolA in Figure 11.

3 DEFINITION OF OUR EVALUATION MODEL

When evaluating the different modeling techniques, we use an evaluation object very similar to the systems developed by ERV, this to provide a relevant reference model that the software developers at ERV easily can associate to. An object in the right context will also help us to catch real-time requirements that are specific for ERV systems.

The evaluation object chosen is the GGSN[9] from the GPRS[10] system. We use the open and official requirements on the node and then we simplify it to limit the size and concentrate on the parts with the most widespread range of real-time requirements. We call this node GGSN-Light. Most of the examples used to explain certain modeling techniques in this report are based on the GGSN-Light system.

3.1 THE GPRS-SYSTEM

GPRS is a technology for sending packet-data over GSM[11] networks. It is one of the first steps toward third-generation mobile telephone networks that will provide multimedia service at high data rate.

The parts of the GPRS System that carry out the switching of packet data are called the SGSN[12] and the GGSN [7]. Figure 13 shows a logical view, according to reference [7], of the GPRS system with the GGSN node connected to the external IP-Network (the Internet) in one end and connected to a number of SGSN nodes in the other end. An IP-Backbone Network physically separates GGSN and SGSN from each other.

[9] Gateway GPRS Support Node
[10] General Packet Radio Service
[11] Global System for Mobile Communication
[12] Serving GPRS Support Node

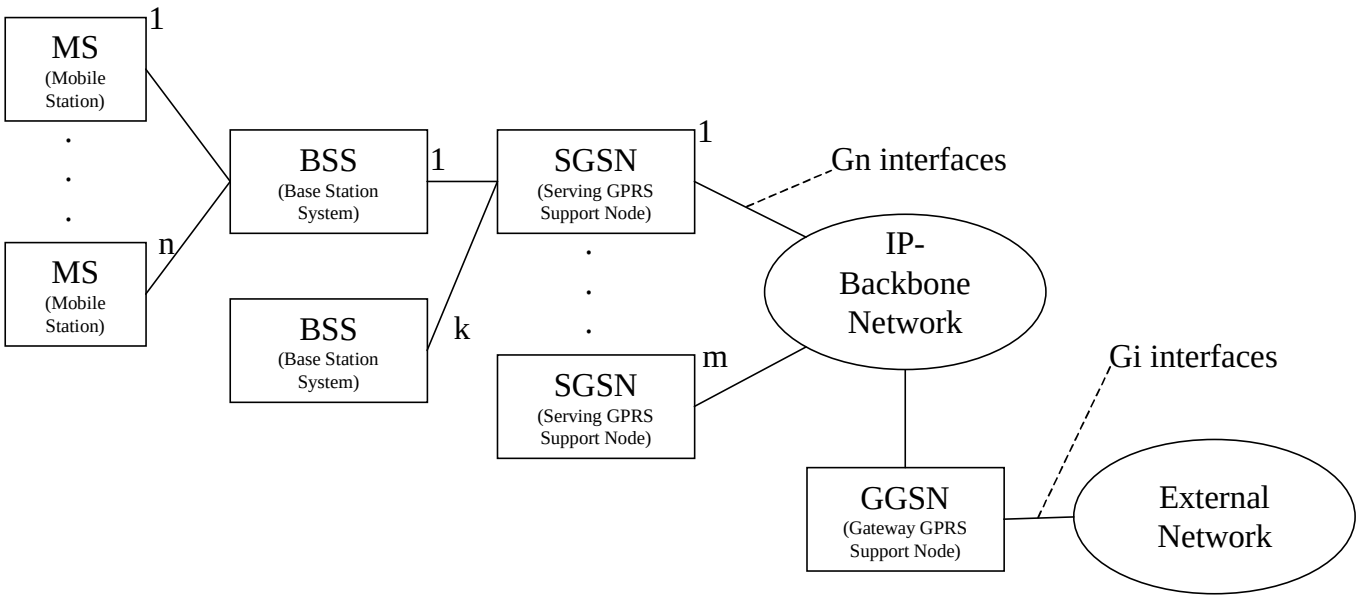


Figure 13. A logical view of the GPRS system, including the names of the interfaces.

3.2 THE GGSN

GGSN is a primary component in the GSM network using GPRS. GGSN provides:

- The interface towards the external IP-Packet Networks. From the external IP-Packet Network's point of view, the GGSN act as a router for the IP-addresses of all subscribers served by the GPRS network.
- GPRS session management, communications setup towards external network.
- Functionality for associating the subscribers with the right SGSN.
- Output of billing data.

3.2.1 The GGSN-Light

The GGSN-Light is, as mentioned earlier, a modified GGSN that serves as our evaluation and work model.

Terminology used for GGSN-Light

PDP Context	This is a connection between a mobile station (or SGSN from GGSN-light's point of view) and the External network.
PDU	Packet Data Unit, a packet from the External Network to SGSN.
IP-Packet	A packet from SGSN to the External Network[13].
G-CDR	GGSN Call Data Record[14], used to store charging data.

Functional Requirements for GGSN-Light

- **Start**. Is initiated when an operator presses the power-on button[15]. During startup the GGSN-Light must gather information about the available hardware in the Control System, distribute processes to the different processors and start the static processes in the Control System. In the Transmission System, the distribution to the different processors must also be done, along with the startup of all the hardware devices. The peer network elements must be informed about the awareness of the GGSN-Light and the GGSN-Light must start to broadcast its IP-address to the IP-Backbone and the external network. The startup of the Transmission System and the Control system should be done in parallel.
- Do **PDP Context Activation**. Is initiated when GGSN-Light receives a *Create PDP Context Request* from the SGSN. The GGSN-Light checks the load situation and negotiates about a QoS level for the session. If they are OK a G-CDR is created and associated with the new PDP Context. The PDP Context is also associated with one Downlink and one Uplink PDU Destination Address. The GGSN allocates a DHCP IP-Address and the sends a *Create PDP Context Response* to the SGSN.
- Do a **Payload Uplink**. The GGSN receives an IP-packet containing a PDU from the SGSN. The GGSN-Light extracts the PDU and obtains the PDU Destination Address to point out the correct PDP Context. The G-CDR is updated if a certain volume limit (VolumeLimit) has been reached and the GGSN-Light forwards the PDU to the external network.
- Do a **Payload Downlink**. The GGSN receives a PDU from the external network and uses the PDU Destination address to point out the correct PDP Context. The G-CDR is updated if a certain

[13] The GTP-U concept has been removed in GGSN-Light.
[14] There is only 1 G-CDR per PDP Context in GGSN-Light.
[15] The possibility to start the node via a management terminal has been omitted in GGSN-Light.

volume limit (VolumeLimit) has been reached and the PDU is encapsulated in an IP-Packet, together with the PDU Destination Address, before it is transferred to the SGSN[16].

- Do a **PDP Context Deactivation**. Is initiated by an internal event[17]. GGSN-Light sends a Delete PDP Context Request to the SGSN, which sends a Delete PDP Context Response back to the GGSN-Light if the Request is accepted. GGSN-Light makes a final update of the G-CDR, closes it and then deallocates the DHCP IP-Address. Finally GGSN-Light removes the PDP Context data.
- **Stop**. Is initiated by an operator request via the management terminal. The GGSN-Light must disconnect all hardware devices in the transmission System and stop all internal applications. GGSN-Light stops the traffic without informing the peer network elements. No traffic data may be lost.

Non-functional requirements for GGSN-Light

- A GGSN Startup must be done within T_1 s.
- GGSN should be able to pass X_1 IP-packets each second between the External Network and the IP-Backbone Network (a pass of a packet is a Payload Downlink or a Payload Uplink). One packet equals X_2 bytes.
- Keep track of X_3 PDP Destination Addresses[18]. A PDP Context has one PDP Destination Address in each direction out of the node.
- Handle $X_3/2$ PDP Contexts.
- Do a PDP Context Activation within T_2 s.

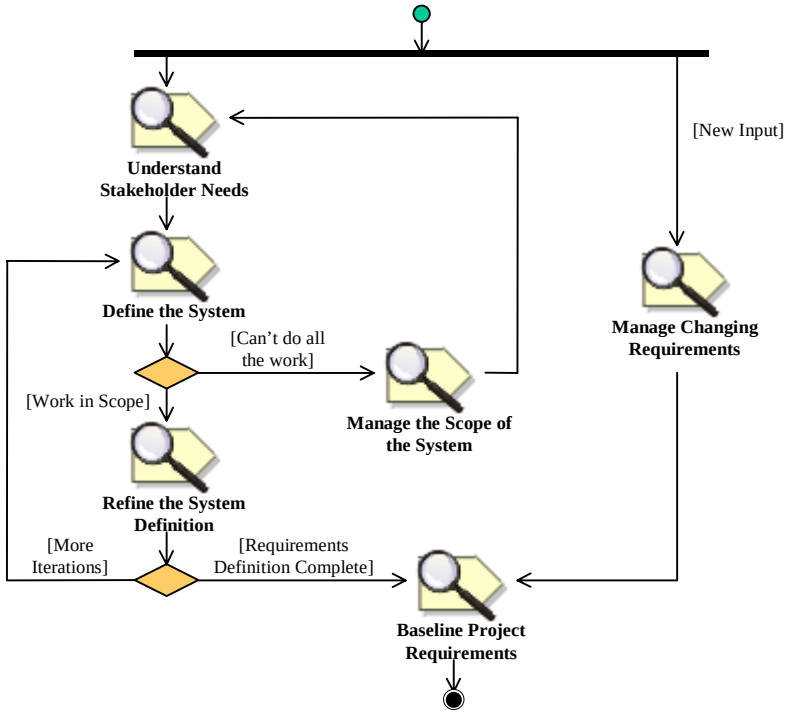
[16] The APN routing is not modeled in GGSN-Light.
[17] This is not the normal procedure, but this is done so we will be able to model an internal event.
[18] In the original GGSN every PDP Context has a TID (Tunnel Identity) in each direction. The TID has been replaced by a PDU Destination Address to simplify the system and the model.

4 MODELING

This chapter presents how UML and UML-RT can be used to model systems with real-time requirements and how Rational Rose and Rose-RT support these methods. The chapter is structured after the different workflow in RUP, starting with the requirement workflow. Only issues and activities that concern modeling of real-time systems will be discussed.

4.1 REQUIREMENTS WORKFLOW

The *Requirements Workflow* in RUP consists of a number of activities (see Figure 14) where the requirements of the system are captured and are gathered in a *Use Case model*. During the requirements workflow the Use Case model is mainly used in the communication with the customer and should have a black-box perspective (i.e. internal details on what the system must do are either missing or described very summarily). The Use Case model is later, during the analysis and design workflow, supplemented with descriptions of the internal flows.



.

Figure 14. The Ericsson GSN RUP adaptation of the requirements workflow.

4.1.1 Use Case modeling

The Use Case model must capture both functional and non-functional requirements. The extraction and exploration of non-functional

requirements are a primary concern in real-time systems, where these requirements often are just as important as functional requirements.

It can be hard to visually model non-functional requirements, but UML provides the capability to model some of these requirements by using OCL (Object Constraint Language) in different ways.

OCL is a formal language, included in the definition of UML, used to express constraints. OCL is a pure expression language. When an expression is evaluated it simply returns a value. It is therefore guaranteed to be without side effects, i.e. it cannot change anything in the model. OCL can however be used to *specify* a change.

Example II:
constraint {LimitX<=LimitY}
guard [LimitX<=LimitY]
invariant {<<invariant>> LimitX<=LimitY}

In chapter 4.1.1.1 and 4.1.1.2 we give a few examples on the possibilities to use OCL expressions in different diagrams that we have seen. Further, it is important to know that an evaluation of an OCL expression is instantaneous, i.e. the states of objects in a model cannot be changed during evaluation.

When collecting and visualizing non-functional requirements through OCL expressions they become easy to model and the traceability of the requirements increase. In the diagrams that are used when communicating with customers it can be appropriate to add the constraints in natural language instead of in the correct OCL syntax. Practice has however shown that specifying constraints in natural language will lead to ambiguities.

There exist tools that are able to verify OCL written constraints but neither Rational Rose nor Rational Rose-RT have this ability today.

4.1.1.1 Use Case Diagrams

The Use Case diagrams give a broad view of the Use Cases and their relationship to the system's Actors. Figure 15 shows a Use Case diagram in Rose-RT where OCL is used to show some non-functional and functional requirements. In chapter 4.2.1.2, "Statechart and Activity diagrams for supplementary Use Case descriptions", we will discuss how OCL can be used for building state machines and from state machines go to Capsules and Protocols.

Guideline 1:	**Use OCL to collect and visualize non-functional requirements in Use Case diagrams as OCL-expressions are unambiguous and can be used to verify the Use Cases.**

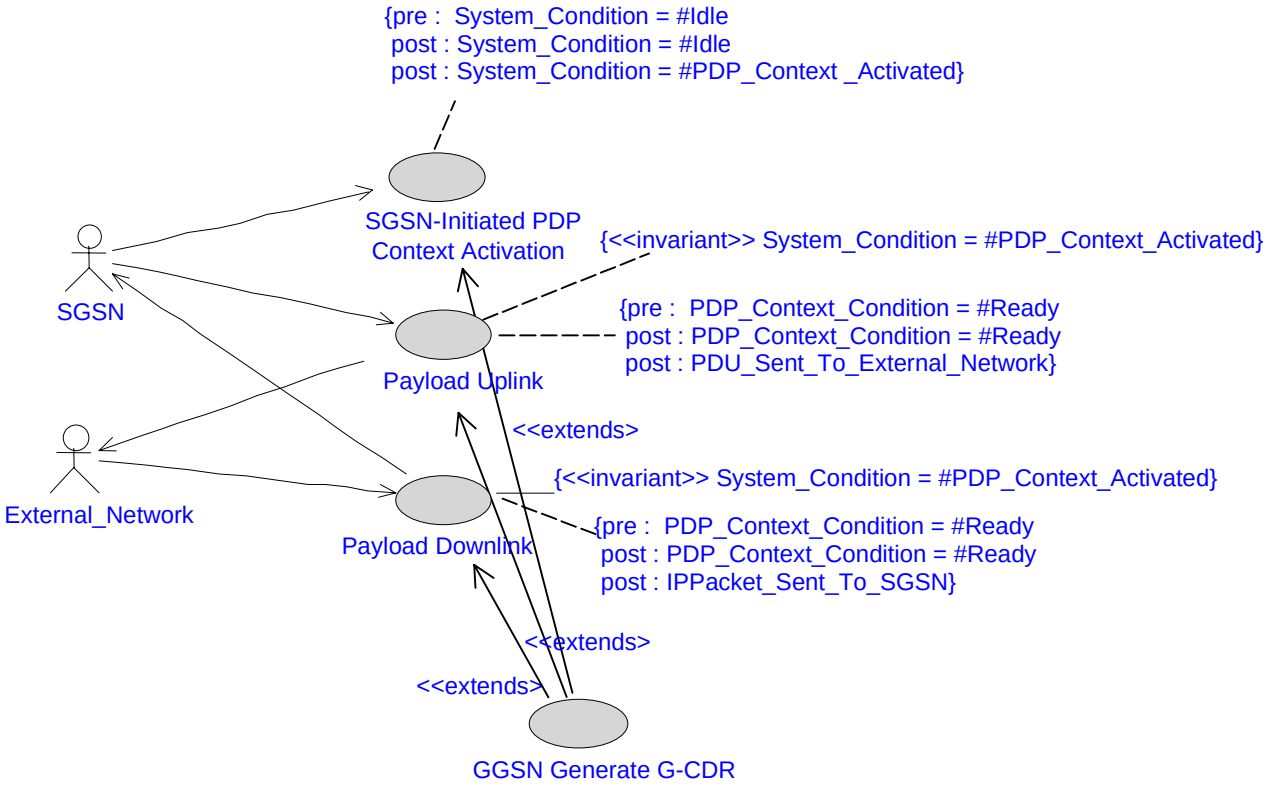


Figure 15. Use Case diagram from Rational Rose-RT with some of GGSN-Light's constraints written in OCL.

In Rational Rose it is not possible to link OCL constraints directly to model entities. Instead the OCL expressions have to be written in *Notes* that are attached to the entities (see Figure 16).

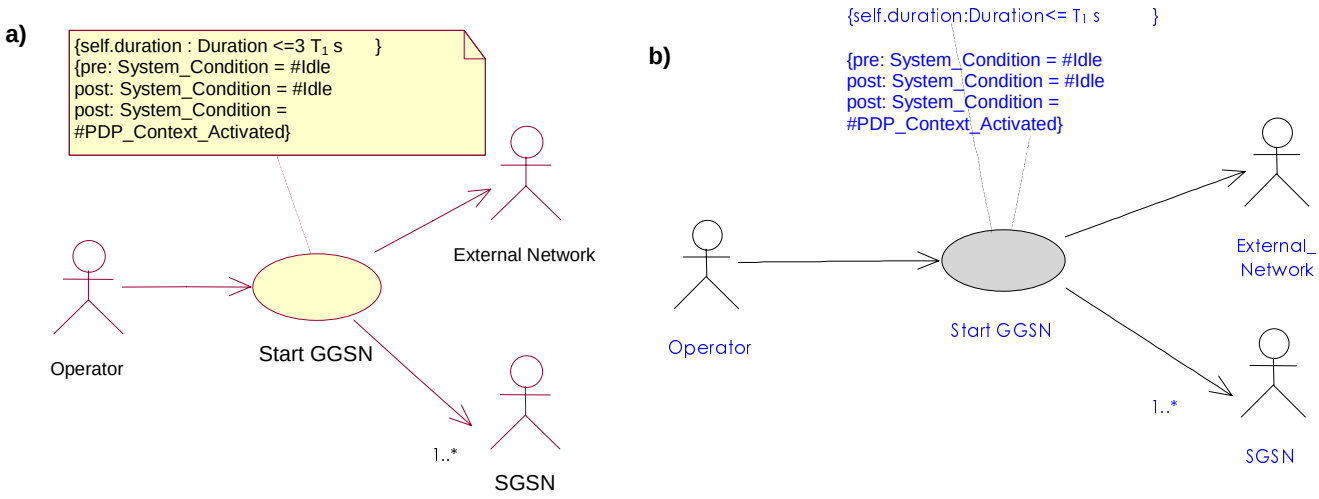


Figure 16. OCL constraints added to a Use Case in a) Rational Rose and b) Rational Rose-RT.

It is important to remember that a Use Case diagram is not a complete description of a Use Case. The Use Case diagram must be complemented with textual descriptions and other diagrams. Despite the advantages of complementing Use Case diagrams with diagrams like interaction diagrams and state machines they are not yet widely used in Use Case models. According to *Use Case Modeling Guidelines* [21] for ERV's GSN-projects each Use Case should have a name, a short description, a Use Case diagram showing the relationship between the Use Case and the Actors involved and a Use Case Specification describing the different flows of events (main flow, exceptional flows and alternative flows). The Use Case Specification is a separate Word document that is linked to the Use Case in the Rose model. There are no guidelines for the use of interaction and Statechart diagrams.

One of the general advantages of using diagrams in parallel with the textual descriptions is of course that they provide a clearer view of flows of events and important requirements. Diagrams are often more unanimous than textual descriptions. It is easier to discover mistakes, errors and issues that have been overlooked.

4.1.1.2 Sequence Diagrams

Sequence diagrams can be used to show the different flows of events of a Use Case and the requirements and constraints on these flows. Since Sequence diagrams focus on time and it is appropriate to show timeliness requirements in these diagrams. In the Sequence diagrams in the requirements workflow the system should be shown as one single black-box object, i.e. only the system's interaction with its environment should be shown, no consideration to what must be done inside the system should be taken. The interaction with the environment can be represented by "message arrows", but at this stage, i.e. in the Use Case model, they should not be viewed as concrete messages (messages at implementation level) but as a notation for high-level interaction.

Performance budgets, such as overall performance and response times can also be computed from a black-box perspective. Later in analysis these overall system reaction deadlines propagate into performance budgets on the individual operations and actions within the system design.

Guideline 2:	**Use Sequence diagrams to show Main and Alternative flows of Use Cases as they show the flows in a clear and unambiguous way.**
Guideline 3:	**Show no more than the system as a "black-box" together with the Use Cases' actors in the Sequence diagrams during the requirement workflow.**

Just as in the Use Case diagrams, Sequence diagrams can connect OCL constraints directly to the model entities in Rose-RT and do not have to use Notes for this as in Rose. Figure 17 shows two examples where OCL has been used.

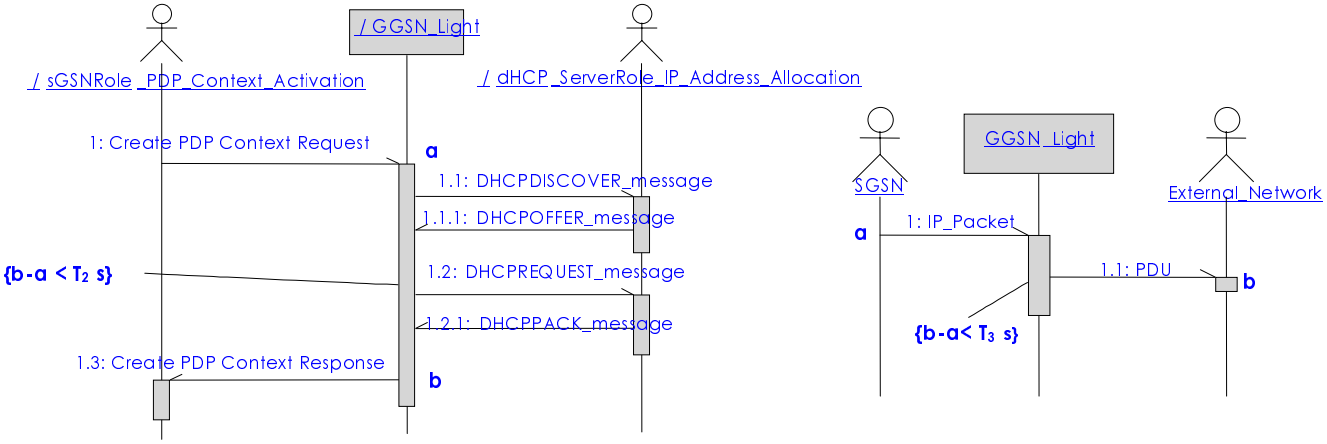


Figure 17. Example from Rational Rose-RT showing how OCL can be used in Sequence diagrams.

As an alternative to Notes, Rational Rose can add OCL constraints as pure text, but then they cannot be connected to model entities (see Figure 18).

Guideline 4:	**Use OCL in Use Case Sequence diagrams to model timeliness requirements like response time, deadlines, throughput and so on. They provide, among others, an aid during the verification of the Use Case.**

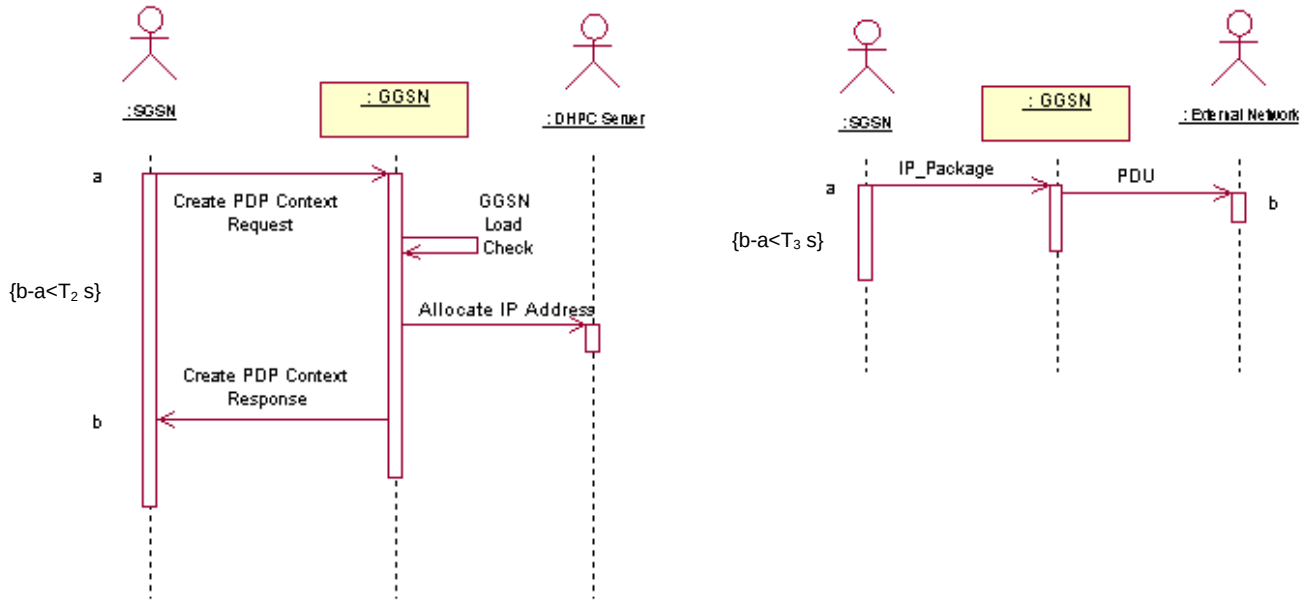


Figure 18. Example from Rational Rose on the use of OCL constraints in Sequence diagrams. The first constraint is a deadline and the second a throughput. The OCL constraints are added as pure text and are not connected to the model entities.

It is important to remember that Sequence diagrams dose not specify the entire behavior of a Use Case. They are helpful in extracting and understanding requirements. They can also be used to define (or generate) test specifications for the system, e.g. customer's acceptance test.

4.1.1.3 Statechart and Activity diagrams

When developing systems with a high degree of concurrency it is important to focus on states and state machines early in the development process, especially when using UML-RT. This because of the fact that the design of Capsules to a high extent revolves around state machines. An early focus on states will therefore make the transformation from the Use Case model to the Design model easier and more efficient. In the requirement workflow, the Statechart diagrams should, just as the Sequence diagrams, only show the system from a black-box perspective. The states in the Statechart diagrams should correspond to the state or condition of the entire system and not to states of different subsystems.

Statechart diagrams can be used to describe pre- and post-conditions for a Use Case, i.e. to specify in which condition the system must be in before and after a Use Case is performed. Statechart diagrams are also powerful when used to describe alternative and exceptional flows in a Use Case. Exceptional flows for example can come anytime during

a Use Case and a Statechart diagram can model this unambiguously. This is exemplified in Example III.

Example III: **Figure 19 shows the Statechart for Use Case "Start GGSN". Here the main flow is "Power on" from operator to state "Startup", when "Startup" has finished the system will be in idle condition, i.e. state "Idle". There is an alternative flow, named A1:Startup Terminated" which will put the system in Failure condition. There is also an exceptional flow, named E1:Restart. This can take place any time during the Use Case and will result in a new start of the system. Figure 20 shows the same states but after supplementing the Use Case description.**

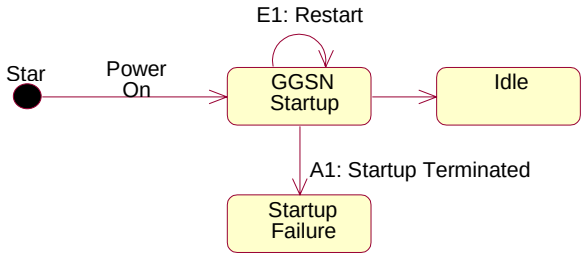


Figure 19. The state machine (from Rational Rose) for Use Case "Start GGSN" showing: 1) Main flow from "Start" to "Idle". 2) Alternative flow to "Startup Failure" and 3) Exceptional flow.

When switching to a white-box perspective (in the supplementary Use Case descriptions) the advantage is even clearer. Here the "GGSN Start"-state is expanded (see Figure 20) with sub-states that specify what must be done within the system. More about supplementary Use Case descriptions and advantages of state machines can be found in chapter 4.2.1.2).

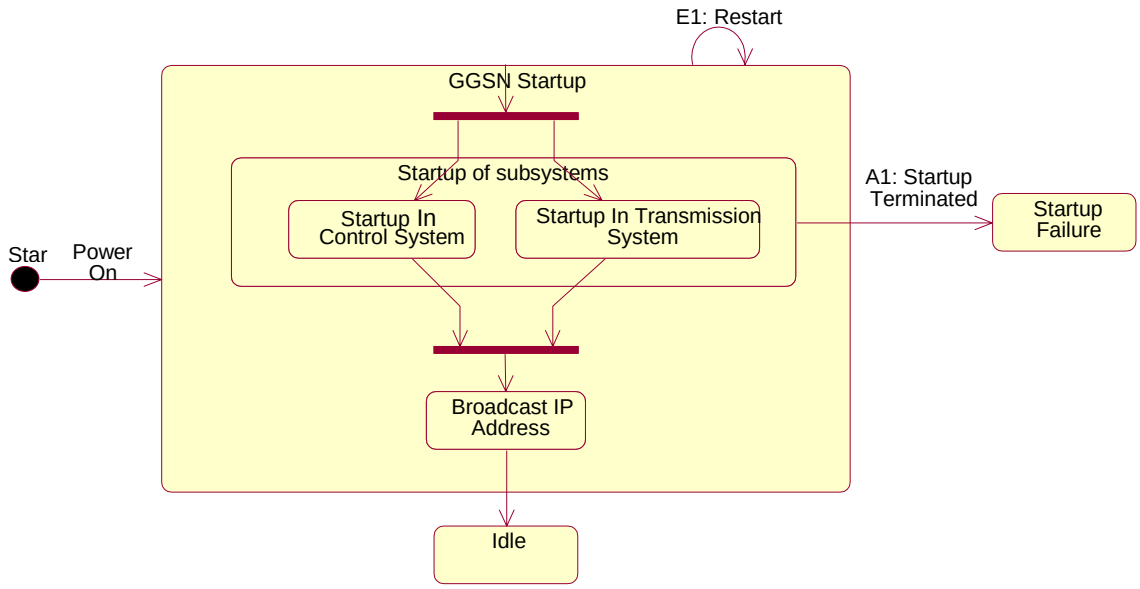


Figure 20. Same Use Case as in Figure 19, but after supplementing the Use Case description.

If concurrency is not shown or cannot be shown in the Sequence diagrams, Statechart and Activity diagrams are even more important and useful, because of their ability to show concurrency (e.g. among and within the Use Cases).

4.2 ANALYSIS & DESIGN WORKFLOW

The purpose of the analysis and design workflow is according to RUP [2]:

- To transform the requirements into a design of the system to-be.
- To evolve a robust architecture for the system.
- To adapt the design to match the implementation environment, designing it for performance.

During the analysis and design workflow the work continue on the Use Case model and an Analysis and a Design model are produced.

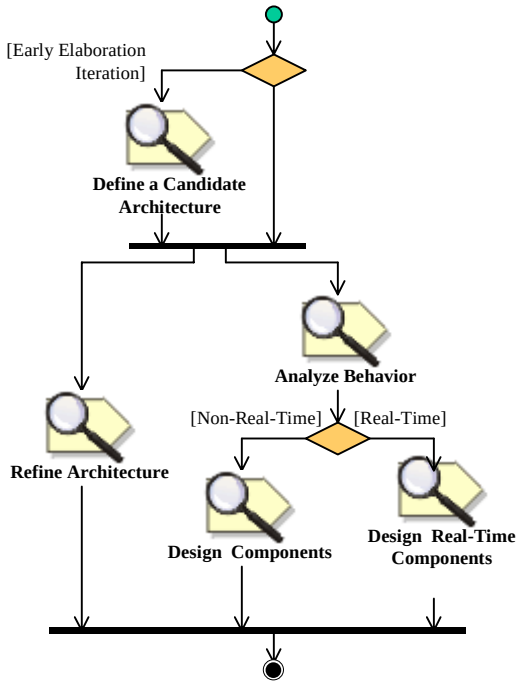


Figure 21. The Ericsson GSN RUP adaptation of the analysis and design workflow.

4.2.1 Supplementing Use Case descriptions

One of the first activities during the analysis and design workflow is the Use Case Analysis (included in "Define a Candidate Architecture" and in "Analyze Behavior"). The first step in this activity is to make supplementary Use Case descriptions. The system should no longer be regarded as a black-box system. To be able to find analysis classes there must in most cases also be a white-box description of the system, i.e. a description of what the system must do from an internal point of view. Diagrams useful when supplementing the Use Case descriptions are Sequence, Statechart, and Activity diagrams.

4.2.1.1 Sequence diagrams for supplementary Use Case descriptions

When supplementing Use Case descriptions the system is still regarded as one single object in the Sequence diagrams, but now internal actions are added with the help of Internal Messages and Local Actions (Local Actions are only available in Rational Rose-RT). One of the main issues for real-time systems is concurrency. Concurrency can be modeled in Sequence diagram by showing parallel focus of control (FOC) and with thread identifiers. The opinions are divided about whether it is good modeling practice to do so. Many tools, among them Rose, do not support parallel flow in Sequence diagrams. In these tools one sequence diagram has to be made for every flow that should be modeled. This can be time-consuming, especially during the requirements workflow were the system is

modeled as a single object. According to the OMG UML Specification [1], there are several different formats of Sequence diagrams. A *generic* Sequence diagram describes all possible sequences and an *instance* Sequence diagram describes one actual sequence. Of course there exist formats in between these that do not show all possible flows, but at least more than one.

Guideline 5:	**When focusing on concurrency, use the sequence diagram format that contains several possible sequences in the same diagram.**
Guideline 6:	**When focusing on execution control, use the sequence diagram format that contains only one actual sequence.**
Guideline 7:	**Sequence diagrams describing Use Case behavior should not focus on execution control. In a real-time system, dealing with concurrency, use the Sequence diagram format that contains several possible sequences in the same diagram.**

In Rational Rose-RT, internal actions can also be indicated by rectangles on the object lifeline, so called *Local Actions* (see Figure 22). More specifically a Local Action represents a significant activity or operation being performed by the object instance at that time. According to RUP the Use Case descriptions made during the Requirements workflow should show how and when the Use Case uses data stored in the system, or stores data in the system. Local Actions provides a god means of doing this in Sequence diagrams. As Rational Rose does not support Local Actions, Notes or Internal Messages have to be used instead.

Guideline 8:	**Use Local Action (when using Rational Rose-RT) or Notes or Internal Messages (when using Rational Rose) to show how data is used during a Use Case.**

Another concept that exists in Rational Rose-RT but not in Rational Rose is co-regions. Co-regions are used to indicate a set of outgoing or incoming messages/events whose ordering is undefined. They can also be used to obtain thread identifiers (letters a, b, ...) in the sequence numbering (see Figure 22). These thread names are very important in Collaboration diagrams where message ordering and concurrency are not shown visually in the same way as in Sequence diagrams. More about thread identifiers and sequence numbering in chapter 4.2.2.3.

The start and end of a co-region are marked with a thick black horizontal line on the object's lifeline.

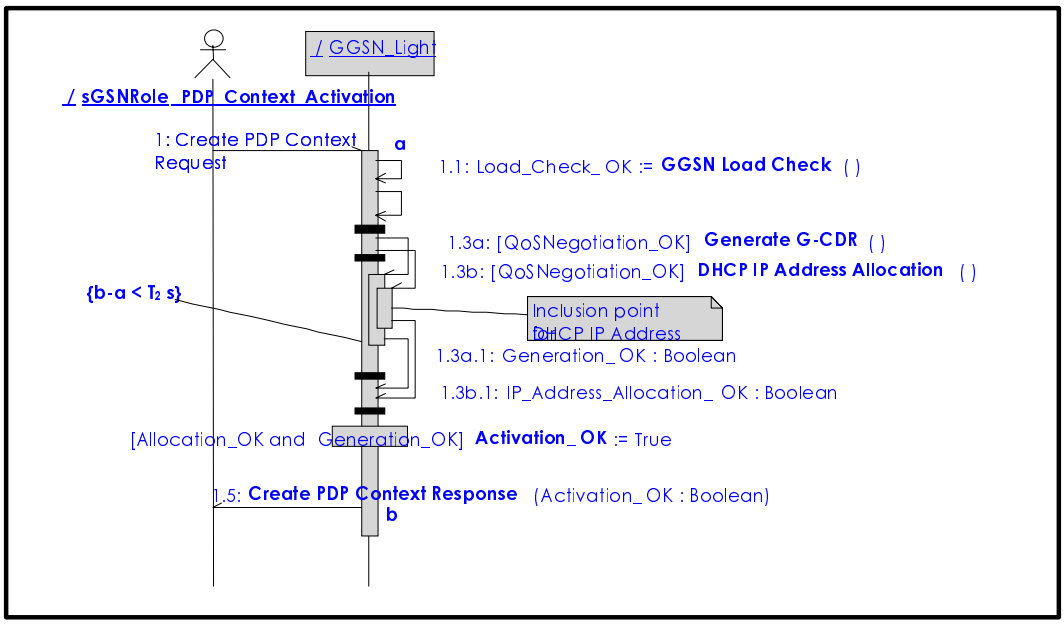


Figure 22. Sequence diagram of supplemented Use Case "Start GGSN" (from Rational Rose-RT) showing a Local Action and Co-regions.

Figure 22 also shows the use of OCL in guards. The guards prevent the activities from being performed if a certain requirement is not fulfilled. The activity "Generate G-CDR" for example may not be performed if the "QoS Negotiation" was not successfully performed. Guards can hence be used to show alternative flows.

Another advantage of Rational Rose-RT (over Rational Rose) is that it can show states in the Sequence diagrams in order to bridge the gap between Sequence and Statechart diagrams. This is mainly useful in the Analysis and Design model, but can also be powerful in the Use Case model if Statechart diagrams are used for defining Use Case behavior. States in Sequence diagrams are shown as rounded rectangles placed on the vertical object lines. They are inserted to mark a change in an object's state. The object then stays in this state until a new state appears further down on the object line.

Guideline 9:	**Only show Use Case states in the Sequence diagram if the Use Case behavior also is described in Statechart diagrams. States in Sequence diagrams bridge the gap between Sequence and Statechart diagrams.**

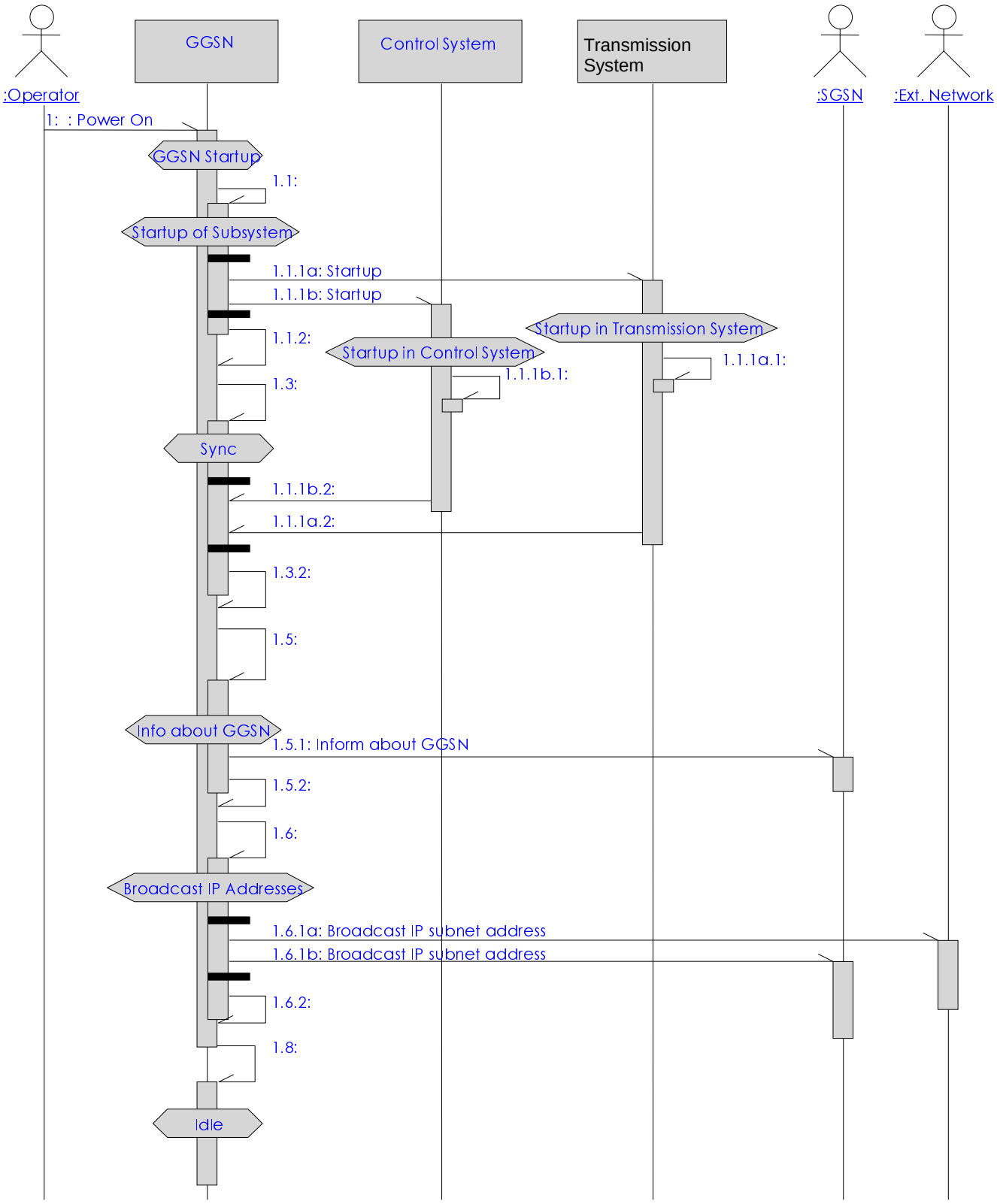


Figure 23. Example from Rational Rose-RT showing the use of state marks in Sequence diagrams. Compare this with the state machine for the same Use Case shown in Figure 23 on page 40.

4.2.1.2 Statechart and Activity diagrams for supplementary Use Case descriptions

Statechart diagrams are typically used for describing the behavior of classes, but they may also be used to describe behavior of the whole system. Modeling Use Case with Statechart and Activity diagrams could be a good choice in a couple of cases:

Firstly, when using Rational Rose-RT and UML-RT, the design focuses on behavior and is based mainly on Statechart diagrams. By using Statechart and Activity diagrams to analyze and describe Use Case behavior, the transformation from the Use Case model to the Design model is made easier and more efficient.

Guideline 10:	**For systems with a high degree of concurrency, use state machines when analyzing Use Cases in order to make the transformation from Use Case Model to Design Model easier.**

Secondly, post- and pre-conditions for Use Cases are often hard to model. Statechart diagrams can be used to verify the pre- and post-conditions of the Use Case. Each Use Case shall have at most one state machine.

Example IV: **According to Figure 24, the end-state of Use Case "SGSN Initiated PDP Context" is the same as the Superstate state of Use Case "Payload Uplink". This means that "SGSN Initiated PDP Context" must be performed before Use Case "Payload Uplink" can start.**

Guideline 11:	**Use state machines to specify the behavior of the system to verify the pre- and post-conditions of the different Use Cases.**

Thirdly, Statechart and Activity diagrams are good means of showing and analyzing concurrency dependencies within and among the different Use Cases. Each Use Case is modeled, in a Statechart or Activity diagram according to Figure 24, with a state machine that represents the change of the whole system during that Use Case and with start and stop states representing pre- and post-conditions of that Use Case. This hence shows concurrency and dependencies within the Use Cases.

Guideline 12:	**Use Statechart and Activity diagrams to describe Use Cases in order to visualize and elaborate concurrency and dependencies within Use Cases.**

All state machines can then be combined to a comprehensive state machine that will show the complex concurrent behavior of the whole system, see Figure 25.

Guideline 13:	**Use Statechart and Activity diagrams to describe Use Cases in order to visualize and elaborate concurrency and dependencies among Use Cases.**

The transformation of the system's state machine into Capsules and Protocols will be described in section 4.2.3.2, Designing with Capsules.

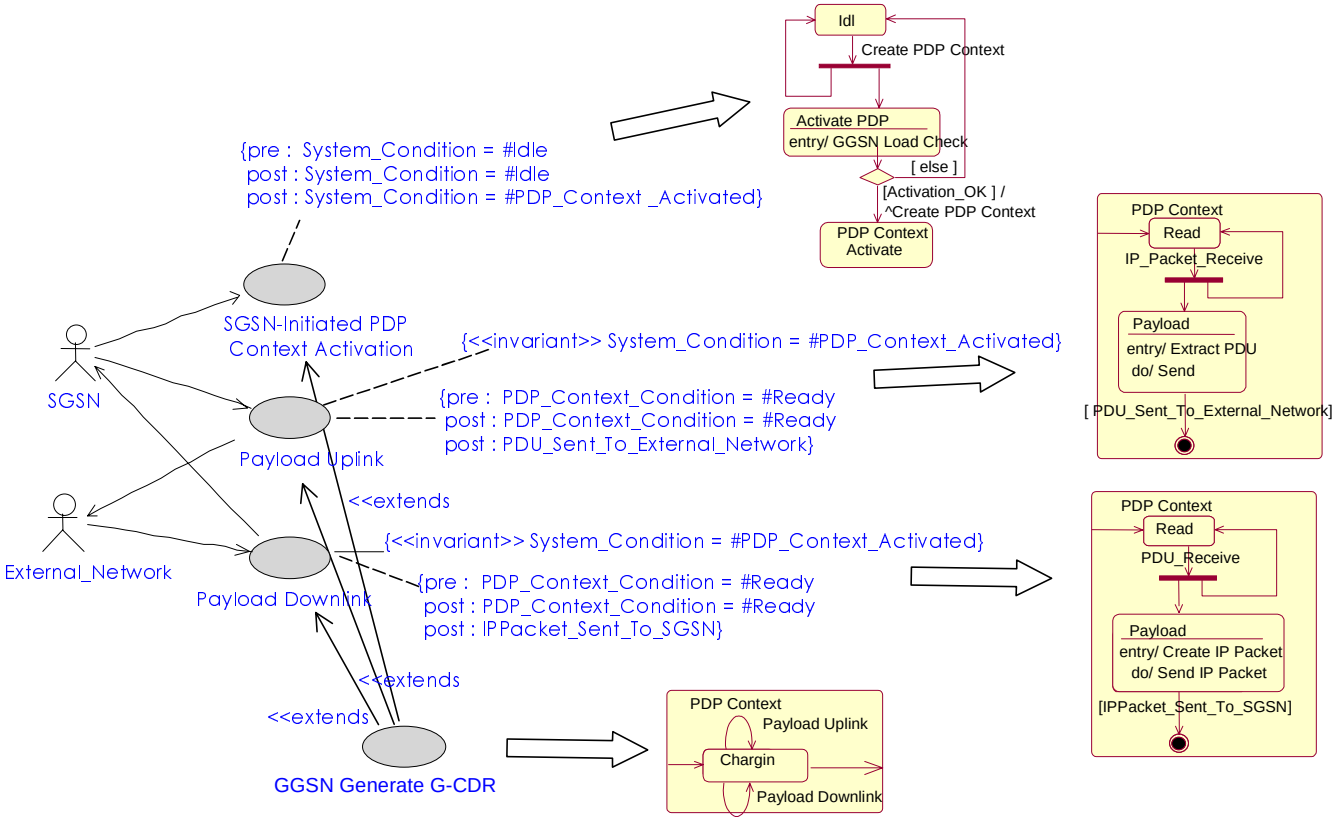


Figure 24. Four different Use Cases are modeled with one state machine each that represents the change of the whole system during each Use Case. The state machines have been modeled in Rational Rose.

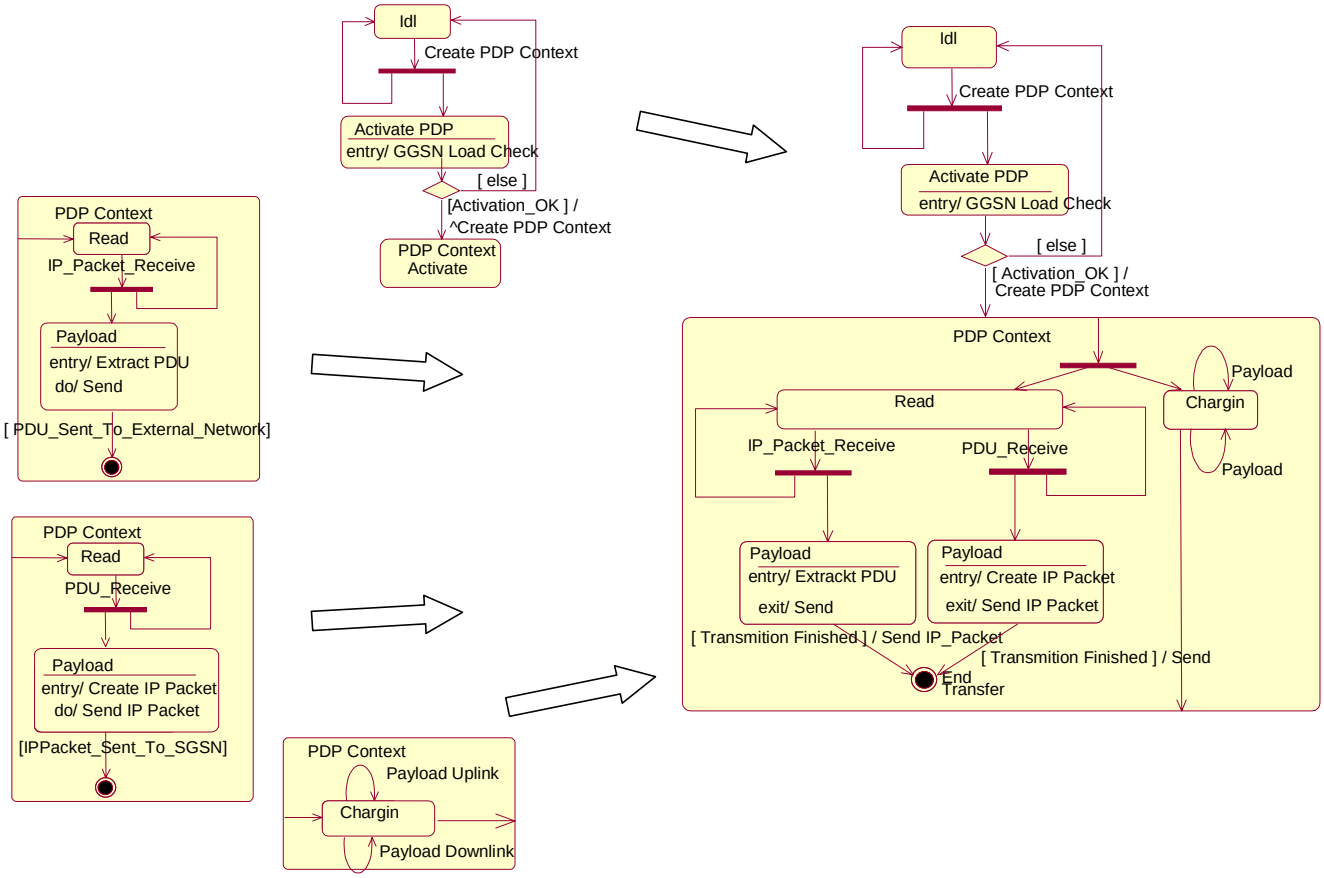


Figure 25. State machines for four different Use Cases (from Rational Rose) combined into one comprehensive state machine showing the complex concurrent behavior of the whole system.

Guideline 14:	**When making Statechart diagrams for Use Cases concentrate only on the states of the entire system and not on the states of different subsystems.**

The UML standard contains *simple* and *composite* states. The composite states are decomposed into two or more mutually exclusive disjoint substates or into concurrent substates (so called *regions*). The *regions* are used to describe concurrency in state machines. Dashed lines are used to divide the composite states into regions (see Figure 26).

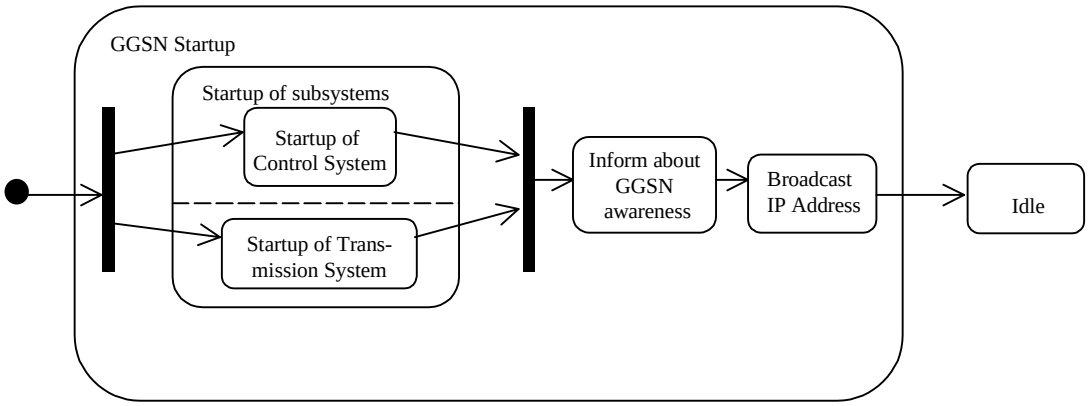


Figure 26. State diagram of Use Case "Start GGSN" used as an example showing the use of regions according to the UML standard. The Startup of the Control System and the Transmission System are performed concurrently.

Composite states can be modeled in the same way in Rose, shown in Figure 27, with one exception; the regions are not marked with dashed lines.

<u>Note:</u> Regions must not be confused with swimlanes. They are not the same thing. Swimlanes show which part of the system that is responsible for the actions and a thread can "move" between different swimlanes during its execution. A region on the other hand symbolizes the encapsulation of a thread and actions in the same thread cannot reside in different regions.

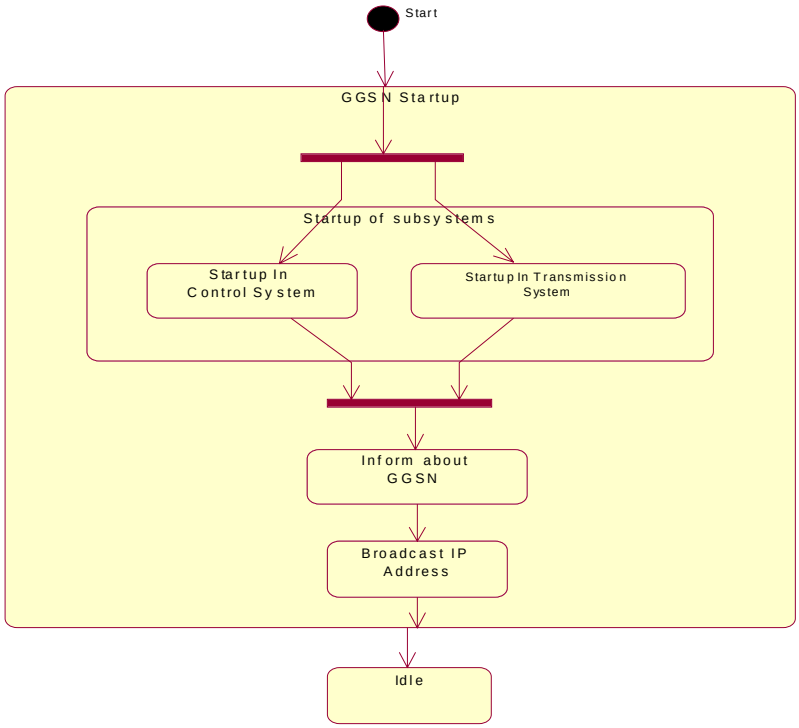


Figure 27. The example from Figure 26 modeled in Rational Rose.

In Rational Rose-RT the use of Statechart diagrams is a bit different and more suited for design. Because of the structure and functionality of Capsules there is no need for the possibility to show concurrency in state machines in the design (more about this in Chapter 4.2.3.2). It is still needed during Use Case modeling but unfortunately Rational Rose-RT does not support both constructions of state machines.

An activity graph is, as mentioned in Chapter 2.2.7, a variation of a state machine. Activity diagrams can be easier than Statechart diagrams to start with when describing Use Case behavior. The requirements and functionality of a system is often easier to formulate on the form: "The system shall first do... and then..." (which graphically corresponds to an Activity diagram) than on the form: "The system is in the state ... and when... occurs the system transits to the state..." (which corresponds to a State diagram). From the Activity diagram the work proceeds with arranging the different actions into states.

Guideline 15:	**When describing Use Case behavior with state machines in Rational Rose, use the Activity diagram as a means for finding the systems states.**

The Activity diagram contains, just as the State diagram, a notation for showing concurrency with the help of synchronization bars. The different actions may be organized into swimlanes to show what part of the system that is responsible for what action.

Rational Rose-RT does not support Activity diagrams. They are only available in Rational Rose.

4.2.2 Architectural analysis, Analysis classes/roles and Use Case realizations

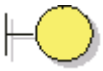
During the first parts of the analysis and design workflow a candidate architecture is defined and a candidate set of analysis classes founded from the Use Case behavior. Now the focus is changed from what should be done to how it should be done. Use Case realizations are made to serve as a bridge between the Use Case and the Design Model and to provide a way to trace behavior between the two.

When defining a candidate architecture and finding analyze classes there are a number of Patterns available that are specific for real-time systems. There are safety related patterns, e.g. *Firewall Pattern* [19], *Multi-Channel Voting Pattern* [19], *Redundancy Patterns* [19], *Watchdog Pattern* [19], *Safety Executive Pattern* and framework related patterns, e.g. *Microkernel Arcitectural Patterns* [19], *Observer Pattern* [18], *Proxy Pattern* [18], *Broker Pattern* [19], *State Pattern* [19]. We will not look closer into these patterns in this report, but we can recommend the following literature for more information:

- Doing Hard Time, Developing real-time systems with UML, Objects, Frameworks, and Patterns [19].
- Pattern-Oriented Software Architecture: Patterns for Concurrent and Network Objects [18].

4.2.2.1 Analysis classes

The analysis classes form a conceptual model of the system. They represent system entities that have responsibility and behavior. These classes later evolve into subsystems and classes in the Design model. According to RUP an analysis class may be stereotyped as one of the following [2]:


Boundary class


Entity class


Control class

Boundary classes represent the high-level interfaces to the systems surroundings. They are the only classes that should have to be changed when e.g. a GUI or a communication protocol is changed.

The entity classes model information and associated behavior that must be stored. They are, just as the control objects, independent of the environment of the system.

Control classes are used to model control behavior specific to one or a few use-cases. Their behavior is of the coordinating type; they often control other objects.

These stereotypes are not part of the UML Core, but are a part of the UML Extension [1].

4.2.2.2 Analysis roles

Since Rational Rose-RT uses UML-RT that is based on ROOM, the tool is suited for analysis with analysis roles instead of analysis classes. The stereotypes for the analysis classes <<entity>>, <<boundary>>, and <<control>> are not standard in Rational Rose-RT, nor are the icons for these stereotypes. The possibility to create new stereotypes and icons in Rational Rose-RT exists, if analysis with analysis classes instead of roles is desirable.

The analysis with roles and the analysis with classes are very similar. We had a hard time finding guidelines and explanations on how to proceed when finding analysis roles. RUP makes no difference in the analysis depending on whether Rational Rose or Rose-RT is used. According to RUP the analysis is always done with analysis classes. The Online Help in Rational Rose-RT does not contain any help regarding the analysis work. It only states that there is no clear distinction between analysis and design in the Rational Rose-RT toolkit. In a Student Manual from Rational University on how to develop real-time software with Rational Rose-RT [13] we found a section about Roles in the chapter "Mapping Requirements to Design". They identify Roles in connection to the Use Case realizations and create Collaboration and Sequence diagrams using these Roles. They also give the following information and guidelines about Roles:

- During initial analysis it is very useful to think about the implementation objects in a very abstract way, we do this using roles.
- The complete behavior of a Use Case has to be distributed to roles.
- To avoid confusion over exactly what a role is, think of them as being objects that we are using to discover the design of the system.
- Roles will eventually map down into design Capsules (most likely) or Design Classes.

- Roles come from actors and Use Cases in the Use Case model and the entities that are described in the Use Case supplementary specifications.

The identification and the use of analysis roles are very similar to the identification and the use of analysis classes. In addition the Student Manual also mentions a technique for identifying roles by using analysis classes (i.e. <<entity>>, <<boundary>> and <<control>>) and says that the analysis classes discovered in the use cases will map directly down into roles.

4.2.2.3 Use Case realizations

When a candidate architecture and a set of analysis classes have been defined, the behavior of the Use-Cases must be distributed to the analysis classes. The different flows of events of the Use Cases and the interaction between the analysis classes during these scenarios should according to RUP be illustrated in collaboration diagrams. The structure of the system in the scenarios is clearer in a collaboration diagram, but sequences are harder to follow since the reader must visually hunt for the next message. Therefore it can be a good idea to complement the collaboration diagrams with sequence diagrams. If either a Sequence or a Collaboration diagram have been made in Rational Rose the tool can generate the other, so no extra work is needed. Rational Rose-RT has according to the online help the ability to show the sequence in a scenario on the Collaboration diagram (by sequence numbers) but we never got this option to work in our version of Rational Rose-RT.

Since Collaboration diagrams do not show time as a separate dimension, the sequence of interactions and the concurrent threads must be described using sequence numbering. Sometimes it can be convenient to use the sequence numbering in the Sequence diagrams too, even though it has a time axis. It can, for example, make it easier to see which message/interaction that corresponds to which, when comparing Collaboration and Sequence diagrams.

The labels of the message/interaction arrows in the UML standard [1] do not only contain the sequence numbering, they also contain information about the message being sent, its arguments and return values, guard conditions, iteration, branching, concurrency and synchronization. The labels have the following syntax:

Syntax: **predecessor/ [guard-condition] sequence-expression: return-value := message-name argument-list**

Example V: **A3,B4/ [limit_X>limit_Y] C3.1 : status := update()**

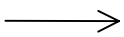
Meaning, the message "update" will be sent (with no arguments) if limit_X is larger than limit_Y. The result is stored in "status". The sequence number of this message is C3.1 where the C is the name of the concurrent thread. The message may not be sent until message 3 in thread A and message 4 in thread B have been sent (synchronization).

Neither Rational Rose nor Rose-RT have full support for this syntax, why we will not here further explain the different terms in the label. The description can be found in the OMG UML specification [1] (available on the OMG home page).

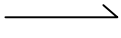
The message labels in Rational Rose and Rose-RT only show sequence numbering, the message name and arguments. In addition, the sequence numbers do not contain a thread name (e.g. A, B, C), which makes it hard to show concurrent threads in a clear an unambiguous way.

Since Rose and Rose-RT cannot show concurrency in the sequence numbering (with letters) the use of the different message stereotypes are of great importance. The UML Core provides the following types:

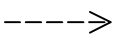
Synchronous: Nested flow of control, typically implemented as a call to a method/operation. The entire nested sequence is completed before the outer level sequence resumes.



Simple: Flat flow of control. Each arrow shows the progression to the next step in sequence. Normally all of these messages are asynchronous. This message type is often used during the requirement workflow and in early analysis when details about the communication are not know or not considered relevant in the diagram, but this use is not explicitly expressed in the OMG UML specification[1].



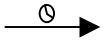
Asynchronous: Asynchronous flow of control. The sender does not wait for a return, but continues to execute after sending the message.



Return: Return from a procedural call. The return arrow may be suppressed since it is implicit in the end of an activation.

In addition the UML Extension provides other kinds of message types:

Balking: If the receiver of the message is not immediately ready to accept the message, the sender aborts the message and continues.



Time-out: The sender waits for the receiver to be ready for the message up to some fixed period of time before aborting the message and continuing.

Unfortunately Rational Rose and Rose-RT do not completely agree with the UML standard when it comes to messages, sequence

numbering and the use of Focus of Control. There are also a few differences between the two tools.

Rational Rose

Rational Rose has all these types of messages (see Figure 28). The definition of the Simple message type is however slightly different. Rational Rose's definition is:

→ *Simple*: For messages with a single thread of control, one object sends a message to a passive object.

In addition the synchronous messages have a slightly different icon.

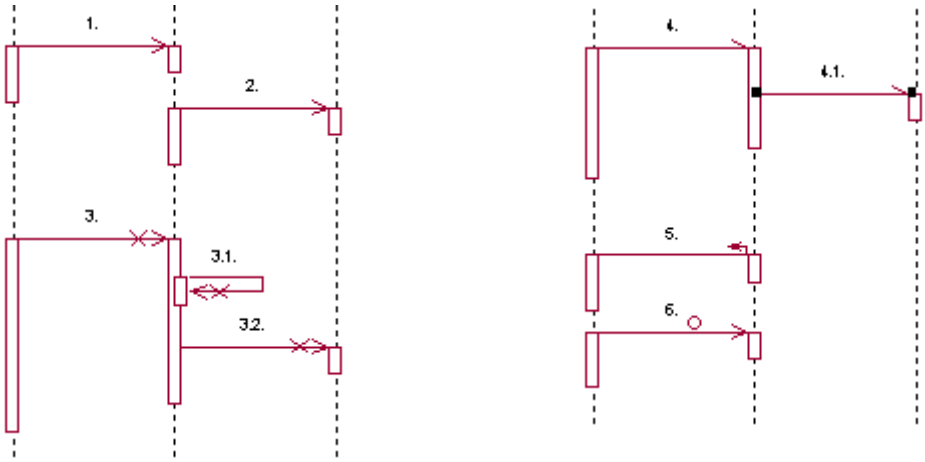


Figure 28. Sequence diagram from Rational Rose showing simple messages (1 & 2), synchronous messages (3, 3.1 & 3.2), asynchronous messages (4 & 4.1), a balking message (5) and a time-out message (6).

The returns from synchronous messages are not added automatically in Rose. These must be added by hand if they should be show explicitly.

As shown in Figure 28, Rational Rose supports hierarchical sequence numbering where a new level is added to the numbers whenever a nested call is made (3 & 3.1).

Rational Rose can also specify that a message should be sent periodically.

The sequence numbers in Rational Rose do not contain thread names (e.g. A, B, C), which is a great disadvantage when modeling concurrency.

Rational Rose-RT

Rational Rose-RT only has three different sorts of messages:

↘ *Asynchronous send*: Used when a Capsule sends an asynchronous message to another Capsule.

Synchronous send: Used when a Capsule sends a synchronous call to another Capsule.

Synchronous call: Used when a call to a (passive) class is made.

Figure 29 shows examples of the different message types. In Rational Rose-RT the return message is always shown when a synchronous message is sent.

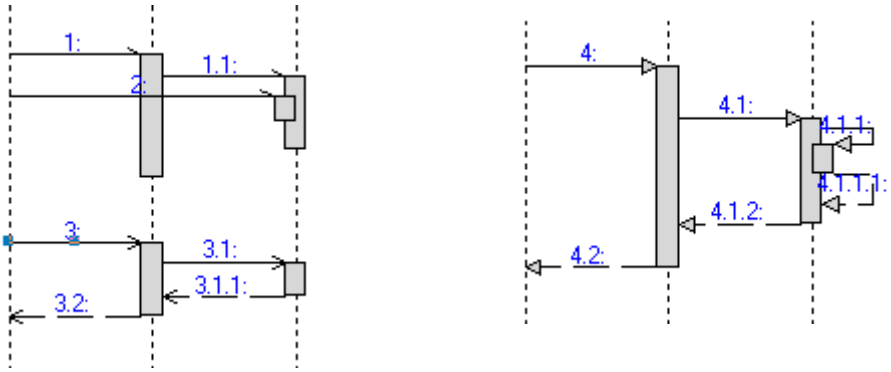


Figure 29. Sequence diagram from Rational Rose-RT showing the different message types and the sequence numbering.

As mentioned earlier, Co-regions can be used in Rational Rose-RT to obtain thread names (see Figure 23).

Guideline 16:	**Use Co-Regions to obtain thread identifiers in Sequence diagrams in Rational Rose-RT.**
Guideline 17:	**Use Collaboration diagrams with thread identifiers in order to model the structure of the thread mechanisms.**

Focus of Control and nested flows in Rational Rose and Rose-RT

A comparison between the Focus of Control in Rational Rose and Rose-RT will show that they differ. One difference is that Rational Rose-RT does not show Focus of Control on the lifeline of the sending object. Another difference appears in connection with asynchronous messages and nested flow (see Figure 30).

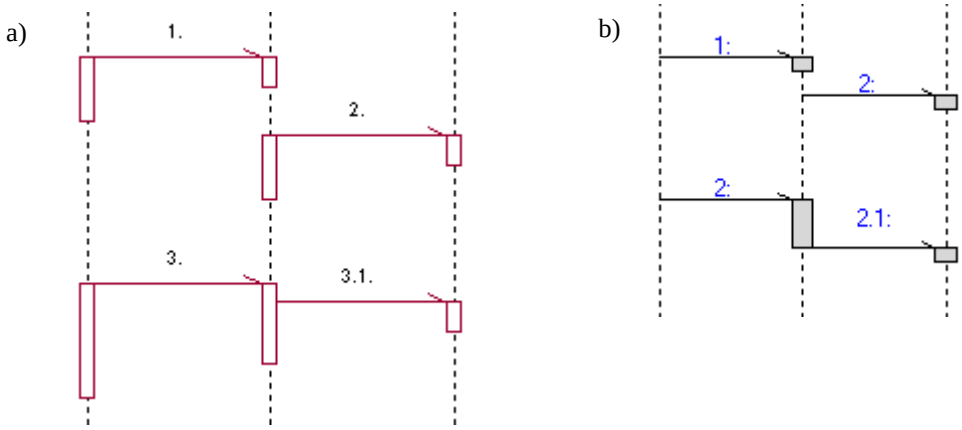


Figure 30. a) Sequence diagram from Rational Rose showing asynchronous messages with flat flow of control (1 & 2) and with nested flow of control (3 & 3.1). b) Sequence diagram from Rational Rose-RT showing asynchronous messages with flat flow of control (1 & 2) and with nested flow of control (2 & 2.1).

From the use of Focus of Control and nested flow it seems like Rational Rose and Rose-RT have two different definitions of nested flow:

A. When the sender sends a message with nested flow the sender must wait for the receiver to finish before continuing (same as synchronous synchronization). This definition would make it incorrect to use nested flow in connection to asynchronous messages (messages 3 and 3.1 in Figure 30a).

B. When two messages are sent with nested flow of control, it means that the second message is sent as a response to the first message. The sender is not forced to wait for the receiver to finish (unless the communication is synchronous). This would mean that it is allowed to use nested flow of control in connection with asynchronous messages (message 2 and 2.1 in Figure 30b).

The OMG UML specification does not explicitly provide a definition of nested flow of control, but when describing the notation of the different message types (provided above) it seems as if they use definition A. In the Rational Rose-RT Exercise WorkBook from Rational [14], definition B is used.

Guideline 18:	**When using Rose, use definition A of nested flow of control, i.e. do not use nested flow in connection with asynchronous messages.**
Guideline 19:	**When using Rational Rose-RT, use definition B as that definition better correspond to the sequence diagrams generated by Rational Rose-RT during test of the system.**

4.2.3 Design Elements, Use Case Design, Distribution and Run-Time Architecture

The analysis classes and analysis roles, in turn, evolve into a number of different design elements. These design elements are found by further analyzing the interaction between the analysis classes. During the Use Case design interactions in the Use Case realizations are refined together with the requirements on the operations of classes, subsystems and Capsules. If a distributed system is modeled, a description of how the functionality of the system is distributed across physical nodes also must be added. When considering the run-time architecture one should among others identify processes, inter-process communication (thread synchronization) and process life cycles, analyze resource sharing and distribute model elements among processes.

The design elements that are the main focus in a real-time system is of course Active classes or Capsules. RUP's advice is to consider Capsules whenever there is concurrency in the problem domain. To do this, it is expected that Rational Rose-RT is used as design tool. In Rational Rose, concurrency is modeled with classes stereotyped as <<active>>.

Figure 31. a) Active class from Rational Rose. b) Capsule from Rational Rose-RT.

A control class from the Analysis model often evolves into one or several active classes.

4.2.3.1 Designing with Active classes

An active object owns a thread of control and may initiate activity. Processes and tasks are traditional kinds of active objects. A role for an active object is shown as a rectangle with heavy border. It can also be indicated by the stereotype <<active>>. In Rational Rose concurrency of a class is set to active in the *Class Specification,* but this do not automatically give the class the <<active>> stereotype, the stereotype has to be set manually.

Active objects are often composites, i.e. one object contains one or several other objects (components) (see Figure 32). These components can be seen as aggregations by value. The composite object is responsible for the creation and destruction of its component objects. The concurrency within an active object can be distributed

completely to its components, where each component[19] is responsible for one thread of control.

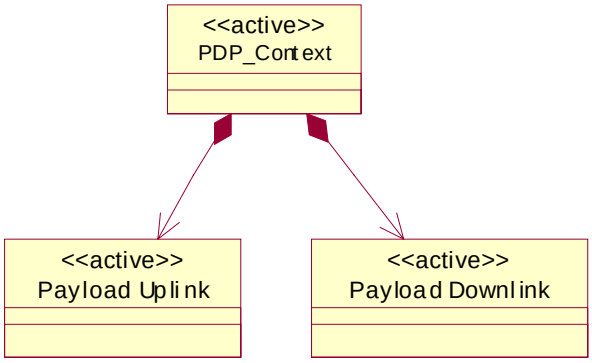


Figure 32. Class diagram from Rose showing an active class "PDP_Context" with the composite objects "Payload Uplink" and Payload Downlink".

4.2.3.2 Designing with Capsules

Capsules are Rational Rose-RT's correspondence to active classes. Each Capsule is described by a state machine and is hence reactive.

Concurrency within a Capsule is completely distributed to its components, Subcapsules. This means that a Capsule's state machine cannot contain concurrent states, instead Subcapsules that have their own state machines handle concurrent flows of control. We will try to elucidate this with an example:

Example VI: **Figure 33 shows two freestanding Capsules, *CapsuleA* and *CapsuleB*. The concurrent behavior of *CapsuleA* is distributed to *CapsuleA1* and *CapsuleA2,* which are Subcapsules to *CapsuleA*. The behavior of the two roles *CapsuleA1Role1* and *CapsuleA1Role2* are identical since they are specified by identical instances of the state machine of CapsuleA1.**

[19] This component can in turn be a composite object with several threads of control distributed to components.

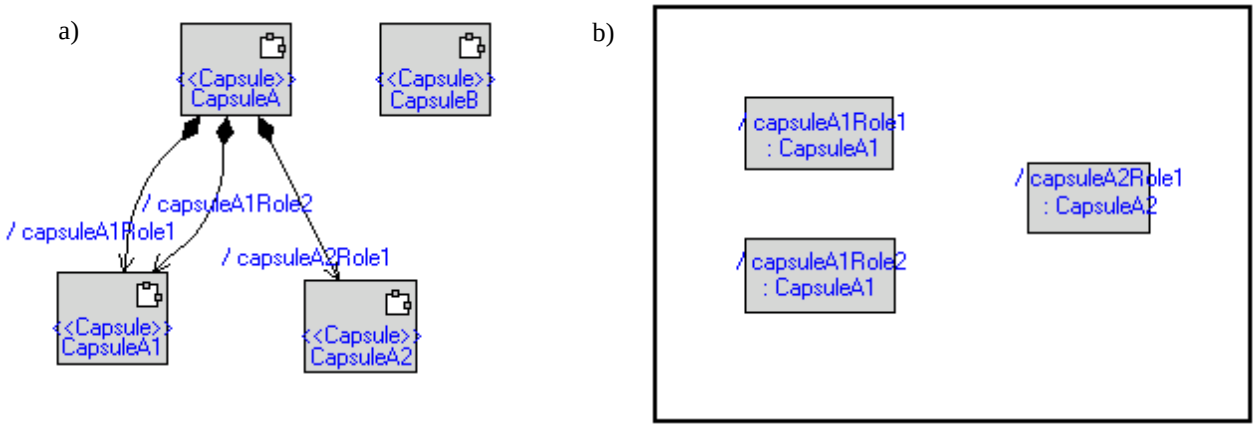


Figure 33. a) Class diagram from Rational Rose-RT containing two freestanding Capsules *CapsuleA* and *CapsuleB*. *CapsuleA1* and *CapsuleA2* are Subcapsules to *CapsuleA*. b) Structure diagram from Rational Rose-RT showing the internal structure of CapsuleA (no ports have been added).

From Use Case state machines to Capsules and Protocols

As mentioned earlier, finding Capsules from the analysis roles is a difficult process and the advises and guidelines on how to proceed are limited. Another approach that we found very effective and that simplified the development of Capsules was to focus on the state machines in the Use Case model and develop the Capsules top-down. The development starts with one main Capsule. Since Capsules can only encapsulate one thread of control, a new independent Capsule or a Subcapsule is needed whenever concurrent activities must be performed. The approach is exemplified in Figure 34 and it shows the Use Case "GGSN Startup" for GGSN-Light. The main Capsule corresponds to the entire system "GGSN". Concurrent activities must be performed during the startup of the subsystems. This responsibility is hence placed on two Subcapsules in "GGSN". These two Subcapsules each have their own state machine that specifies what must be done during the startup of the subsystems.

The functionality of the system is then extended through several iterations. In the early iterations the Protocols will be on a high level and will only define events or messages. In later iterations they will evolve into more concrete Protocols that define signals and data.

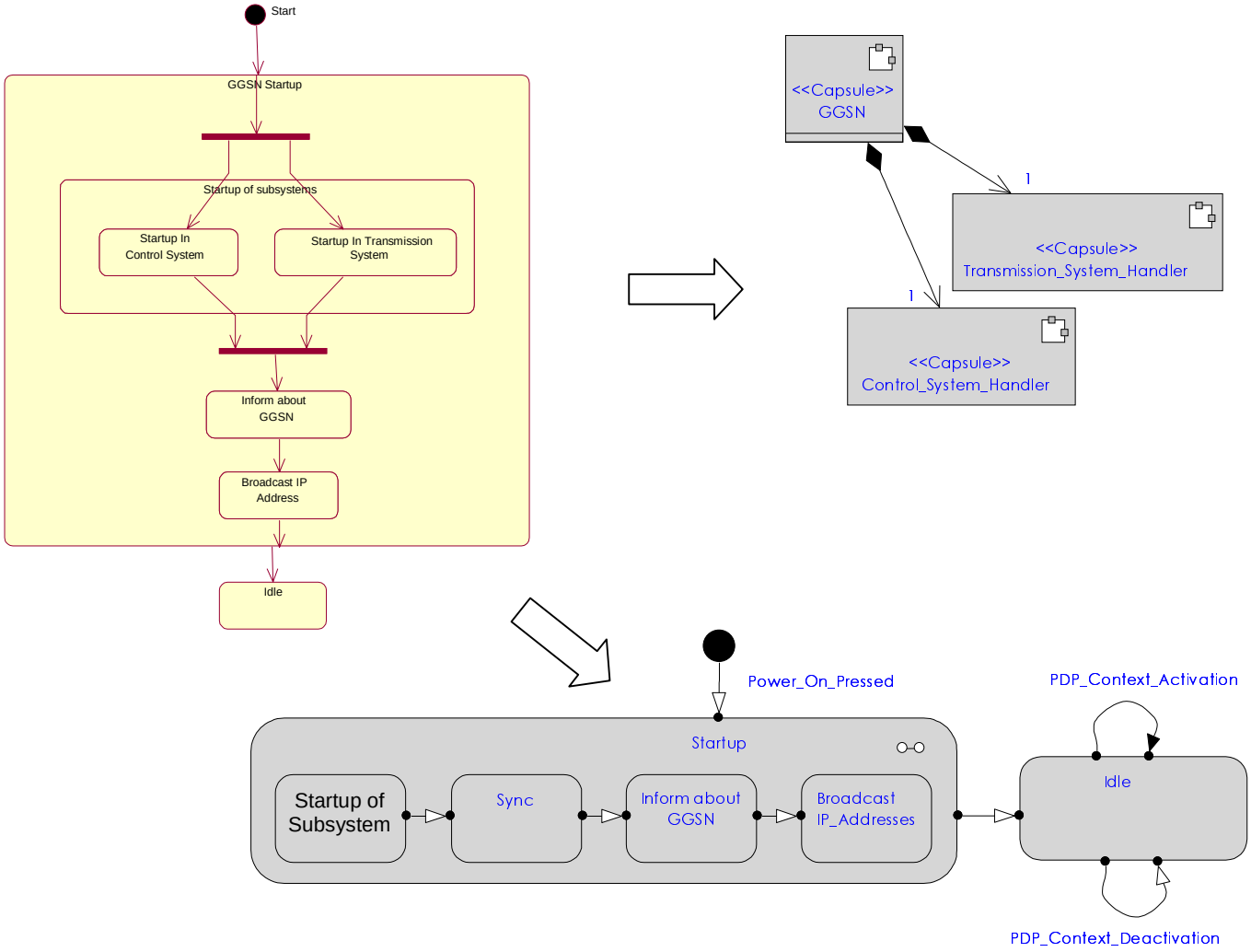


Figure 34. To the right: The state machines for Use Case "Start_GGSN". To the left: The corresponding Capsule of the Use Case and the state machine for Capsule "GGSN".

The state machine in Figure 34 is simplified and shows only the significant behavior of GGSN-Light, and not the behavior of its aggregated roles.

4.2.3.3 Object creation and destruction

As mentioned earlier a composite class is responsible for the creation and destruction of its components. Commonly all components are created/destroyed at the same time as the composite class itself is created/destroyed, but this is not always the case. Rational Rose-RT can, for each component, choose if it shall be fixed, optional, or a plug-in.

Fixed: One (or more, depending on the cardinality) instance of the Capsule is automatically created when the container Capsule is initialized.

Optional: The container class must explicitly instantiate the Capsule role within the detailed code of the container Capsule state machine.

Plug-In: A Capsule role is never directly instantiated, but rather an already existing instantiation from another Capsule decomposition is imported into the role. That is, an existing Capsule is dynamically "plugged in" to the specified role under the program control of the container class. The container class state machine must explicitly request the plug-in of a Capsule at run-time within the detailed code.

In Rational Rose-RT's Sequence diagrams special types of messages show the creation and destruction of objects/instances. When dynamically creating instances the lifelines of the created instances begin further down on the Sequence diagram. A cross visualizes the destruction of an instance.

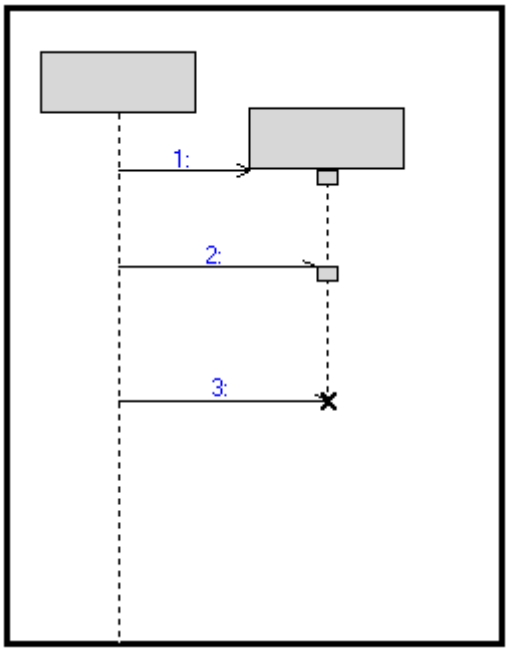


Figure 35. Sequence diagram from Rational Rose-RT showing the dynamic creation (1) and destruction (3) of a Capsule Role.

4.2.3.4 Thread synchronization

Resource sharing can be seen as a case of the more general problem of thread synchronization. We already introduced the most common strategies for synchronization (asynchronous, synchronous, balking and timed-waiting) when we described the different message types in Chapter 4.2.2.

When executing a Rational Rose-RT model a Service Layer is used between the model and the operating system. This layer handles the messaging and the queues for messages and events. It is also responsible for the startup of the execution of the Capsules state

machines. When an event triggers a sequence in a Capsule this sequence is executed without interruptions (run-to-completion). Events that are generated during the execution of the sequence are queued up in the Service Layer. Together with the fact, that ports are the only interface to the Capsules environment, this gives the Capsules a built-in semantic for mutual exclusion. Encapsulating them in Capsules can hence automatically guard resources.

When designing with standard UML, mutual exclusion must be explicitly designed, e.g. by the use of specific patterns. There are a lot of different design patters that solves the problems with thread synchronization and resource sharing available in today's literature. Some examples on design patterns that could be helpful when working with thread synchronization are:

- Design Patterns: Elements of Reusable Object-Oriented Software, [18].
- "Strategized Locking", [22]. Parameterizes synchronization mechanisms that protect a component's critical sections from concurrent access.
- "Rendezvous Pattern", [5] & [19], that allows synchronization of concurrent tasks.

An example of a state pattern is the "Barrier State Pattern" [19] that handles thread synchronization. If the underlying ROTS do not support them, they can be constructed with the help of this pattern. The pattern is used in Figure 34 for synchronization of the two threads: "Transmission_System_Handler" and "Control_System_Handler". The figure below shows how the pattern works when three threads are synchronized.

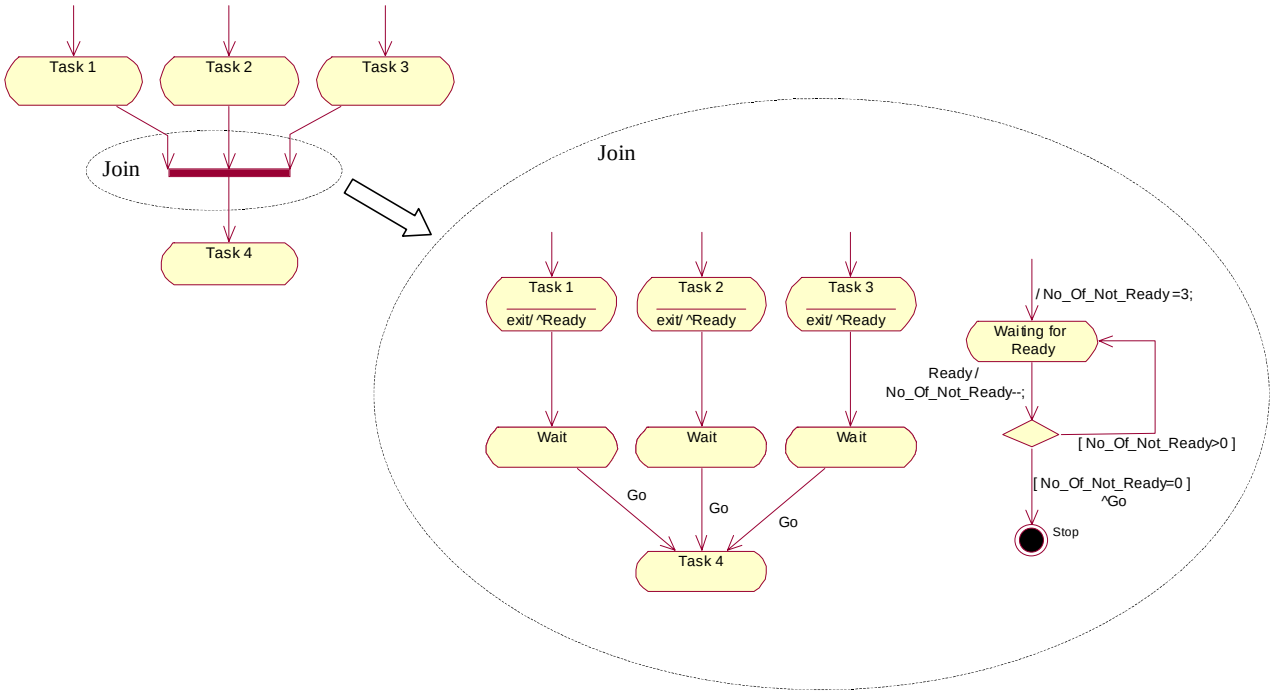


Figure 36. Construction of a Barrier for Thread Synchronization of 3 different threads.

4.2.3.5 Scheduling

The scheduling is done by the operating system, but can be controlled by the developer through priorities and by setting parameters of the scheduling algorithm. Rational Rose-RT gives the possibility to specify the priorities of the messages.

4.2.3.6 Distribution

Real-time systems are often distributed. Deployment diagrams are used to model the distribution of processes, components and devises to processing nodes. The deployment diagram in turn uses components from a Component diagram that shows compiler and run-time dependencies between software components.

4.3 IMPLEMENTATION WORKFLOW

The implementation of the model can be done either manually or by code generation.

Using standard UML, manual implementation is easy in the sense that classes, attributes and methods map directly to classes, attributes and methods in common object oriented programming languages. Implementing a UML-RT model manually, on the other hand, is significantly more complex because of the complex realization of

Capsules, Ports and Protocols. Information on how this realization can be done is available in [6].

Rational Rose and Rose-RT both support generation of code, but not to the same extent. In Rose, classes, attributes, and aggregations, among others, are generated. The methods are produced as skeletons that have to be completed by manual programming in the source code files. In Rational Rose-RT a complete implementation is generated. Implementation details are still necessary, but they are added directly within the model framework and not in the source code files.

4.4 TEST WORKFLOW

The advantage of Rational Rose-RT in the test workflow is that the model can be executed and verified within the tool. Monitors of the Capsule instances state machine and structure are available, which makes it possible to follow the state transitions and dynamic changes in the structure during the execution. These possibilities are not available in Rational Rose.

5 HOW WILL RUP BE AFFECTED

As mentioned earlier RUP has some limitations when it comes to modeling of real-time systems. One of the main limitations is the choice in the analysis and design workflow, where designers have to choose between "Design Components" and "Design Real-Time Components", see Figure 37. The guards at this choice are [Non Real-Time] and [Real-Time], but RUP does not provide any definition or explanation of what is considered to be Real-Time. In addition, RUP does not explain how to model real-time systems without Capsules and Protocols.

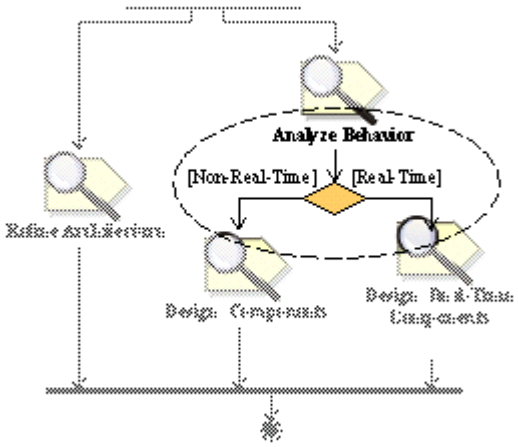


Figure 37. Part of the analysis and design workflow from the ERV RUP adaptation with the choice between "Design Components" and "Design Real-Time Components".

During our modeling work with the GGSN-Light we have also found that not all stages in the RUP process are fully adapted to the development of Capsules and Protocols. In the Use Case and Analysis model, for example, activities and artifacts can be added and existing activities and artifacts can be modified in order to make the development of Capsules and Protocols easier and more effective.

We here present the possibilities (that we have discovered during our modeling work) to modify and enlarge parts of the RUP workflows in order to improve the modeling of real-time systems. The activities and artifacts that have been added or modified are listed in chapter 5.4. Apart from this, the analysis and design workflow in this "new" model contains three different tracks shown in Figure 38, marked with broad, gray arrows. The figure also shows the points of choice in the three different workflows, guarded by guards, surrounded by [ ][20].

Track 3 is suited for systems with high degree of concurrency, systems in need of an early evaluation or prototyping. The prototyping can be

[20] [ ] surrounding the guard condition, **[guard condition]**, is OCL standard.

performed in parallel with or prior to the more careful modeling of the system (track 1 and 2) or can for small systems and systems that are not in need of a deep analysis be used instead of track 1 and 2. Track 1 and 2 include a more exhaustive and well-defined analysis than track 3. Furthermore, track 1 uses standard UML only, while track 2 and 3 includes UML-RT. Track 3 is a distinct top-down, "state machine driven" development process, where testing with the help of Rational Rose-RT can be started at an early stage of the process.

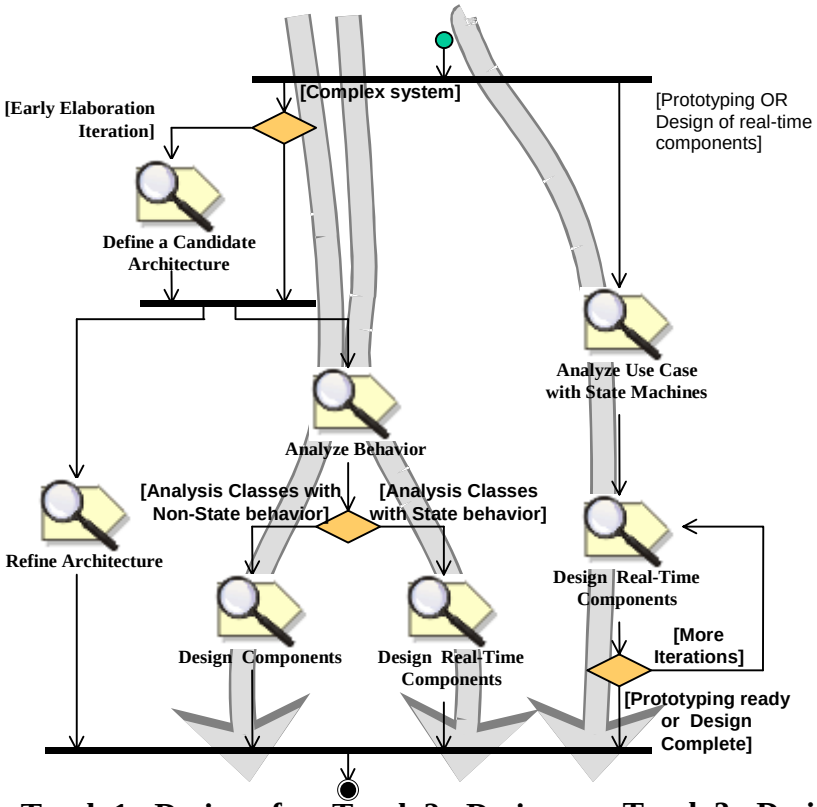


Figure 38. The three different tracks presented and discussed in this report.

Chapters 5.1 to 5.3 give a presentation of the three tracks and provide checklists for the different guards in the tracks. The checklists list important issues that must be taken under consideration when entering the track in question. The checklists contain aspects that speak in favor or against choosing the topical path. Most systems and projects contain aspects from both categories. In these cases the developers have to compare the different aspects and decide which are the most important.

5.1 TRACK 1: DEEP ANALYSIS AND DESIGN OF NON-REAL-TIME COMPONENTS

This track corresponds to the process mostly used by companies that follow RUP today. The development process uses only standard UML and does not include the concept of Capsules and Protocols from UML-RT. The process produces a design model where control in the system is represented by classes stereotyped as <<active>>. Methods and attributes of the classes in this model map clearly to methods and attributes in the source code, which makes the model easy to implement even if the code is not automatically generated (see Implementation, 4.3).

There are two guards along this track:

1. [Complex system]
2. [Analysis Classes without State behavior]

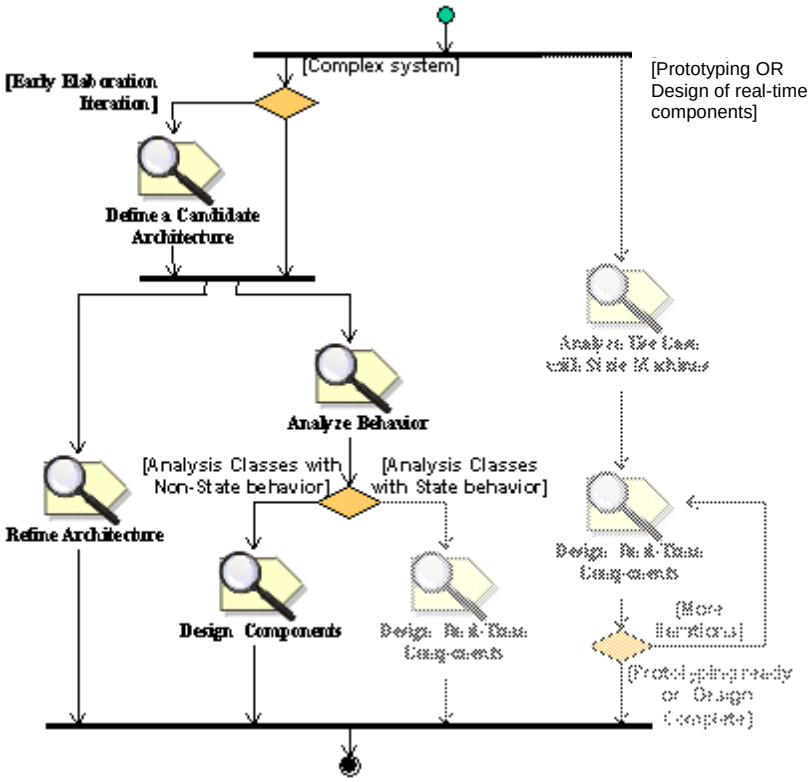


Figure 39. The analysis and design workflow where behavior is analyzed with Analysis classes or Roles and design is done with classes. This is track 1 in Figure 38.

The first guard, [Complex system], has been inserted into the workflow so that developers of prototypes and smaller systems can choose to take track 3 only, i.e. to skip the more exhaustive analysis that comes with track 1 and 2. This is of course seldom the case for real-time systems and we think developers of complex real-time systems should

always choose track 1 or 2, but they can also choose track 3 in parallel with track 1 or 2 for prototyping, early tests or to make the design of real-time components (Capsules) easier and more straightforward.

The path that follows track 1 and 2 involves an exhaustive analysis, which is often needed for large complex systems. The analysis here is clearly defined and has detailed guidelines in RUP. The path leaves the choice between automatic code generation and manual implementation open, but if the project because of certain limitation cannot generate code (e.g. because of the programming languages supported) then it is strongly recommended to choose this path. This path is the only possible if you want to completely follow Standard UML or do not want to put work into learning about UML-RT and the Capsule and Protocol concept.

The following checklist summarizes these important issues:

Checklist: Guard [Complex system]

This is the only possible path (track 1 & 2) if:
- ❑ you want to completely follow standard UML. (NB: This will further on lead you into track 1)
- ❑ you do not want to put any work into learning about the concept of Capsules and Protocols. (NB: This will further on lead you into track 1).

Aspects that speak in favor of this path (track 1 & 2):
- ❑ the system is in need of deep and exhaustive analysis
- ❑ you are developing a non-prototype model
- ❑ you cannot afford the reduction in the performance that the Rational Rose-RT Service Library involves. (NB: This will further on lead you into track 1)

You can choose this path (track 1& 2):
- ❑ whether or not you are designing a system with a high degree of concurrency
- ❑ whether or not you want to generate code automatically
- ❑ whether or not you have the need to strictly specify the possible communication between objects in the system

You should not choose this path (track 1& 2):
- ❑ if you only want to explore the system
- ❑ if you want an early evaluation of the system

The second guard, [Analysis Classes without State behavior], separates track 1 from track 2. The guard-name "Analysis Classes without State behavior" does not imply that it is impossible to model systems with state behavior in track 1. To achieve the best results,

systems without state behavior should be modeled by following track 1 and systems with state behavior should be modeled by following track 2. But if the project because of other restrictions is forced to take track 1, it is still possible even if the system has state behavior.

Taking track 1, the system will completely follow Standard UML and control in the system will be modeled by classes stereotyped as <<active>>. The concepts from UML-RT will not be used. If you do not want or cannot use automatic code generation this track is recommended. Manual implementation is easy in the sense that classes, attributes and methods map directly to classes, attributes and methods in common object oriented programming languages (as mentioned in chapter 4.3). Since UML-RT strictly specifies the possible communication between objects you should rather take track 2 if this is important and nothing else prevents you from taking this track.

The following checklist summarizes these important issues:

Checklist: Guard [Analysis Classes without State behavior]

This is the only possible path (track 1) if:
- ❑ you want to completely follow standard UML
- ❑ you do not want to put any work into learning about the concept of UML-RT.

Aspects that speak in favor of this path (track 1):
- ❑ the significant Analysis Classes/Roles of the system reacts on calls independent of the condition or state of the Analysis Class/Role
- ❑ you do not want to generate code automatically.

Aspects that speak against this path (track 1):
- ❑ you have the need to strictly specify the possible communication between objects in the system

5.2 TRACK 2: DEEP ANALYSIS AND DESIGN OF REAL-TIME COMPONENTS

This development track uses UML-RT (Capsules and Protocols) and is suited for complex systems with a high degree of concurrency. It contains an exhaustive analysis (just as track 1) that can be performed with analysis classes or analysis roles.

There are two guards along this track:

1. [Complex system]
2. [Analysis Classes with State behavior]

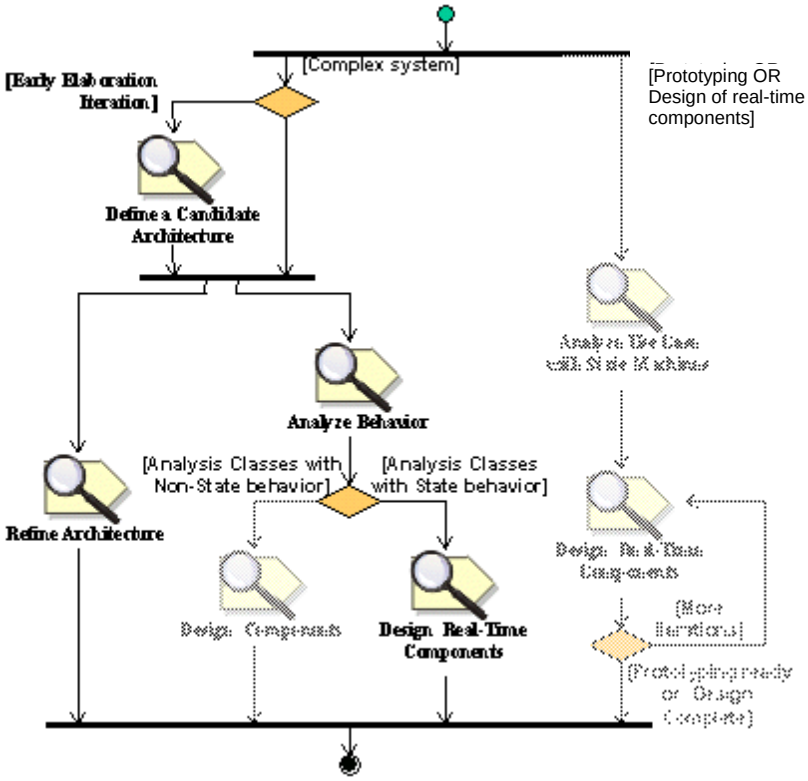


Figure 40. The analysis and design process workflow where analysis is done with Roles or Analysis Classes and design is done with Capsules and Protocols. This is Track 2 in Figure 38.

The first guard, [Complex system], is the same as for track 1 and is explained in chapter 5.1.

The second guard, [Analysis Classes with State behavior], separates track 2 from track 1. Track 2 should be followed when the system has a high degree of concurrency and important Analysis classes have state behavior. Since the communication between Capsules is strictly specified (see chapter 4.3) and they have a built-in mutual exclusion semantics, track 2 is recommended if you want to take advantage of this. As mentioned in the previous chapter it can be better to take track 1 if you do not want to or cannot generate code automatically, since it is very complicated to implement Capsules and Protocols manually.

Capsules have run-to-completion semantics, which means that when an event is received, it is completely processed regardless of the number or priority of other events arriving (see chapter 4.2.3.4). In addition, the implementation uses a Service Library that includes some reductions in the performance of the system. Both of these properties must be regarded when choosing track 2.

The following checklist summarizes these important issues:

Checklist: Guard [Analysis Classes with State behavior]

Aspects that speak in favor of this path (track 2):
- ❑ you are designing a system with a high degree of concurrency
- ❑ the significant Analysis classes/Roles of the system have state behavior, i.e. the system's reaction on calls and events depends on the condition or state of the system
- ❑ you have the need to strictly specify the possible communication between objects in the system
- ❑ you want built-in mutual exclusion semantics in the objects

Aspects that speak against this path (track 2):
- ❑ you do not want to automatically generate code
- ❑ you do not want to have "run-to-completion" semantics
- ❑ you cannot afford the reduction in the performance that the Service Library involve

You should not choose this path (track 2):
- ❑ if you only want to explore the system
- ❑ if you want to verify the system immediately

It is not possible to choose this path (track 2) if:
- ❑ you want to completely follow standard UML

5.3 TRACK 3: ANALYZE USE CASE WITH STATE MACHINES AND DESIGN OF REAL-TIME COMPONENTS

In track 3 Use Cases are described by state machines during analysis and result in a system containing Capsules and Protocols through a top-down development process. The process is adjusted after the abilities and limitations of Rational Rose-RT. It takes advantage of the possibility of testing in early stages of the development and takes in consideration that Rational Rose-RT has limited support for analysis. The analysis in this track is in other words not as exhaustive as in track 1 and 2. These properties of the track makes it suitable for prototyping and exploration of new systems. This track also makes the design of real-time components (Capsules) easier and more straightforward and it can therefore be good to use in parallel with track 1 and 2.

The track introduces a new activity "Analyze Use Cases with State machines" described in chapter 5.4.2.

The only guard along this track is:

1. [Prototyping OR Design of real-time components]

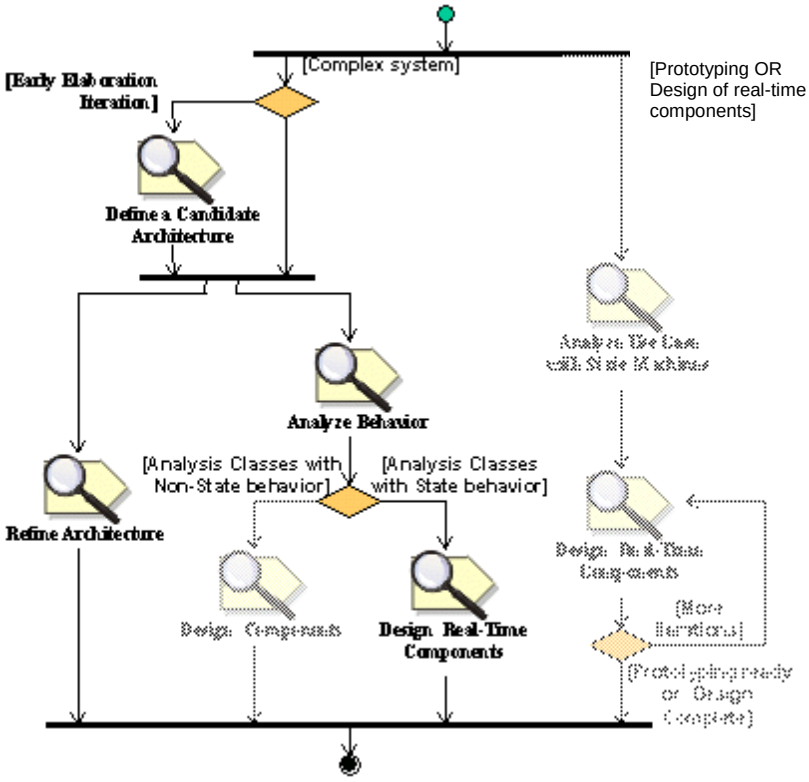


Figure 41. The analysis and design workflow where analysis is done with state machines and design with Capsules and Protocols directly from the state machines in the Use Case model. This is Track 3 in Figure 38.

The point of start will be to analyze each Use Case in order to find out its parallel behavior and its relations to other Use Cases; this is described more in 4.1.1.3. Then each state machine is assigned to a Capsule or Capsule role, this is described more in chapter 4.2.3.2. Starting with state machines, it is an easy process to go from Use Case to Code. It is also easy to follow the requirements from Use Case to Code.

The following checklist summarizes these important issues:

Checklist: Guard [Prototyping OR Design of real-time components]

Aspects that speak in favor of this path (track 3):
- ❑ you are modeling a simple system
- ❑ you are developing and evaluating a prototype system
- ❑ the system has state behavior, i.e. the system's reaction on calls and events depends on the condition or state of the system
- ❑ you want to be able to make tests early in the development
- ❑ you have the need to strictly specify the possible communication between objects in the system
- ❑ you want built-in mutual exclusion semantics in the objects

Aspects that speak against this path (track 3):
- ❑ the system is in need of deep and exhaustive analysis
- ❑ you are developing a non-prototype model
- ❑ you are not designing a system with a high degree of concurrency
- ❑ you do not want to have "run-to-completion" semantics
- ❑ you do not want to automatically generate code
- ❑ you do not want to put any work into learning about the concept of Capsules and Protocols
- ❑ you cannot afford the reduction in the performance that the Service Library involves

It is not possible to choose this path (track 3) if:
- ❑ you want to completely follow standard UML

This development workflow starts with the system as top-Capsule and then that Capsule gradually evolves into a prototype. After rather small efforts it is still possible to explore the problems with the system, especially architectural issues.

Guideline 20:	**During prototyping, use Capsules based on Use Case state machines to explore the system. This will give you an early understanding of architectural and domain problems.**

5.4 MODIFIED AND ADDED ACTIVITIES AND ARTIFACTS

The most extensive changes have been made to the activities in the analysis and design workflow. In the requirements workflow a few artifacts have been included in the existing activities.

5.4.1 The Requirements Workflow

In the requirements workflow, the "Detail a Use Case" activity (performed in "Refine the System Definition", see Figure 42) can be extended so that it also includes the production of Statechart diagrams for Use Cases (see 4.1.1.3).

Figure 42. The RUP activity "Detail a Use Case" that is performed in the "Refine the System Definition" in the requirements workflow.

The following guidelines can also be added to this activity:

Guideline 1:	**Use OCL to collect and visualize non-functional requirements in Use Case diagrams as OCL-expressions are unambiguous and can be used to verify the Use Cases.**
Guideline 2:	**Use Sequence diagrams to show Main and Alternative flows of Use Cases as they show the flows in a clear and unambiguous way.**
Guideline 3:	**Show no more than the system as a "black-box" together with the Use Cases' actors in the Sequence diagrams during the requirement workflow.**
Guideline 4:	**Use OCL in Use Case Sequence diagrams to model timeliness requirements like response time, deadlines, throughput and so on. They provide, among others, an aid during the verification of the Use Case.**

5.4.2 The Analysis and Design Workflow

Statechart diagrams should also be produced when making the supplementary Use Case descriptions in order to explore concurrency within and among Use Cases. This is done in the activity "Use Case Analysis" (that is performed in "Define a Candidate Architecture" and "Analyze Behavior", see Figure 43) in the analysis and design workflow.

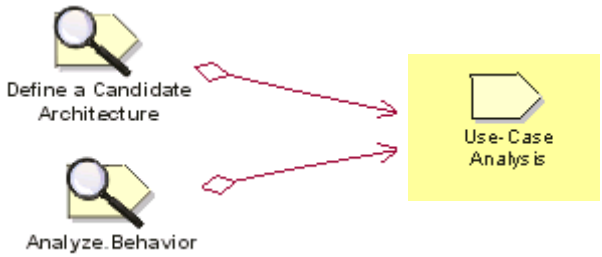


Figure 43. The RUP activity "Use Case Analysis" that is performed in the "Define a Candidate Architecture" and the "Analyze Behavior" in the analysis and design workflow.

The guidelines that concern supplementary Use Case descriptions and that should be added to this activity is:

Guideline 7:	**Sequence diagrams describing Use Case behavior should not focus on execution control. In a real-time system, dealing with concurrency, use the Sequence diagram format that contains several possible sequences in the same diagram.**
Guideline 9:	**Only show Use Case states in the Sequence diagram if the Use Case behavior also is described in Statechart diagrams.**
Guideline 10:	**For systems with a high degree of concurrency, use state machines when analyzing Use Cases in order to make the transformation from Use Case Model to Design Model easier.**
Guideline 11:	**Use state machines to specify the behavior of the system to verify the pre- and post-conditions of the different Use Cases.**
Guideline 12:	**Use Statechart and Activity diagrams to describe Use Cases in order to visualize and elaborate concurrency and dependencies within Use Cases.**
Guideline 13:	**Use Statechart and Activity diagrams to describe Use Cases in order to visualize and elaborate concurrency and dependencies among Use Cases.**
Guideline 15:	**When describing Use Case behavior with state machines in Rational Rose, use the Activity diagram as a means for finding the systems states.**

In the "Use Case Analysis" activity you also find the Analysis Classes of the system and then distribute the Use Case behavior to these classes. Since UML-RT introduces the concept of Analysis Roles, this activity should be extended with a definition of Roles and guidelines on how to best find the different Roles in the system. Unfortunately, we have not found enough information about Roles and have not been able to put enough work into the modeling with Roles, to be able to present guidelines in this area. The modeling work that we did do showed that the analysis with analysis classes and the analysis with

roles resulted in fairly similar systems. The advantage of the analysis classes was that we could clearly see which classes that where control classes, boundary classes and entity classes. When using Roles, no similar types exist. The advantage of Roles is that this is the analysis method supported by Rational Rose-RT. You can still use analysis classes in Rational Rose-RT.

Readers that are not familiar with RUP should note that the activity "Design Components" (see Figure 44) does not lead to a system without real-time components. The difference between "Design Components" and "Design Real-Time Components" is that "Design Components" does not involve the UML-RT concepts (Capsules and Protocols).

Figure 44. The RUP "Design Components" and "Design Real-Time Components" activities from the analysis and design workflow.

The following guidelines can be added to the "Describe the Run-Time Architecture" activity performed in "Refine the Architecture" in the analysis and design workflow (see Figure 45):

Guideline 16:	**Use Co-Regions to obtain thread identifiers in Sequence diagrams in Rational Rose-RT.**
Guideline 17:	**Use Collaboration diagrams with thread identifiers in order to model the structure of the thread mechanisms.**

Figure 45. The RUP "Describe the Run-Time Architecture" activity performed in "Refine the Architecture" in the analysis and design workflow.

The "Analyze Use Cases with state machines" has been added as an activity in track 3. Since this is a completely new activity in RUP we here present the steps that should be performed during the activity.

Figure 46. The new activity: "Analyze Use Case with State Machines" from the new track in the Analysis and Design workflow.

The purpose with this activity is to use the benefits of UML-RT and Rational Rose-RT. It includes an early exploration of relations and concurrency within and among the Use Cases.

The steps are:

1. Find the pre-and post conditions for the system during the Use Cases.
2. For each Use Case, find conditions for the system that should be fulfilled during the whole Use Case, let this be "Superstate" for the system during that Use Case.
3. Build a state machine for the system for each Use Case.
4. Put these state machines together into one state machine for the whole system.

Input artifact: *Use Case*.

Output artifacts: *Use Case Realization* and *Statechart diagrams* for each Use Case and for the whole system.

Worker: *System analyst*, (*Architect*).

The following guidelines can be added to this activity:

Guideline 11:	**Use state machines to specify the behavior of the system to verify the pre- and post-conditions of the different Use Cases.**
Guideline 12:	**Use Statechart and Activity diagrams to describe Use Cases in order to visualize and elaborate concurrency and dependencies within Use Cases.**
Guideline 13:	**Use Statechart and Activity diagrams to describe Use Cases in order to visualize and elaborate concurrency and dependencies among Use Cases.**
Guideline 14:	**When making Statechart diagrams for Use Cases concentrate only on the states of the entire system and not on the states of different subsystems.**
Guideline 15:	**When describing Use Case behavior with state machines in Rational Rose, use the Activity diagram as a means for finding the systems states.**
Guideline 20:	**During prototyping, use Capsules based on Use Case state machines to explore the system. This will give you an early understanding of architectural and domain problems.**

6 SUMMARY AND CONCLUSIONS

This chapter provides a summary of how UML and UML-RT can be used for modeling of real-time systems. It also includes a summary of the advantages and disadvantages of Rational Rose and Rose-RT and the modifications that can be done to RUP to make it better suited for development of real-time systems with the tools Rational Rose and Rose-RT.

6.1 UML AND UML-RT FOR MODELING OF REAL-TIME SYSTEMS

During the modeling of our evaluation model, GGSN-Light, we have found several different ways to improve the modeling work. The most powerful possibilities that UML and UML-RT provides for the modeling of real-time systems are:

- Non-functional/QoS requirements can be added to the model with the use of OCL (Object Constraint Language). OCL provides an unambiguous specification of the requirements and can also be used to verify the model. OCL can be used in all of the UML diagrams.
- Timeliness requirements are of great importance in real-time systems. Since sequence diagrams focuses on time they are, together with OCL, well suited for capturing of timeliness requirements.
- State machines for specification of Use Cases makes it possible to show concurrency within and among Use Cases. They also makes the transformation from the Use Case model to the Design model easier and the design of real-time components (Capsules) straightforward.
- In systems with a high degree of concurrency, Capsules and Protocols decreases the risk for software-errors like deadlock and sharing problems, among others. The use of ports and protocols limits the communication to the channels specified in the architectural model.
- Thread identifiers included in the sequence numbering in Sequence and Collaboration diagrams can be used to show concurrency and synchronization.

6.2 ADVANTAGES AND DISADVANTAGES OF RATIONAL ROSE AND ROSE-RT

The following table summarizes the advantages and disadvantages of Rational Rose and Rose-RT:

Table 1. Advantages and disadvantages of Rational Rose and Rose-RT.

Rational Rose	Rational Rose-RT
General	
+ Fully follows the UML standard.	– Focuses on design and is not fully suited for requirements modeling.
– Generates code that must be completed manually.	– Do not fully follow the UML standard.
+ The analysis class stereotypes <<active>>, <<boundary>> and <<entity>> are part of the tool's base stereotypes.	+ Generates complete code.
– OCL constraints cannot be directly connected to the model entities.	– You have to create the stereotypes and icons for the analysis classes yourself.
	+ OCL constraints directly in the model.
	+ Verification of the model.
Sequence Diagrams	
+ Has support for balking and time-out messages.	+ Easier to work with when it comes to messages and Focus of Control.
– Cannot show concurrency within an instance.	+ Can show concurrent activities within an instance.
– Does not use thread names.	+ Supports Co-Regions and Local actions.
	+ Messages can be drawn to/from boundary without specifying the receiver/sender.
	+ Dynamic creation/destruction of object instances is shown explicitly in the diagrams.
	+ Supports states in the sequence diagrams.
Statechart Diagrams	
+ Supports concurrency in Statechart diagrams.	– Does not support concurrency in Statechart diagrams, which is a disadvantage during requirements and analysis phases.
Activity Diagrams	
+ Supports activity diagrams with parallel actions and means for synchronization.	– Does not support activity diagrams at all.

6.3 MODIFICATIONS TO RUP

We have also made some modifications to RUP that makes it better suited for development of real-time systems with the tools Rational Rose and Rose-RT. The most important are:

- A clearer guard for the choice between "Design of Real-Time and Non-Real-Time Components.
- A new track that takes advantage of the possibilities of Rational Rose-RT. This track is suited for prototyping and early verification and minimizes the step from Use Case model to Design model.
- Checklists have been added to guide the developers in the choice between the different paths in the process.

7 DISCUSSION AND FUTURE WORK

In software development, UML combined with real-time modeling is a fairly new area. There are a lot of opportunities to explore and we will here present some possibilities to expand this work together with some reflections on our work.

7.1 REFLECTIONS ON OUR METHODOLOGY AND WORK

Overall our work has gone smoothly, much because of the good support from our supervisors at ERV and Chalmers. Unfortunately the time frame did not allow us to fully evaluate the effects of the different modeling techniques and the lack of system knowledge (of GGSN and GPRS) made it hard for us to decide which modeling technique that in the end resulted in the better system. We still feel that we have found many ways to improve the modeling techniques used for modeling of real-time systems today.

The choice to use all the different diagrams supported by UML in all the phases of the development process proved to be successful. This made it possible for us to among others explore the benefits of Statechart diagrams in the Use Case model. It also resulted in the development of "Track 3".

We developed and worked according to the process described in Figure 47.

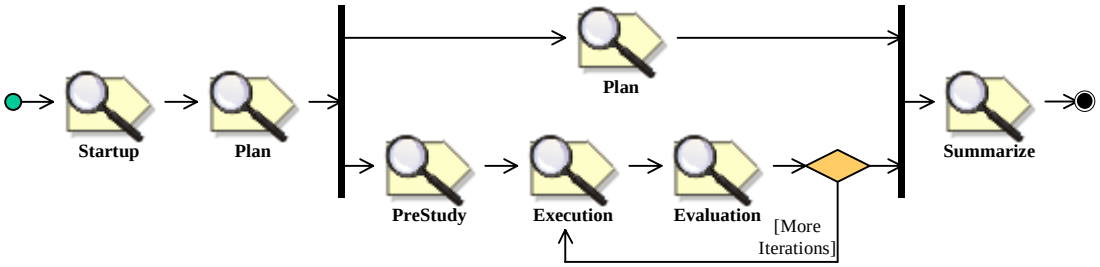


Figure 47. The process developed for and used during our thesis work.

During *startup* we got acquainted with Rational Rose, Rose-RT and other tools needed during the thesis work. We also began to look at the UML-RT concepts: Capsules, Ports and Protocols.

During *plan* we updated the general specification of the thesis work with a more detailed time plan and a description of the purpose, scoop and limitations. At first, the execution and evaluation phases were not iterative, but as the work progressed we improved the process by adding iterations to get faster feedback on our work and to get the possibility to change focus.

During the *pre-study* we defined the system that served as an evaluation model during the thesis work. It was important for ERV that the evaluation model had a nomenclature that is connected to the GSN-projects. To achieve this we took a node well known to the project, changed it and reduced it in order to get a system small enough to model that still collected significant real-time requirements. This was something that could have been done better, a lot of effort was put into the work to understand the system (GGSN and GPRS) and to adapt it. Another approach could have been to model an elevator system instead; this is a small system with a lot of real-time requirements.

During the *pre-study* we also studied different theories in relevant reports, articles and books. We studied the possibilities that UML and other languages provide and how these can be implemented in RUP and the tools used. The area (UML modeling of real-time systems) is a new area with a lot of different theories and difficulties. Therefor it was a good thing to spend a lot of time in the *pre-study* phase (about 30% of total time) and after that test and evaluate the different theories in an iterative process.

7.2 TECHNICAL ABILITIES OF DIFFERENT TOOLS

As mentioned in the introduction to this report we have focused on modeling methodology and have not considered the tools more "technical" abilities, as for example the support for reverse-engineering, round-trip, different programming languages and frameworks and the ability to interface to other tools.

LMC (Ericsson in Canada) has made an evaluation of the technical abilities of a number of visual modeling tools that are available commercially today. The results are presented in a technical report [23] (this report is only available within Ericsson). The evaluation is made with respect to how they are suited for the development of the TelOrb application developed by LMC. The technical abilities of the tools Rational Rose 2000, Rose-RT 6.1, Rhapsody 2.3, Together 4.0 and Telelogic Tau 4.0 are compared, but the report does not include a deeper comparison between the modeling possibilities in the tools.

A deeper comparison between the modeling possibilities of the different tools would be interesting to see, e.g. to what extent the tools are suited for the different development phases (requirements, analysis, design, implementation, test), if concurrency can be modeled in Sequence and Statechart diagrams, the different types of messages and sequence numbering that are used, if OCL can be directly attached to modeling entities and so on.

7.3 CODE GENERATION

We have not looked further into the aspects of code generation in Rational Rose and Rose-RT, but this will be investigated further in another thesis work at EMW (Ericsson Microwave Systems AB, Sweden). Contact Peter Ericsson at EMW for more information.

7.4 VERIFICATION OF OCL CONSTRAINTS

Another interesting area to look further into is the verification of OCL constraints. One way to do this is through the implementation of functions in Rational Rose and Rose-RT that verify constraints and invariants that have been attached to model entities.

7.5 UML IN THE FUTURE

The OMG is currently working on a series of standard concerning real-time applications [15]:

1. A standard that will address the issues of modeling time and time-related facilities and services. This includes a UML profile that defines standards paradigms of use for modeling of time, scheduability and performance related aspects.
2. A standard for modeling fault-tolerant systems.
3. A standard for modeling the architectures of complex real-time systems.

OMG has submitted a number of RFPs[21] (Request for Proposals) for the next UML version, UML 2.0. These are available on the OMG homepage www.omg.com and concern the UML infrastructure and superstructure as well as an OCL metamodel. They include important real-time issues like communication channels, internal structure of objects, the scalability and encapsulation of state machines and definition of parallel execution of interaction.

[21] With an RFP, OMG requests proposals on solutions that fulfill the users' requirements.

8 REFERENCES

[1] OMG Unified Modeling Language Specification, version 1.3

[2] Rational Unified Process 5.5 (Build 12). Rational Software Corporation.

[3] Philippe Kruchten, *"The Rational Unified Process, An Introduction"*, Addison-Wesley, 1999.

[4] Bruce Powel Douglass, "*Real-Time UML*", Addison-Wesley, 1998.

[5] Bruce Powel Douglass, "*Real-Time UML, second edition*", Addison-Wesley, 1999.

[6] Bran Selic, Garth Gullekson & Paul T. Ward, *"Real-Time Object-Oriented Modeling*", John Wiley & Sons, Inc. 1994.

[7] Student text: *"GPRS System Survey",* EN/LZT 123 5374 R1B.

[8] Cecilia Ekelin, Jan Jonsson, *"Solving Embedded System Scheduling Problems using Constraint Programming",* Department of Computer Engineering, Chalmers University of Technology, Gothenburg, Sweden.

[9] Jan Jonsson, "*Realtidsystem E, Föreläsning #1*", notes from lecture #1 in Real-Time Systems, Department of Computer Engineering, Chalmers University of Technology, Gothenburg, Sweden.

[10] Bran Selic, Jim Rumbaugh*, "Using UML for Modeling Complex Real-Time Systems"*, 1998, www.objectime.com/otl/technical/umlrt.pdf

[11] Andrew Lyons, "UML for Real-Time Overview", 1998, www.objectime.com/otl/technical/umlrt_overview.pdf

[12] Ralph Melton, Software Architecture Reading Group, 1996, www.cs.cmu.edu/afs/cs.cmu.edu/project/compose/www/sarg/feb05-06.html

[13] Rational Software, Rational University, Student Manual *"Developing Real-Time Software with Rational Rose RealTime"*, Volume 1, Version 1.0.

[14] Rational Software, Rational University, Exercise WorkBook *"Developing Real-Time Software with Rational Rose RealTime"*, Volume 1, Version 1.0.

[15] Bran Selic*, "A Generic Framework for Modeling Resources with UML"*, IEEE.

[16] Hans-Erik Eriksson, Magnus Penker, *"UML Toolkit"*, John Wiley & Sons, 1998, ISBN 0-471-19161-2.

[17] Craig Larman, *"Applying UML and Patterns"*, Prentice Hall PTR, 1998, ISBN 0-13-748880-7.

[18] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *"Design Patterns: Elements of Reusable Object-Oriented Software"*, Addison-Wesley, 1995.

[19] Bruce Powel Douglass, *"Doing Hard Time, Developing real-time systems with UML, Objects, Frameworks, and Patterns"*, Addison-Wesley, 1999.

[20] Neil Storey, *"Safety-Critical Computer Systems",* Addison-Wesley, 1996.

[21] 1/102 60-FCK 110 104, *Use Case Modeling Guidelines*, Ericsson Wide Internal Guideline, PB1, 2000.

[22] Douglas Schmidt, Michael Stal, Hans Rohnert, Frank Buschmann, *"Pattern-Oriented Software Architecture: Patterns for Concurrent and Network Objects"*, Wiley, 1999.

[23] Hung Phan, Minh-Tri Nguyen, *"Technical Report on the Selection of Visual Modelling Tools for TelOrb Application development",* LMC, 2000.