



Examensarbete MMK
2000:14 MDA 129

Investigation of Models for Real-Time Systems: AIDA through UML and ROOM

by
Erik Wyke



Stockholm
2000

M.Sc. Thesis
Mechatronics Lab
Department of Machine Design
Royal Institute of Technology, KTH
S-100 44 Stockholm Sweden

| | | |
|--|---|---------------------------|
| Mechatronics Lab Department of Machine Design Royal Institute of Technology S-100 44 Stockholm, Sweden | MSc Thesis MMK 2000:14 MDA129 | |
| | <i>Document type</i> | <i>Date</i> 2000-02-18 |
| <i>Author(s)</i> Erik Wyke | <i>Supervisor(s)</i> Martin Törngren | |
| | <i>Sponsor(s)</i> | |
| <i>Title</i> Undersökning av ett Modeller för Realtidssystem: AIDA genom UML och ROOM | | |
| <i>Abstract</i> <p>I mekatroniska system läggs alltmer funktionalitet i mjukvara istället för i hårdvara. Med mer funktionalitet i mjukvara så blir interaktionerna mellan olika funktioner viktigare. Det är möjligt att använda samma sensorer och aktuatorer från flera funktioner. Det ger även möjligheter att optimera systemen på ett nytt sätt. Reglersystem i maskiner går mot distribuerade datasystem med många sensorer och ställdon kopplat till sig. Detta ger upphov till en mängd nya design problem, t.ex. partitionering och allokering av funktionerna.</p> <p>För att kunna hantera den ökande komplexiteten i ett system så måste systemet kunna beskrivas fullständigt. Projektet AIDA syftar bland annat till att ta fram ett ramverk för att kunna göra en sådan beskrivning. För nuvarande består AIDA av en specifikation på vad ramverket bör innehålla och grafisk notation för vissa av modellerna. Ett system kan betraktas ur olika aspekter eller vyer. I AIDA är beskrivningen av ett system uppdelad i struktur och beteende för de olika domänerna: applikation, dator och mekanik.</p> <p>I den här rapporten presenteras en översikt av två existerande modelleringspråk, "Unified Modelling Language (UML)" och "Real-time Object Oriented Modelling (ROOM)", och hur de kan användas för att beskriva modellerna i AIDAs ramverk. I rapporten presenteras även andra förslag till att göra UML bättre lämpat för realtid, t.ex. UML-RT.</p> <p>Det huvudsakliga resultatet är att både UML och ROOM är bra för att beskriva struktur och händelsestyrt beteende hos ett system. Det bästa förslaget för att uttrycka krav på tidsbeteende, är att använda sekvensdiagram med tidsmarkörer på meddelanden. Dessa kan sedan användas i uttryck för kraven på tidsbeteendet. Struktur kan uttryckas med objekt diagram i UML eller med actor diagram i ROOM. Tillståndsmaskiner kan användas för att beskriva internt beteende hos ett objekt. Sekvensdiagram är bättre för att beskriva interaktionen mellan objekt.</p> | | |

Table of contents

| | |
|--|-----------|
| 1 Introduction | 1 |
| 1.1 Background | 1 |
| 1.2 M.Sc. thesis problem formulation | 3 |
| 1.3 Structure of this report | 3 |
| 2 Industrial experiences | 4 |
| 2.1 Methods, analysis and the use of models | 4 |
| 2.2 Tools used in industry | 7 |
| 2.3 The ideal tool | 7 |
| 2.4 Discussion | 7 |
| 3 Modelling of distributed Real-Time Systems and OO approaches .. | 9 |
| 3.1 Models, views and methods | 9 |
| 3.2 Object-orientation | 10 |
| 3.3 Modelling tools | 13 |
| 4 Overview of the Unified Modelling Language | 14 |
| 4.1 Use Cases | 14 |
| 4.2 Class/Object diagram | 15 |
| 4.3 Sequence diagrams | 16 |
| 4.4 Collaboration diagrams | 17 |
| 4.5 Statechart diagrams | 18 |
| 4.6 Activity diagrams | 20 |
| 4.7 Component diagrams/Deployment diagrams | 20 |
| 4.8 Extension mechanism | 21 |
| 4.9 Discussion | 22 |
| 5 Overview of Real-time Object-Oriented Modelling (ROOM)[5] | 23 |
| 5.1 Background of ROOM | 23 |
| 5.2 Actors | 24 |
| 5.3 Interface components | 25 |
| 5.4 Behaviour component | 28 |
| 5.5 Events in a ROOM model | 29 |
| 5.6 Implementation of ROOM models | 30 |
| 5.7 Discussion | 31 |
| 6 Real-Time approaches and UML | 32 |
| 6.1 Real-Time UML | 32 |
| 6.2 The ACCORD approach | 34 |
| 6.3 UML profile for Scheduling, Performance and Time | 34 |
| 6.4 ARTiSAN Real-Time studio | 35 |
| 6.5 Discussion | 39 |
| 7 A basis for model evaluation | 40 |
| 7.1 AIDA | 40 |
| 7.2 Object orientation for distributed real-time system | 42 |
| 7.3 UML for Real-Time, a combination of UML and ROOM | 43 |

| | | |
|-----------|--|-----------|
| 7.4 | Criteria for evaluation | 44 |
| 7.5 | Approach for evaluation of models | 45 |
| 8 | The AIDA framework vs. ROOM and UML | 46 |
| 8.1 | Mapping between AIDA and ROOM | 46 |
| 8.2 | Mapping between AIDA and UML | 49 |
| 8.3 | AIDA and the best of OO | 55 |
| 9 | Conclusions | 56 |
| 9.1 | The approach to the work done | 56 |
| 9.2 | Results | 56 |
| 9.3 | Suggestions for further work | 58 |
| 10 | References | 59 |

1 Introduction

1.1 Background

In traditional machine disciplines more functionality is nowadays put in software and electronics. A good description of the evolution of embedded systems, is given by automotive applications.

- Cars that are developed today carry tens of electronic control units. A very trivial example is the use of electronic window lift instead of the old-fashioned crank. An electronic window lift can be implemented as a stand-alone function. But with more and more functionality in software and electronics the interactions of the functions are more important. This makes it possible to optimize the systems overall performance[1]. Environmental aspects is one example where the companies compete about customers. New functionalities are developed to optimize fuel economy and emission reduction[4]. With more functions in software it is easier to make this optimization. For example, the mentioned window lift uses some power to fulfill its task to raise or lower the window. At the same time a more critical function, that also consumes a non-negligible amount of power, might want to execute. Then it might be useful to have some kind of power manager that turns the lift task off while executing a more important one.
- According to [4], a couple of years ago when a new function should be implemented, a specific electronic control unit (ECU) was designed for that purpose and then it was added to the car. When doing it this way the new ECU was a separate new module that did not interact with other ECUs. To reduce the cost for peripheral devices, i.e. sensors and actuators, and cabling to connect them, the ECUs were connected to a network. It is now possible to share data from sensors or to use the same actuators from different ECUs. At some point the limit for how many black boxes that can be added is reached. To deal with this problem and to reduce costs, integration of more than one function to one ECU is made.

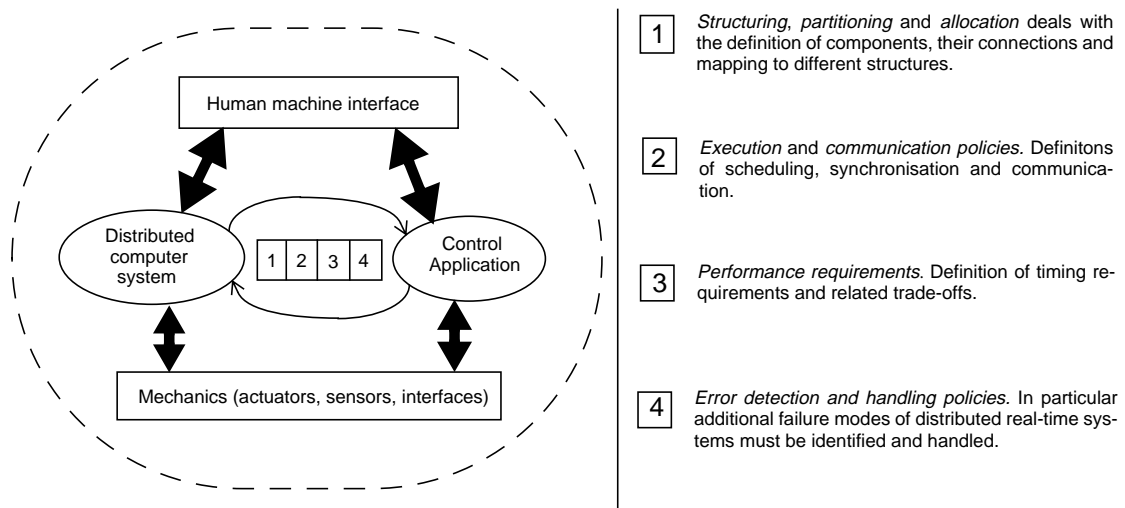


Figure 1. Design issues for distributed real-time systems

The control systems in machine designs are going towards a distributed computer system connected to a lot of actuators and sensors. This leads to some important design issues, see figure 1. It's important to bridge the gap between different engineering disciplines

such as control, mechanical and software engineering[1], because their respective discipline is interacting in some way. Some advantages of distributing an embedded control system in new machines is the cost reduction by using less cabling, hydraulics and other kinds of physical couplings between elements [1].

It might be hard to design a distributed control system if the control structure is not given, that is one part of the functionality of the system. On the other hand the implementation might cause the control strategy to change due to different factors, e.g. delays. This connection between modelling and implementation issues has to be considered as important when designing new systems or extending already existing systems. A control system can become unstable if there is too much delay introduced by some implementation choices.

Real-time embedded systems is not just an issue for distributed control systems. In [2] a set of requirements that are the same for different kinds of embedded systems, single chip based microprocessors to large complex real-time networks, is listed. Some of the points in [2] are:

- the system must function correctly.
- outputs must occur at the correct instants in time, in some cases the input needs to be sampled at the correct time as well.
- software partitioning has to be near optimum; communications overhead can be large and non-negligible.
- implementation specific factors has to be efficient; factors are such as OS choice, multitasking structures, scheduling, processor utilisation and so on.

With increased complexity, requirements on reliability and safety, economy and, as mentioned above, increased demand on environmental aspects, it is clear that the system designer need tools and methods to help him meet the requirements[1,2]. What is not so clear is how such a tool or method should look like. One approach is the work in the AIDA project and this is the focus of this thesis.

The complexity issues of distributed embedded real-time systems and the lack of tools to support design of such system is also given some attention in [25]. It is stated that partitioning of a complex system to different ECUs haven't received much attention in the discussion about tool-based development. The use of hardware-in-the-loop testing is one way of discovering problems in the design, but this may be too late in the development process.

The aim for the project *Automatic control In Distributed Application (AIDA)* [23] is to develop a modelling framework with an integrating tool-set to support the design issues pictured in figure 1. The focus is on solving a few concrete design problems. Its stated therefore to be more of a supporting modelling approach, i.e. it can be the complement to other design methods that lack the ability to take care of those design issues. The AIDA models are supposed to be used in an iterative way.

1.2 M.Sc. thesis problem formulation

Original title:

Utvecklingsverktyg för inbyggda realtidssystem: framtagande av verktygsprototyp
Development tool for embedded real-time systems: construction of tool prototype

Original goals:

- Look at the problems in constructing embedded real-time systems and find scenarios that need tool support.
- Evaluate graphical models for description of systems that supports the problems in the construction of embedded real-time systems. Specifically UML models and the AIDA framework.
- Look at suitable technique for implementation of tool prototype and its possible connection to existing commercial tools.
- Construction and evaluation of tool prototype.

Final title:

Development framework for embedded real-time systems: AIDA through UML and ROOM.

Final goals:

The goals for this thesis project have been slightly restated. The parts about the tool implementation has been suppressed. In addition to UML also ROOM, i.e. Real-time Object Oriented Modelling, has been studied.

- Look at the construction problems that exists in industry, find scenarios that needs tool support and if those matches the intents of the AIDA framework.
- Find how the AIDA frameworks fits into Object-Oriented thinking, i.e. look how AIDA can be expressed in terms of UML and/or ROOM.
- Make conclusions and propose further work.

1.3 Structure of this report

In the introduction the problems with the design of embedded distributed real-time systems is described shortly. In the next part interviews with SAAB CombiTech and the Swedish Space Corporation (SSC) where handled to give some views of how the process and the use of methods and tools is made in the industry. Then an introduction to modelling concepts and object-orientation is given. This is made so that some of the concepts treated in the chapter 4 and 5, the overviews of UML and ROOM, are more easily understood.

In chapter 6 some approaches to make UML better suited for real-time system design is presented. One of the interesting approaches is the Request for proposal from the OMG about making a UML profile for real-time issues. Chapter 7 gives an overview of the AIDA framework and in chapter 8 a mapping of AIDA to ROOM and UML is suffested. Chapter 9 covers the conclusions of the work.

2 Industrial experiences

Two interviews have been made in this thesis project to get some industrial views of the development of embedded real-time systems. These interviews were made by Martin Törngren, Di-Jiu Chen and Erik Wyke and we talked to Thomas Strömqvist at SAAB CombiTech and Gunnar Andersson at the Swedish Space Corporation (SSC). The project we all had in mind when doing these interviews was primarily satellite project called SMART, but experiences from other projects were included as well. It seems natural that Thomas and Gunnar spoke about the same project which gave us two views of the same project. These interviews are the basis for this part.

The SMART project is a bit special compared to normal projects. That is because the European Space Agency (ESA) is a quite demanding and experienced customer. ESA is forcing more analysis of the design decision. The decisions have to be motivated. Another aspect that makes this project a bit special is the politics involved. The assignment of sub-tasks has to be done to countries that are members of the EU.

In the SMART project Thomas is the software architect and Gunnar is more of a system architect with a little more focus on the hardware.

2.1 Methods, analysis and the use of models

The development process

The projects at the SSC are generally divided into three phases. Phase A is a possibility study, where a very rough cost estimation is made. If it looks good to the customer and if the reputation of SSC in this area is good then a phase B study is made. Phase B is an implementation study. A rather detailed description of the design is produced. This can consist of strategic component choices, buses, architectures and other critical parts. This leads to a quotation.

ESA has something called PSS-05 which is a development process that has to be followed. It makes the responsibility issues clear. People responsible for the different subsystems meet in the system group to discuss the system. Up to some point in time the communication in the system group is made in an informal way. Thomas has the opinion that it must be more formal meetings that forces people to talk and not just keep on working. The flow of information between the different subsystem groups has to be defined. According to Thomas, even if it is Gunnar that ties the subsystems together, much of the information goes by the software group.

In PSS-05, something called technical notes are used. They capture the informal thoughts, rough analysis and reasoning about an investigation of something. This does not put requirements on the system but it can help in taking and documenting basis for decisions. They do not need to be structured in any particular way, but they do exist in the document structure. Thomas has the opinion that this is a good way to capture informal descriptions that otherwise might get lost. Technical notes can show how something was discovered and not just what was discovered.

There are no templates for the use of models. ESA prescribes a document structure and gives a list of requirements on system level. These requirements are broken into subsystem requirements. When the subsystem architecture is done the requirements are broken

down one more time. All requirements are stored in a database. Gunnar is not fully satisfied with doing it this way.

Something that is important in this kind of projects, according to Gunnar, is that some of the people involved have quite a lot of knowledge and experience for what is needed. In early phases a rough estimation of the cost is made. This is not done so much by analysis but more by experience of this kind of projects. Another fundamental aspect of this kind of systems is that they are not mass-produced and therefore it is almost never profitable to have a large utilization of processors etc. Optimization gives longer development times. It is better to take hardware with a lot of capacity so there is a lot of reserves in the system. The information for this is based on estimations and not by analysis. There exist some hard limitations on the system, e.g. some physical parameters on radiolinks, physical size and weight. With this information and the knowledge about what antennas and what receivers that are used, a bitrate can be calculated. This dimensions the amount of memory that is needed.

It is not obvious what models that will be used in the project. A model is okay if the persons that are supposed to understand it. One of the models used in an early stage is the CAN Interface Communication Description (ICD). It describes how the communication is made between nodes, what protocol are used and what packages that are sent on the bus. This is a document that later can be implemented in executable models. Very rough estimations of loads are done at an early stage, the information for this was captured by conversations in the hallway.

One problem in this type of project is that the different subsystems are suboptimized. Every one has agreed on a system architecture and then the work continues in the separate subsystems. It is the system engineer's task to make sure the subsystems are not suboptimized. Every week the persons responsible for the different subsystems have a meeting with the system engineer to discuss overall system issues. A new requirement has to be put in its context to see if it is meaningful for the system.

Analysis

As mentioned above, in the beginning of the project there isn't a lot of analysis. It is rather estimations of parameters based on experience from earlier project. When the system has evolved a bit it is more open to analysis. Then it might be possible to analyse what influence different parameters has on the system, e.g. how the system reacts to a high sampling frequency.

Everything isn't done just by experience. The control engineers have to specify some fundamental parameters on communication. This has to do with the inertia in the system and how fast the control has to be done. There should be a lot of reserves in the system. One important input is the dataflow that the control system needs and the maximum tolerable delay. The control engineers have their Matlab and Simulink models to help them communicate with the other team members.

The analysis includes schedulability analysis, memory budgets, execution time analysis and timing analysis. The requirements are checked to see what has to be analysed and what should be reported. ESA demands that a schedulability analysis is made. Then it has to be decided who is responsible for testing different things. It is important to try to understand the other subsystems requirements and to bring up own requirements. Other

analysis can be to look in detail at the functions of the processor, which can be captured in technical notes.

Architectural patterns from OO software community are examined to find possible candidates that can be used as components when building the system. The patterns might have to be adjusted to fit better into development of embedded systems. It is possible to get an idea of structures but the pattern is also adjusted to fit ObjectTime. When interesting patterns are found, a technical note is written on that pattern. According to Thomas, it is important to have in mind how one can calculate on how they work from the view of real-time aspects.

Models and mapping between models

At the early stages of the project, boxes and arrows are used to describe the system. This is informal models that may not have any meaning outside the room where the discussion takes place. This kind of models helps the designers to identify possible functionality and objects. From the requirements it is possible to get a hint of the communication between objects. It is already given what sensors and actuators that are placed on the nodes. This makes an object-oriented approach good for capturing high-level design. In some sense an object oriented method is used in this project, at least in the software part. It is not always so obvious what is meant with object oriented development. If it is looked at as design then it is more connected to implementation. If an object is a C++ object then it can't be used before the design level. At an analysis phase, where one tries to find objects that communicate, it is just a matter of how the boxes and arcs are drawn and that is what is needed in the beginning, i.e. this in fact corresponds to a functional approach.

For example UML is quite stuck in the lower design levels with class diagram. Thomas thinks ROOM is nicer in that way. At the architectural level boxes and arcs are the main building parts and this is what is done in ROOM. But one must still have in mind what the target environment is.

The control functionality is implemented as a function call in an ObjectTime model. As said above, the architecture is made with patterns and technical notes as a base. These are then used to build a ROOM model of the architecture. The architectural components are found with some feeling and are named by using patterns. Having an initial architecture the requirements can be expressed using message sequence charts (MSC). This gives a kind of requirements traceability. By adding more and more statemachines and actors with statemachines the architecture is going more and more towards design and towards more and more accurate code. This gives a good mapping because the same models are used throughout the process. If something is changed in for example the control system then new code is generated by pressing a button. Code generation is used very much in this project. When using ObjectTime, design and implementation is principle the same thing.

Traceability from a system part to a certain requirement is made by doing a print out of the database for that system part. During design some requirements may be discovered. These requirements are stored in the database. One problem with this is that everybody in the project can edit the database. There is only a relation downwards in the hierarchy, it is not possible to group subsystem requirements to see from what system requirement it comes.

System made for testing

Testing will be more than half of the work in this project. A simulator is built for system testing. The world around the system is modelled. A real system is built and is executed in the simulator. This will also be done during the mission to test commands before they are executed on the real satellite. Making test cases is an activity that is really time consuming. The subcontractors should be able to test their design in a similar environment before delivery.

The detailed description of design of the final system is made by drawings in electronic CAD. There is no feedback to the requirements from the detailed design and a detailed analysis. A lot of testing is done during development. The verification of requirements on delays is made by testing on a prototype. Three types of models of the hardware is used. The Test Engineering Model is an early prototype where some of the components are cheaper than the real ones. On this model informal testing is made, a sort of debugging the hardware. The Engineering Model (EM) is used to verify design. One of EM is a benchtest model where all the sensors and actuators are present. Testing is documented and design iteration may be appropriate. Software is tested on a simpler system with basically a processor board and by connecting simulated nodes to it. One model is made so the customer can make test flights, i.e. fly it on the ground.

2.2 Tools used in industry

The tools used in the SMART project:

- ObjectTime and ROOM is used to model the software. It is used for architecture design. It is a kind of boxes and arcs diagram, but in a more formal way. Every functional requirement is then expressed in a Message Sequence Chart.
- To make the schedulability analysis a tool called PERTS, www.tripac.com, is used. It has some connections to ObjectTime and VxWorks.
- MatLab/Simulink is used for control design
- Traditional electronic CAD systems are used for detailed hardware design.

2.3 The ideal tool

What is needed, according to Thomas, is a tool to create tests. There should be a library with components that models physical things. A database with components with parameters to facilitate analysis. Execution time analysis is one type of analysis that one wants to do. Tools to help partitioning is needed. Such a tool doesn't need to optimize the partitioning, it could instead tell what happens if the partitioning is done in a certain way. It is used to see if the partitioning is working.

A tool needed in this kind of project is a tool to manage requirements. The tool should help in capturing relations between requirements and traceability between design and requirements.

2.4 Discussion

The process used in this project may not be streamlined to success but with its strong demands from the customer to have a structured way of working it might have taken a step towards it. A set of tools are used in the development process:

- ObjectTime Developer
- PERTS
- Matlab/Simulink
- Electronic CAD
- Some Requirements tool

Some of these tools were used together, e.g. ObjectTime Developer and PERTS have some common interfaces. This shows that the integration of tools is given some interest in industry. The transformation of the control functionality is made by using Real-Time Workshop to generate code for a function that then is called from some object in the software. Even if the tools themselves don't have an interface directly, the mapping of the control system to the rest of the system is automated.

3 Modelling of distributed Real-Time Systems and OO approaches

3.1 Models, views and methods

The use of models makes it easier to build a system and to analyse it. Models help the architect in the communication with the client. The architect must be sure that the system does what the clients wants and if it's impossible to fulfil all requirements, a model can help describing it to the client. So, with the help of models an architect might get a clearer picture of the requirements by discussions with the client, it's easier to get the critical parts of a requirement right. The solution of a problem is dependent of what models are used. Therefore a good set of views should cover the whole system and be as orthogonal as they can be, i.e. contain different information about the system [9]. Some important views as stated in [8] are:

- **Purpose/objective**
- **Form**
- **Behavioural or functional**
- **Performance objectives or requirements**
- **Data**
- **Managerial**

The purpose and objective models describe what the client wants. Form is a view which describes physical elements of a system. A block diagram is an example of the form view. It can show how physical parts in the system are interconnected or a type of system flow diagram [8] that shows the flow of information in the system. Software systems form can be described by structure charts or class and object diagrams.

The performance view describes requirements of properties like latency and reliability. Behavioural or the functional models of a system give a picture of how the system behaves, i.e. how it does what it is supposed to do.

While the views are chosen to be quite independent there is some connection between them. For example the system behaviour is not independent of the systems form [8]. Different kinds of models and methods for modelling are used in different views. The abstraction level is important, too much details can make the model hard to understand but with less details some information might get lost. Another approach in dividing the modelling into different views is described in [3], the "4+1 view model", where the description of an architecture is made up of five views:

- **Logical**
- **Process**
- **Development**
- **Physical**
- **Scenarios**

The logical view is describing the object model of the design, that is showing what the system looks like in functionality. A process view should include the aspects of concurrency and synchronisation. The physical view is the view where non-functional requirements are modelled, i.e., reliability, performance and scalability. Scenarios is the "+1" view of this approach and is instances of use cases. Scenarios are used to look in detail at

some critical functionality of a system. In a logical view where object relationships are displayed, nothing is said explicitly about how the objects are partitioned to tasks. Therefore there should be some mapping of information from the logical view to the process view. This can be done in numerous ways and in [3] two strategies is mentioned, inside out which starts with the logical view and outside in which starts from the physical view. Not all the views are important to develop a particular system. In [3] it is stated that for example the physical view may be omitted if there is just a single processor in the system. That is a bit confusing as there might be non-functional requirements on a single processor system as well.

These two approaches are just descriptions of the views that a system architecture can be looked at. They don't say anything about how the views should be displayed, exactly what type of diagrams should be used and so on.

3.2 Object-orientation

An **object** is a unit that has a well-defined purpose and that encapsulates the resources to achieve that purpose[20]. The object is described in terms of its attributes, operations, dynamical aspects and the overall function of it. One of the key aspects with an object is that it hides the implementation from the outside and provides interfaces so it can interact with other objects. An object is defined by *attributes* and *methods*.

Objects with common features, i.e. same attributes and operations, are organized into **classes**. The description of a class is referred to as *object types* and the term *class* is used for the implementation of them.

Abstraction is a view where the details are put aside and the object is treated as a black box with an interface. An object can have an **encapsulation** shell that hides its data structure and implementation details from the rest of the system. Encapsulation makes the objects contents of a black box with a well defined interface. All the communication with the object is made through its interface. Other terms in the structuring of objects includes **aggregation**, i.e. a group of objects that form some composition of components. The components in a aggregate are not invisible to the outside, but they do form some kind of entity. The encapsulation of a group of objects is a much stronger form of organization than an aggregation.

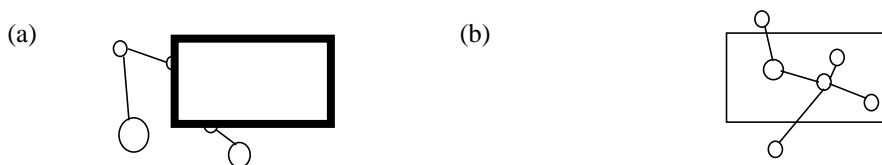


Figure 2. The users view of encapsulation (a) versus aggregation (b)

A class can inherit features from a superclass. **Inheritance** (or generalization) means that a subclass¹ has all the attributes and methods as its superclass. It is also possible that a subclass extend the superclass by adding attributes and methods (specialization). This type of relationship is often called “a kind of” relationships. Both a bicycle and a car is “a kind of” vehicle.

1. *Subclass* often referred to as child and *superclass* referred to as parent.

Objects communicate by **message** passing. This ensures the validity of data structures that are encapsulated in an object. Often messages are implemented as function calls[20]. This also makes the coupling between objects as weak as possible.

Object-orientation has become very popular in the development of systems. A software development process is normally made up of three important activities[20]. Specification, architectural modelling and implementation or analysis, design, coding and testing. One of the benefits with object-orientation is that it tries to use the same concepts for the different stages of the development. Other important benefits often mentioned when speaking about object-orientation are extensibility and reusability. Object-oriented systems could be built by already written components. The system will be easy to extend without the need for changes to the components that constitutes it. This reduces the effort put into the system development. The benefits mentioned above does not just apply to the reuse or extensions of code. Even design and analysis efforts can be stored in libraries that later can be used again. A lot of design patterns exist, that can be used to find already proven structures. Object-orientation has been used as a paradigm for programming and when it evolved and became more mature the interest for object-oriented design methods and object-oriented analysis and specification rised. For example UML, has become a popular modelling language in object-oriented development. It contains a set of diagrams for modelling different views of a system, an overview of UML is presented in section 4 of this report.

Stated advantages of using object-orientation[20]:

- It uses the same concepts in different stages
- It should be easier to extend a system
- It should be easier to make reusable components

Some additional stated advantages with object-orientation is given in [13]:

- Improved problem domain abstraction
- Improved stability when requirements changes
- Better support for reliability and safety concerns
- Inherent support for concurrency

As mentioned above, some of the key benefits are reusability and extensibility. When re-using classes that has been used in several projects the classes can be considered more bugfree. They have gone through an evolution towards higher quality. Well designed objects is the basis for building systems from reusable components.

Some pitfalls in an object-oriented approach is for example [20]:

- Deadlines on project can make the system be built by less reusable components.
- Managing libraries with components is difficult.
- Culture shifts in the companies, e.g. structured analysis versus object-orientation.

Object-oriented analysis

Analysis is usually done by decomposing a problem to smaller parts. Traditionally, analysis was made top-down with the use of structured analysis or other methods that use

functional decomposition. Object-oriented methods are traditionally not top-down. Rather they are bottom-up, i.e. candidate objects are found and then they may be composed to larger entities. Some of the usually mentioned object-oriented analysis methods[20] are:

- Shlaer/Mellor OOSA
- Coad/Yourdon
- OMT
- OOSE

One example of a object-oriented analysis method is the object modelling technique (OMT). It is shortly described below.

OMT

The Object Modelling Technique (OMT) is an object-oriented method that was very popular some years ago. It is influenced by traditional structured methods. OMT has a detailed notation and is quite complicated. James Rumbaugh is one of the main contributors to this method. OMT consists of three main phases [20].

- Analysis
- System design
- Object design

The *analysis* part consists of an object model (OM), dynamic model (DM) and functional model (FM). These models evolve from an initial requirements specification. It is an iterative process, where operations that come up in DM and FM are added to the OM. In the OM, both classes and instances of such are supported. There is one DM to every object and this is captured in a state transition diagram. The FM is just like a dataflow diagram in its traditional meaning and is often used at a high abstraction level.

In *system design*, subsystems are created and concurrency is identified using the DM. Allocation of subsystems to tasks and processes are also made in this phase.

Object design completes the OM with information extracted from DM and FM. Some other steps are left in this phase such as design algorithms, packaging and of course documentation of the more detailed OM, FM and DM.

Object-oriented design methods

Object-oriented analysis and object-oriented design is not so easily separated. Design can be divided into logical and physical design. Design adds more details and implementation aspects to the analysis models. Object-oriented design methods share a set of design steps. How these steps are dealt with in the different methods varies a lot. In [20] these steps are listed:

- Identify objects, methods and attributes.
- Decide the visibility of the objects with respect to each other.
- Make the interface descriptions of each object.
- Implement the objects and test them.

There exist a set of emerging object-orientated design methods. In these methods different ways to identify objects are used. Some of the methods use data flow diagrams to identify

the objects and their methods, e.g. early Booch methods. Other methods extract objects and methods from a textual description of the requirements. Of course, a method might use a combination of these techniques for identifying objects and methods. Below a short description of one object-oriented design method follows.

HOOD

The European Space Agency is the developer of HOOD[20], a method directed to Ada development. Objects are either passive or active. Passive objects can only use services provided by other passive objects while active objects can use any service. HOOD uses a top-down design methodology [20]. A number of steps are used in HOOD to decompose each object. It can use diagramming techniques from other methods like other structured analysis and design methods. A context diagram can give hardware objects and a dataflow diagram can give other types of objects. Requirements or problem statements are usually captured in text. The noun and verb method, where nouns represent objects and verbs represent methods, is supported. The next step is to form some solution strategy and this is made by making some initial HOOD diagrams and an outline of the strategy in text. This is the base for the solution. Further, a more formal solution strategy is needed. This step is taken by identifying objects and describing them in more detail in terms of operations and attributes. In this step, the grouping of objects is made as well as HOOD diagrams showing relationships and operations. The object definition skeleton (ODS) is then further refined, i.e. defining types, constants and data members. Dataflows can be shown and are based on structure charts.

3.3 Modelling tools

To be able to manage these different views and to check for inconsistencies in the models, tools are used. Some tools exist and of course they differ in functionality. Some tools are more or less drawing tools to support the user by checking the diagrams for inconsistencies, other tools might have code generators or some analysis functions. There are a lot of tools on the market that support UML or at least parts of it. Some tools do have some specific approaches towards real-time systems or at least have the intention to have it.

A list of some tools to help object-oriented development:

- Real-Time Studio by ARTiSAN, www.artisansw.com
- Rhapsody by i-Logix, www.ilogix.com
- Rational Rose by Rational, www.rational.com
- ObjectTime Developer by ObjectTime, www.objecttime.com

4 Overview of the Unified Modelling Language

The Unified Modelling Language (UML)[9] is a language, evolving from Booch, OMT, OOSE, and other methods, for modelling and specification of object-oriented systems in a visual way. It is possible to extend the UML for special purposes. This can be made by using stereotypes, tagged values and constraints. But even without extensions, UML is stated well suited for modelling of many types of system. If there really is a need for an extension one should have in mind that it might not be understood by everyone that is familiar with UML. The UML has nine different graphical diagrams for describing different views of the system. In [12] the diagram types are classified in a user view, a structural view, a behavioural view, an implementation view and an environment view:

- **User view**
Use case diagram
- **Structural view**
Class/object diagram
- **Behavioural view**
Statechart diagram
Activity diagram
Sequence diagram
Collaboration diagram
- **Implementation view**
Component diagram
- **Environment view**
Deployment diagram

UML has a strong emphasis on graphical notation. The following is a brief description of the different diagram types of UML. The main part is described in [12] and a more detailed description is found in the UML documents[10,11].

The architecture of UML consist of four layers. They are meta-metamodel, metamodel, model and user objects. The meta-metamodel defines the language itself. An instance of a meta-metamodel is a metamodel. Elements of the metamodel is used when modelling with UML. The meta model is divided into a set of packages. Examples of some standard modelling elements in the core package are (definitions from[11]):

- **Class** is a description of a set of objects that share the same attributes, operations, methods, relationships and semantics. A class may use a set of interfaces to specify collections of operations it provides to its environment.
- **Classifier** is an element that describes behavioural and structural features; it comes in several specific forms, including class, data type, interface and others that are defined in other metamodel packages.
- **Association** defines a semantic relationship between classifiers; the instances of an association are a set of tuples relating instances of the classifiers.
- **Method** is the implementation of an operation. It specifies the algorithm or procedure that effects the results of an operation.

4.1 Use Cases

Use case diagrams are used to show how actors interact with the system, which is a way of describing the purpose or the objective of the system. Use case diagrams consist of use

cases and actors. An actor can be either a human being or another system. A use case is a service provided by the system for the user. Use case diagrams can be used in different levels of abstraction, that is, a use case in one diagram can be expressed in more detail with more use cases in another diagram. The `<<uses>>` stereotype¹ indicates a common

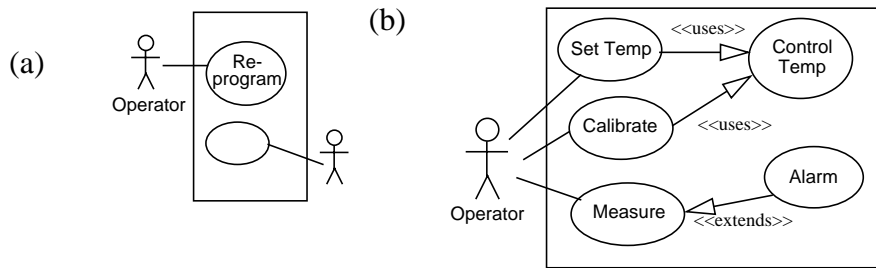


Figure 3. Use case diagram on a low-level (a) and a high-level (b)

functionality shared by two or more use cases. In figure 3b, both the Set Temp use case and the Calibrate use case are using the functionality specified by the Control Temp use case. The `<<extends>>` stereotype is used to indicate that the Alarm use case is optional from Measure use case. It can be used to capture exceptional functionalities. The documents of these diagrams is called a use case model.

4.2 Class/Object diagram

A class diagram is used to depict the static structure of a system. It consists of classes and associations. Classes represent things with common characteristics and features, such as attributes, operations and associations. A number of associations exist, for example aggregation, composition and generalization. Class modelling might be done at different levels. That is, operations, attributes and associations are defined in a more detailed class

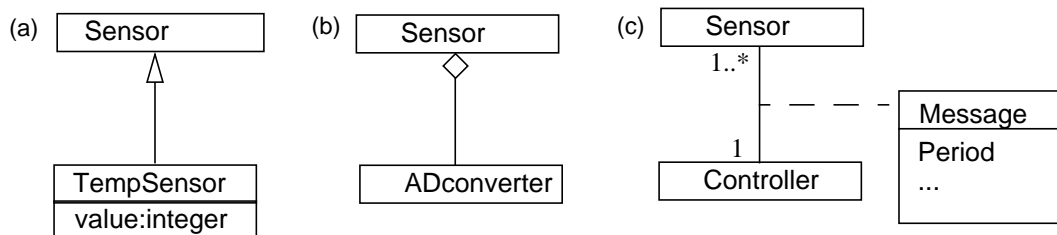


Figure 4. Class diagrams with examples of associations

diagram. In figure 4, three examples of associations are shown. The triangle in figure 4a is representing inheritance. TempSensor is a kind of Sensor. The diamond in figure 4b represents an aggregate association, a sensor has one A/D-converter. If the diamond was filled it would represent a stronger form of aggregation, i.e. composition. In that case the aggregate object is responsible for the creation and destruction of its parts. An unfilled diamond is an aggregate that is a weaker form of containment than composition. In figure 4c, Message is an association class.

1. A stereotype is one of the standard extension mechanisms in UML see 4.8 of this text.

Definition: An association class is an association that is also a class. It connects a set of classifiers and also defines a set of features that belongs to the relationship, not to any of the classifiers.

An association class is used to capture information about the relation between classes, not about the objects themselves. The Message class captures some information about the association between a sensor and a controller. Another good example, from the world of humans, is given in [5]. A marriage is an association between two persons. But where does the information about date of marriage and location of marriage fit. Not to any of the persons, rather to the association between them.

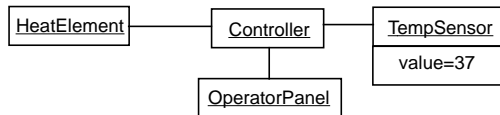


Figure 5. Object diagram

It is possible to have multiplicities in class diagrams to show how the number of instances are related in implementation. Figure 4c shows that a controller should be connected to at least one sensor.

Object diagrams, see figure 5, are used to capture a static view of a system with an instance of a class diagram. It captures a snapshot of a detailed state of the system at a certain point in time. With this diagram it is possible to check the class diagram so the rules of multiplicity are okay. In [10] it is stated that the use of object diagrams is fairly limited. Basically it doesn't show more of the structure than a class diagram.

4.3 Sequence diagrams.

Scenarios are instances of use cases or specifies use cases and capture message passing between objects. One of the diagrams that can be used for modelling scenarios is the Sequence diagram. It is said to be a part of the dynamic description of the system, i.e. in the behavioural view. Scenarios can be used to validate a sequence diagram. That is when the

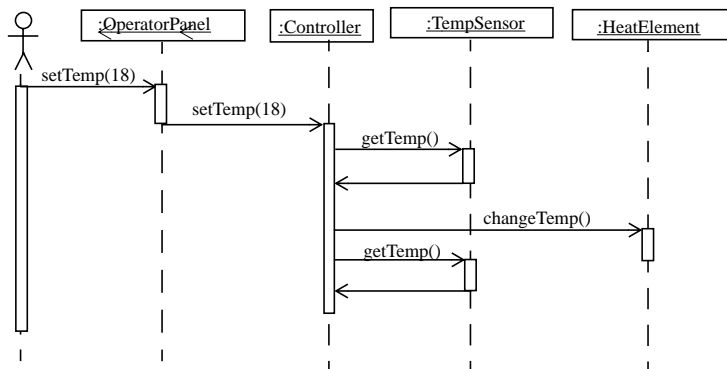


Figure 6. Sequence diagram for heater system, change temperature

sequence diagram is walked through with actual values on the messages. In figure 6 the scenario “set temp to 18 degrees”, is used and it is the instance form of the sequence diagram showing the sequence of messages for the “set temp” use case. A scenario shows the

instance form of a use case. The operator uses some form of panel to give the new temperature. It is then sent as a message to the controller object which then asks the sensor for the actual temperature. Further, the controller performs some computations and then sends a change message to the element.

Definition: A message¹ is a specification of a communication between instances. The receipt of a message instance is normally considered as an instance of an event.

Sequence diagrams contain models of class roles, lifelines and activations. Class roles represent the participating objects. In some sense it specifies what is required from a class to participate in a particular message exchange sequence. On the horizontal axis the message exchange is shown and the vertical axis represents the time. The time axis doesn't need to have any scaling, it only shows sequence. But it is possible to add timing notes or making the time axis in a time scale. The dashed lines in figure 6 are graphical notation for lifelines. They model the existence of an object over time. A lifeline ends when the last return message is sent or when the role object is destroyed, marked with a cross in the sequence diagram. An activation is depicted as a rectangle, showing the time that an object performs some action. Activations may be recursive, i.e. operations in an object may call itself.

Definition: A signal is a specification of an asynchronous stimulus communication between instances. The signal is received by a statemachine.

Messages may be simple, synchronous or asynchronous. A simple message is used when the details about communication are not known or not important at the current time. Synchronous messages passes control from the sender to the receiving object. A sender of asynchronous messages signals the receiver object and continues with its own activities without pausing. In this case the receiver has to be a an active object. A message might take time to perform its sending and this is shown by a slanted arrow.

4.4 Collaboration diagrams

Collaboration diagrams is one of the diagrams showing behavioural aspects of a system. It shows how interactions between objects takes place. It practically shows the same as a sequence diagram with the difference that a collaboration diagrams shows the associations between classes as well as the sequence of message passed among the objects. This kind of diagram might be better to use when the structure of classes and associations has evolved a bit. In the early phases of analysis a sequence diagram might be better to use, it can help finding classes or what is needed from classes to participate in a collaboration.

1. In some sources *event* and *message* is used synonymously, e.g. in [13]

A collaboration is defined by a set of classes, associations and exchanges of messages that

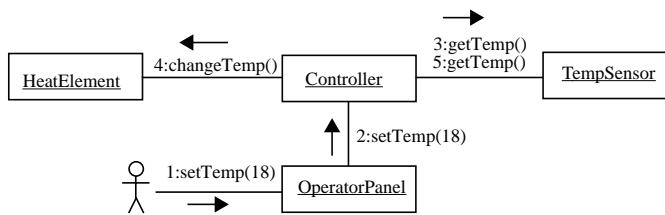


Figure 7. Collaboration diagram for change temperature

realizes a certain behaviour. The time is not shown in this type of diagram, instead just the order of the passed messages is represented. It may exist in both generic and instance form. A collaboration is a static view of a context that a set of classes participate in, this is shown in a collaboration diagram without messages, just showing associations. When a message sequence is added to a collaboration diagram, it is said to show an interaction. In figure 7, the collaboration diagram for the “Set Temp” use case, the message sequence is given with the numbers on the messages. It shows the messages passed in a particular scenario where the temperature is to be set to 18 degrees.

4.5 Statechart diagrams

A statechart is used to model complex dynamic behaviour. Statechart diagrams capture the behaviour of an object in the sense of how it reacts to events. For example, if an object has a set of operations, the use of a statechart can constrain the execution order of the operations. The object can be in a finite set of states, each having a defined set of events that it reacts to and actions taken under certain circumstances. One definition of a statechart given in [21] is:

“A finite state machine is an abstract machine that defines a finite set of conditions of existence (called “states”), a set of behaviors or actions performed in each of those states, and a set of events that cause changes in states according to a finite and well-defined rule set.”

Definition: A state is a condition in an objects life when it satisfies a certain condition, performs some activity or waits for a certain event to occur.

When an object is in a particular state it can only perform a subset of all its defined actions. It reacts only to a subset of all possible events and can only change state directly to a subset of the states it can be in. A transition between states generally takes place when a certain event occurs. A behaviour defined by a statechart is made up of states, transitions and actions. An action is a non-interruptible behaviour, that is executed at specific points in a statechart. It can be executed when a transition takes place and when entering or exiting a state. An action can either be a simple statement or the invocation of an operation.

A statechart can be assigned to classes and methods. It describes the behaviour that an object of a class has or it can be used to describe the algorithm of a method. A state may have internal transitions, i.e. actions that are performed as a response to events without making the object to change state. These internal transitions do not invoke the possible entry and exit actions of the state. Entry and exit actions are actions that always execute when entering or exiting a state. A self-transition is a transition that is taken to the same state as it

was in when the event happened. This type of transition is different from the internal transition, i.e. when a self-transition is taken the entry and exit actions are executed.

UML statecharts supports nested states, i.e. a superstate can contain a set of substates. This can continue down to an arbitrarily deep level. In a particular context, e.g. a high-level statechart, the object must be in one of the states. In figure 8a one of the states 1,2,3

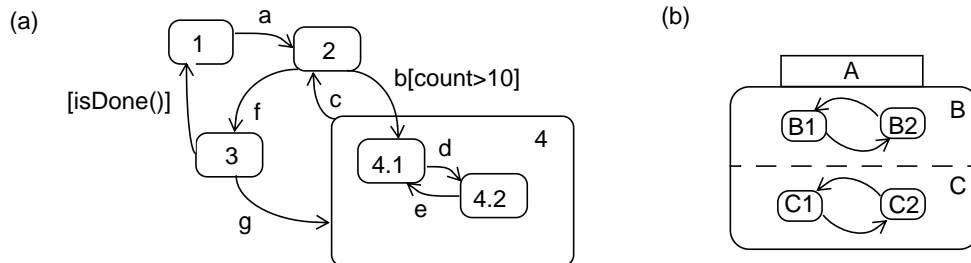


Figure 8. Example of nested statemachine (a) and a statemachine with orthogonal substates (b)

or 4 has to be active. In turn, when the superstate 4 is active the object has to be in either of the substates 4.1 or 4.2 at every moment in time.

A state can be expanded to see its internal structure or the details can be suppressed to clarify the diagram. It can be decomposed into concurrent substates or orthogonal substates. The state model has to be in one state from each of the orthogonal substates. In figure 8b the state A is divided in the orthogonal substates B and C. At any time B has to be in either B1 or B2 and at the same time C has to be in C1 or C2.

Definition: An event is an occurrence that may trigger a transition.

There exist different types of events that can trigger a transition [10]:

- **ChangeEvent**
A ChangeEvent is notated with the ‘when’ keyword and contains a boolean expression. The event occurs when the expression changes from false to true.
- **TimeEvent**
A TimeEvent can be notated with the keyword ‘after’. An elapsed time can be specified, e.g. after(15 ms) or after(20 ms since ‘event A occurred’).
- **SignalEvent**
A SignalEvent is the receipt of a signal from one object to another.
- **CallEvent**
A CallEvent is the receipt of an operation call from an object.

In figure 8a, when the statemachine is in state 2, it accepts the events ‘b’ and ‘f’. Both of them triggers a transition. The event ‘b’ makes the object transition to the state 4.1 and ‘f’ makes the state machine transition to state 3. A transition from a superstate applies to the nested substates of that state. The transition that takes place when event ‘c’ occurs applies to both 4.1 and 4.2. This notation simplifies the diagram since the number of arrows representing transitions are reduced.

If an event that does not match any of the events accepted in the particular state occurs it

is simply neglected. For example in figure 8a, if event ‘g’ occurs when the active state is 1 then the event ‘g’ is lost. A transition can have a guard expression to decide if an event fires the transition. If the event ‘b’ in figure 8a occurs the boolean expression [count>10] is evaluated. If it evaluates to true then the transition fires, otherwise the event is discarded. A transition can also be taken without a particular event occurring. That is the case with the transition between state 3 and state 1. The guard condition on that transition is evaluated every time state 3 is entered. If it evaluates to true it fires immediately, otherwise the evaluation is done the next time the state is entered. This type of transition is called a null transition.

4.6 Activity diagrams

The activity diagram is a special kind of statechart, where the states are Activities and the transitions fire when the ongoing activity is finished. Activities represent the execution of a procedure.

Definition: An activity diagram is a variation of a state machine where most of the states are activities that represent ongoing operation.

There can be ordinary states in an activity diagram, but most of them should represent Activities and most of the transitions should be triggered by the ending of an activity. The focus is on describing internal processing, not on transitions between states that are triggered by external events. An action normally represents a step in the execution of an algorithm. Decisions can be included in an activity model. A decision is expressed by guards with boolean expressions, and is notated by a diamond.

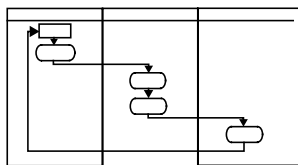


Figure 9. Schematic picture of an activity partitioned into swimlanes

Swimlanes is a construct for dividing the overall activity into groups. Each group may be implemented by one or more objects that is responsible for that part of the overall activity. The order of the swimlanes has no semantic meaning. The transitions between swimlanes are not allowed to represent transitions between threads, so this might make it hard to use for the partitioning of functions to different active objects. The semantics of transitions in an activity model, [11] p.125, are quite vague.

Definition: Object flows are associations between action states and objects. They are used to model that an action state uses an object.

Objects that are input or outputs to actions are shown as object symbols in an activity diagram. Object flow is denoted by dashed lines between an action state and an object symbol.

4.7 Component diagrams/Deployment diagrams

Component and deployment diagrams show different aspects of implementation. The

component diagram shows the structure of the source code and the deployment diagram shows the run-time system structure. Components reside on nodes that are shown in deployment diagrams. Nodes represent computational resources and may contain aggregates of other components, nodes, objects and processes.

4.8 Extension mechanism

Even if the UML includes a rich set of modelling concepts and notations, users may want to use additional notations or features. There exist three built in extension mechanisms in UML, which makes it possible to add new modelling elements. The extension mechanisms are stereotypes, tagged values, properties and constraints.

Stereotypes

UML provides certain predefined stereotypes some of them more interesting for a real-time purpose than others, e.g. <<*process*>> and <<*thread*>> stereotypes. These stereotypes applies on classifiers, e.g. on a class.

Definition: A stereotype represents a subclass of an existing modelling element with the same form but with a different intent.

The semantics of the <<*process*>> stereotype is that it represents an active class with a heavyweight flow of control. The <<*thread*>> stereotype represents a lightweight thread of control, i.e. one sequence of execution.

Tagged values

Tagged values are used for property specification. It is a keyword-value pair. The keywords are called tags. A tag represents properties that the modelling element it is attached to has. The value is a string that represents the value of the property that the keyword represents. There exist some predefined tagged values in UML. Location is a tagged value that when applied to a component specifies on what node the component is located.

Constraints

Constraints are used to add new semantics to a model element. This is done by writing the constraint in some appropriate language. For example in a language designed for writing constraints (e.g. OCL), natural language or mathematical notation.

4.9 Discussion

The UML is a rich modelling language, at least if the number of diagram types is a measure of richness. This gives the user a lot of capabilities to model a system. But this might make it too general and with too many diagram types. Another question is the one about the semantics. It is quite hard to grasp the semantics at a glance, so this might give room for different interpretations of the model dependent on who interprets it. An extension may not be understood by everyone that uses the UML. Even if it is a general modelling language it is quite stuck with software related issues. This might of course be dependent on personal ability to make abstractions of things. The possibilities to extend and tailor UML for personal use is therefore both an advantage and a disadvantage.

A statechart is powerful in making models of behaviour, but that is for event-triggered system mainly.

The overview given here is just showing a subset of all the different notation possible. It shows all the diagram types, but for example composition can be shown by including the component classes inside the symbol of a composite.

5 Overview of Real-time Object-Oriented Modelling (ROOM)[5]

5.1 Background of ROOM

There exist a set of properties that characterize the types of systems that ROOM might be applied to. That is, timeliness, dynamic internal structure, reactivity, concurrency and distribution. Those listed issues are quite complex even when they are treated alone and the complexity of modelling and design increases when they are taken together. In [5], it is stated that the ROOM modelling language was developed for the purpose to make it easier to model real-time systems. It is also mentioned that ROOM may not fit so well to some systems that often are referred to as real-time and it might be applicable to other systems not typically called real-time. That is because the definition of what constitutes a real-time system varies.

One useful technique to deal with complexity in modelling and design of a system is to use abstraction. Abstraction is a driving mechanism for modelling in ROOM. In ROOM the highest levels of abstraction are referred to as architectural levels. To be able to handle different levels of abstraction some strategies have to be used. In ROOM the most important ones are, recursion, incremental modelling and reuse. The term Recursion used in ROOM is the same as functional decomposition. It is to go inside a modelling element and make a more detailed model of a system. A typically top-level model is the system, which then can recursively be decomposed into more detailed parts that in turn can be further decomposed and so on. Incremental modelling is the possibility to not care about details in the beginning. Instead of modelling a system down to a detailed level it's better to make a simple model that can be returned to later and make refinements.

The creation of the ROOM modelling language was made around three important points in modelling strategies[5], namely the *operational approach*, a *phase independent modelling abstraction approach* and the *object paradigm*. In a development process there are always discontinuities of different aspects. The modelling of requirements, design and implementation emphasizes on different set of details. The modelling methods and languages chosen for a development process can introduce discontinuities in the process. This can be removed if the chosen modelling approach bridges those gaps. In [5] a set of discontinuities is discovered and divided into different types:

- **Scope discontinuities** are introduced if different notations are used for different levels of the representation.
- **Semantic discontinuities** occurs if there are different notational sets for modelling of different views of a system.
- **Development phase** discontinuities are due to the lack of formal coupling between requirements, design and implementation models. For example that is the problem if there are different notation in different phases.

In the *operational approach* early models are treated as programs that are coded in a high-level language that can be executed. In an executable modelling language all the elements are formal and executable parts of the whole model, e.g. one feature in ROOM is executable state machines. This puts away the scope and semantic discontinuities in the process. One stated side-effect of making executable models early is that the required effort increases in early phases of development. That is because more details than traditionally has to be added in early phases to make a model executable.

To deal with *phase discontinuities*, the ROOM modelling language uses a single set of concepts in requirements definition, design modelling as well as in implementation, but it can be discussed if this is achieved. It is stated that this doesn't have to imply that the distinction between the phases disappears. There must be consistence between the design and implementation.

In the *object paradigm* used in ROOM, objects are encapsulated in a shell that defines an interface for interacting with other objects. As in other OO languages ROOM represents models by class definitions that can be incarnated to represent an object. Inheritance is supported by ROOM, but not multiple inheritance. Multiple inheritance is when a sub-class inherits properties from two or more superclasses. One drawback with multiple inheritance is when two superclasses have, for example, attributes with the same name. An alternative to multiple inheritance is delegation and there is a support for that in ROOM. Delegation is when an object passes messages to an component object that has the capability to deal with it, the object delegates the responsibility to perform an operation. From the ROOM point of view a system is a set of interacting objects that use message passing for communication. Objects in ROOM are defined as independent logical machines that do not need to be implemented in software. Those concurrent active logical machines are called *actors* in ROOM.

5.2 Actors

Actors are used to model high-level structures of a system. Actors represents active objects with a defined purpose. They are active in the sense that an actor may have its own thread of execution and can operate concurrently with other active objects. The scope of an actor may be as large as desired, i.e. any system can be looked at as a part of another larger system. Actors are created from actor class definitions that are the basic components of a ROOM model. In contrast to an actor, passive data objects exist. The functionality of a passive data object has to be activated through an actors thread of execution. A passive data object exist only in the context of one single execution thread, i.e. data objects can only be accessed by the actor that encapsulate it. There is no sharing of data be-

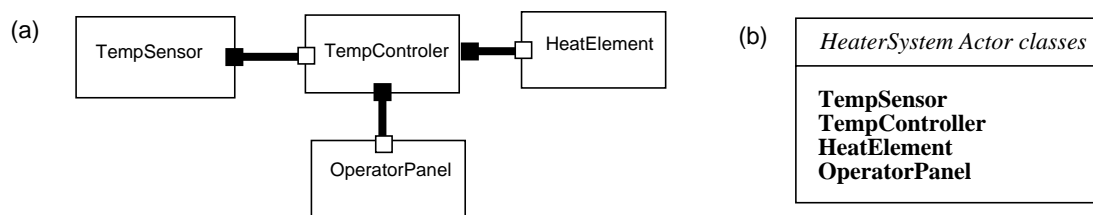


Figure 10. Heater system structure(a) and a list of actor classes (b)

tween concurrent threads in a ROOM model. Instead the data objects are copied and sent by messages through the interface of the actor. A passive data object is therefore not part of the high-level structure of a system.

Actors may be used to model physical objects, e.g. sensors or actuators, but can of course represent software entities. As in figure 10a where a temperature sensor is connected to a temperature controller. These blocks may give some candidate actor classes which can be captured in a list, see figure 10b. The actors in figure 10a are said to be references of actors and this view is used to capture the structure of the parts constituting the system.

Definition: A reference is neither an object or a class. It is a reference to another class definition. Changes are not allowed at the reference level.

The ROOM language supports hierarchical structure modelling. Several actors can be composed into another actor or an actor can be decomposed into several actors and this can continue to an arbitrarily low level. The HeaterSystem actor classes defined in the list above might be composed to form an actor of larger scope, i.e. HeaterSystem. This is shown in figure 11. The connections between the actors are called bindings and they represent some underlying communication channel.

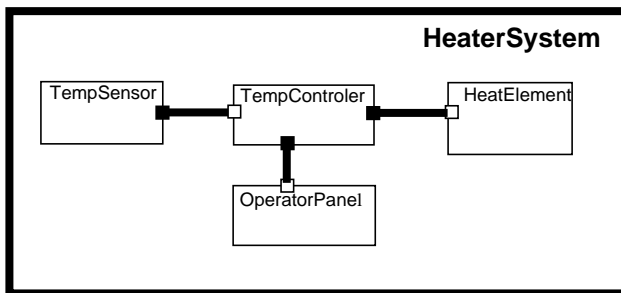


Figure 11. Heatersystem top-level actor with its components structure given

5.3 Interface components

Communication with other actors is done by sending messages through one of the interface components of the actor. An actor can have more than one interface component which makes it possible to have different views of the same actor. It is stated in [5], that this is really useful for real-time systems, since most of the concurrent objects collaborate with more than one actor simultaneously.

Three types of interface components exist in ROOM:

- Ports
- Service access points (SAPs)
- Service provision points (SPPs)

Ports are used for communication in the same layer while SAPs and SPPs are used to communicate between layers in a layered architecture.

Protocols

Messages are grouped and structured by protocol class definitions. For example, the destination of the messages might be a basis to conform them into a protocol. A protocol is defined of a direction (in or out) for every message set, a signal that identifies the message and possible data objects that are sent or received with the message. There also exist two optional specifications in the definition, i.e. a specification of valid message exchange sequences and a specification of the expected quality of service. A message-exchange specification can be of different types, e.g. a message sequence diagram or a state machine. In [5] the message exchange sequence for a protocol is mentioned but not further examined.

Definition: A protocol is conformed of a set of messages exchanged between two parties.

The TempController actor in the example system participates in message passing with other actors in the system. Messages are identified and can be grouped in sending and receiving ones. The messages in the case of the TempController can for example be:

- Receive:** tempParam(wantedTemp)
- Receive:** operatorCommand(on,off)
- Receive:** sensorStatus
- Receive:** elementStatus

- Send:** setTemp(temp)
- Send:** calibrate
- Send:** sendTemp
- Send:** shutOff, turnOn
- Send:** systemStatus

These messages can then be grouped to form a set of protocol classes. One protocol class could be an OperatorDialog, where the messages of interest are tempParam, operatorCommand and systemStatus. The protocol classes are then added to the list of HeaterSystem classes. The list is separated into the different types of classes and each column has an icon to indicate the type of classes in that column. Because the protocol has incoming

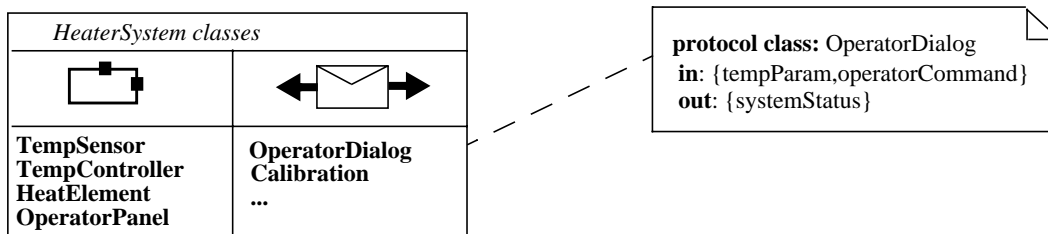


Figure 12. Heater system classes and a more detailed definition of the operator dialog protocol

and outgoing messages it is said to have a polarity. The protocol is defined from the view of one of the participants in the communication. Instead of defining the protocol again with the message sets switched a conjugated protocol can be used. The graphical notation for a conjugated port is the same symbol but with inverted colours.

Ports

The interface of an actor is defined by ports. A port is a declaration that a set of messages defined by the protocol is part of the actors interface. A more detailed view of an actor is an Expanded Structure Definition. This type of diagram shows the ports and what protocols that defines these ports, see figure 13. These ports do not represent either a protocol class or an instance of one. Instead a port represents a reference to a protocol class. It is possible for two ports to reference the same protocol. For example the elementCalibration and sensorCalibration plays two different roles for the TempController actor. Both reference the Calibration protocol class but use it in different ways. If the message sequence

when calibrating the sensor is changed then the change has to be in the Calibration class, not in the definition of the TempController class. This of course changes the properties of the calibration of the element as well. An alternative could be to have two different protocols, one for the sensor calibration and a separate one for the element calibration.

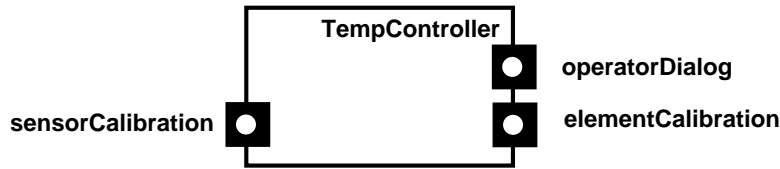


Figure 13. Expanded Structure Definition of TempController

Communication is, as said before, made by message passing through ports and ports can be of different kinds:

- Internal End ports
- External End ports
- Relay ports

Ports are used for actor communication in the same layer. End ports are used to communicate with the actor’s behaviour directly. An actor has behaviour that is defined by a behaviour component. A behaviour component is defined by the use of a ROOMChart which is described more in section 5.4 of this text. A relay port makes it possible for a contained actor to let the containing actor use an interface of the component. In figure 14

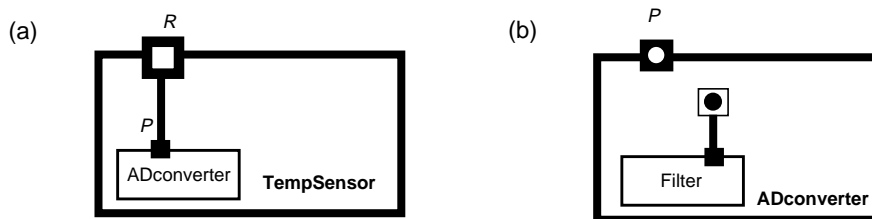


Figure 14. Example of different port types

port *R* is a relay port that passes the message through to port *P*. Port *P* may in turn be a relay port and delegate the message further down the hierarchy. The graphical notation for a reference of an actor suppresses the details of the possible contained actors in ADconverter and does not make any distinction between what types of ports the squares represents. Figure 14a shows the expanded structure definition of TempSensor. From figure 14b, the expanded structure definition of the ADconverter actor, it is possible to see that port *p* is an external end port, i.e. a part of the ADconverter interface that is directly connected to the behaviour component of the ADconverter. An internal end port is a port that connects a component actor to its composite actor behaviour directly. In figure 14b, the filter actor is connected to an internal end port of ADconverter and therefore has a connection to the behaviour component of ADconverter. The graphical notation for an internal end port is the same as for an external, but it is pictured drawn inside the border and not on it

Messages

The information sent between actors is first transferred to a message object, which is later sent between the actors with the help of some communication service. A ROOM model allows both synchronous¹ and asynchronous² communication. A message is a type of data object with a message signal attribute, a message priority attribute and an optional message data object. Actors is the basic unit for describing structure in a ROOM model. But there exist other more traditional objects in ROOM. That is objects representing abstract data types. In ROOM these are called data objects and are instances of data classes. Data objects that are encapsulated in actors are called variables. Data classes are used to define both variables in actors as well as for the data objects that are part of messages. In figure

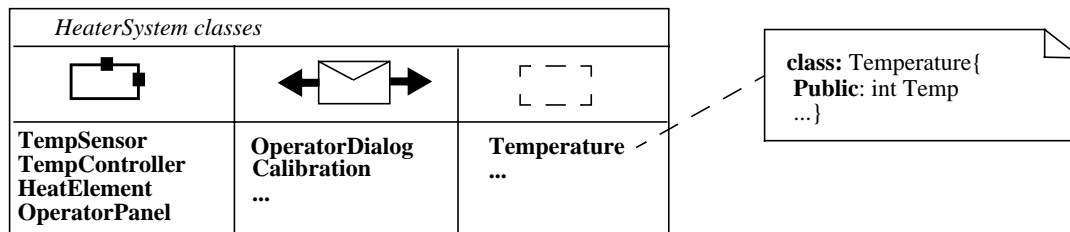


Figure 15. A more refined list of HeaterSystem classes

15, a data class, Temperature, is added to the ROOM model of the HeaterSystem. The textual note added to the list is a Expanded definition of the data class Temperature.

Layers

Layering is an abstraction mechanism to help reduce the complexity of a system. In ROOM a layer is just a kind of actor with a special kind of interface. To support layering the concepts of Service Access Points and Service Provision Points are introduced. Layering is a form of hierarchical relation. A higher level layer implementation depends on the lower layers. This is part of the structural constructs in the ROOM language.

5.4 Behaviour component

Behaviour is captured by the use of a ROOMchart. The behaviour component is part of an actors specification. A ROOMchart is basically a traditional finite state machine and a variant of an extended state machine. Extended state machines have, beside of the states, some extended state variables that are defined in the context of that particular behaviour component. They are used to have some variables that don't affect the states. A ROOM-chart supports nested hierarchical states. States that cannot be further decomposed are called leaf states.

In ROOM an event is generated when a message is sent through the interface component of an actor. The behaviour component of an actor can be in two different modes. It is either waiting for new events or is busy handling some event that has already occurred.

A transition in a ROOMchart is triggered by the arrival of messages through an end port

1. *Asynchronous* communication, the execution continues in the sending actor.
2. *Synchronous* communication, blocked until returned reply.

connected to the behaviour component. Transitions are defined by $t:\{p,s,g\}$, where p is the name of the port, s is the name of the signal and g is an optional guard function. If the guard evaluates to false then the message is discarded and the transition isn't taken. Actions performed when a transition is taken, are defined by adding these statements to the ROOMchart. A statement can be a simple action or a call to a function performing a sequence of actions. For example it can be sending a message to another actor or changing values of an extended state variable. An action can be one or more statements and may be attached to:

- Transitions.
- States, in form of entry actions or exit actions.

The initial transition is taken when a new behaviour is created, i.e. on the creation of an actor. The initial state is represented by a circle with an "I" inside. The initial state is just a pseudo state, i.e. the actor cannot stay in that state. The case when the state machine represents the top level behaviour of an actor, the initial transition is said to represent the creation of an concurrent thread of execution. This is because an actor is an active object, i.e. a concurrent object with its own thread of execution.

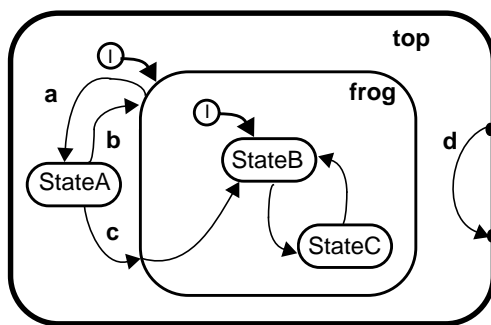


Figure 16. Example of ROOMchart

It is possible to perform group transitions, that is when one event triggers a transition to a higher level state no matter what state the behaviour is in. In figure 16 the transition 'a' is taken when the associated event occurs no matter what substate of the state 'frog' the actor is in. The transition 'b' returns to a history state, that is the state it was in the last time state frog was active. The transition 'd' is taken at whatever the current state is, it is a top-level grouptransition. This might be good for capturing exceptional behaviour. This transition ends on the border of the top-level state. When possible actions have been completed, the behaviour returns to a history state, i.e. the state it was in before the event occurred. Group transitions can be overridden by having an explicit transition from one state to another, with the same definition as the group transition.

Message Sequence Charts can be used to show sequences of messages sent between actors. In modelling with ROOM the use of MSC's is not a big issue. It is stated in [5] that some improvements on that topic is needed in future versions of ROOM. It is anyway a good way to capture interactions between actors.

5.5 Events in a ROOM model

The model used for processing of events in ROOM is the so called "run-to-completion" (RTC) model. One advantage of this model is simplicity[5]. The biggest stated disadvantage[5]. The biggest stated disadvantage is that the processing of an event cannot take too long. This RTC model does not

mean that the processing of an event can take all the computing power of the processor, it applies to events sent to an already executing actor. High priority events sent to an actor that is idle can pre-empt another actor handling an event with lower priority.

In ROOM priorities are assigned to events instead of threads or tasks. This is because in an event-driven system an actor can participate in different scenarios with other actors, i.e. an actor can have the responsibility for activities with different priorities. The semantics of event priorities is that events with higher priorities get *some* precedence over events with lower priorities.

In [5] page 221 it is stated that “*it is a good idea to avoid models that are critically dependent on priorities*”. That is because of the weak semantics of event priorities. The scheduling of event processing is done by the scheduling system within the ROOM virtual machine¹. A ROOM model can be distributed so the scheduling of events can be different in different processing sites. No assumptions are made about that the different scheduling systems are synchronized.

5.6 Implementation of ROOM models

The implementation of a ROOM model can be done in two different ways:

- Have an implementation of the ROOM virtual machine and then execute the model directly on it. In this case makes the model is the implementation.
- Do a mapping of the model to the target environment.

ROOM virtual machine

The ROOM virtual machine is a device that can execute ROOM model specifications. It is possible to implement this in hardware, but is preferably implemented as software to make it portable to different target environments. A ROOM virtual machine provides the system with the services that are needed. The virtual machine is not further examined since the focus on this work is on modelling, it is enough to know that the ROOM virtual machine exist.

Mapping it to target

The target environment is usually made up of a programming language and a run-time environment (e.g. a Real-Time Operating System). In [5], a set of assumptions is made about the environment. A primary assumption about the environment is that it support some form of concurrency paradigm, either by the programming language or by the run-time environment.

- The operating system kernel supports some form of thread that has its own context where it can store state information when it is idle.
- Thread synchronization is not needed since in a ROOM model shared memory is not allowed. But if the underlying language supports shared memory then this can be used to increase performance. Then usual mechanisms can be used (e.g. semaphores).
- Communication between threads is based on message passing. Asynchronous is the simplest type of communication. Some form of queuing is needed when multiple messages can arrive concurrently. This can be handled by for example a mailbox.
- Basically the scheduling in ROOM is based on event priorities, that have quite weak

1. A Model executing environment.

semantics. Therefore the scheduling services of an underlying kernel is usually used. A stated problem with this is that most scheduling schemes is based on thread priorities and not event priorities. This may lead to some rethinking if the design is based on event priorities.

- As a basic assumption each actor has its own address space. This comes from the semantic that actors don't have shared memory. But if the implementation environment allows shared memory then some memory protection may be needed.

Decisions about the implementation of the structural parts of a ROOM model has to be taken before the design starts. If the choice is to map the model specifications directly to the target without a virtual machine, then two alternatives exist. Either the design is restricted to use concepts that are supported in the target environment. As an example, if the operating system doesn't support task hierarchies then the concept of actor containment may not be used in the models. The other alternative is to realize the structural relations as good as possible. For example let an aggregate actor create its component actors.

5.7 Discussion

In ROOM it is possible to capture a hierarchical structure relationship in a nice way. This in fact is the same as functional decomposition. Every actor in a ROOM model is an active object with its own thread of execution. This can be a problem if the target environment doesn't support hierarchical thread structures. One solution to this is to create a separate schedulable environment inside an actor [5].

The behavioural description of a ROOM model is made with ROOMcharts, a form of state machine. This is a powerful way to describe behaviour, at least for event-driven systems. The possibilities to express explicit timing requirements solely in a ROOM chart is hard to do. A timeout event isn't so well suited to express timing requirements, i.e. in many real-time control systems the timeout should never occur.

The fact that a message sequence diagram can be used for capturing behaviour is obvious. In ROOM that topic isn't so deeply examined. To capture timing requirements notes could be added to the points in the diagram when the messages are sent. The MSCs in ROOM looks to be used for validating message sequences when executing the model.

Something that seems to be a problem is that there aren't any possibilities to model systems that are based on thread priorities. It is stated in [5] that it is possible to emulate thread priorities by giving all the messages sent to an actor the same priority.

6 Real-Time approaches and UML

There is at this moment no standard real-time extension to UML, i.e. adopted by OMG, but some approaches are coming up. The focus in some of those approaches are time, i.e. how to express timing issues in UML. Other important issues are these about synchronisation and messages. The term Real-Time UML should maybe be used with some caution.

The approaches looked at are:

- Real-Time UML
- The ACCORD project
- UML profile for scheduling, performance and time.
- ARTiSAN Real-Time Studio

One source of an approach to something addressed as Real-Time UML is [13]. This book tries to show a way of developing embedded real-time systems with the use of OO and specifically UML to represent it.

6.1 Real-Time UML

The development is divided into analysis and design steps. Analysis is made up of requirements and identification of objects and classes. The design step is divided into architectural, mechanistic and detailed design levels. The author of [13] uses UML and makes extensions to UML in way to make it applicable on the design of embedded real-time systems and the complexity therein.

At a high-level or in an early analysis phase, a context diagram is good to show to what external entities a system is connected to. This is representing the system architecture. In [13] and [14] it is stated that a use case diagram doesn't cover the aspects that a context diagram should. Instead it is the object collaboration diagram that is used to express the context of a system. Both approaches uses an object collaboration diagram without numbering of the messages. This makes it possible to capture messages sent between the system and the environment. The approach in [14] doesn't focus on what the characteristics of messages are. In [13] this is however considered important and stereotypes are introduced for the different types.

The fundamental communication mechanism supported by UML is messages. In UML a named event is called a signal. For this a standard `<<signal>>` stereotype exists. Events are classes and can be organized in a class hierarchy. According to [13], the nature of the message arrival is important to describe timing behaviour. Since UML doesn't define this, in [13] two arrival patterns are defined.

Definition: A periodic event has a period and optionally a jitter attribute can be specified.

Definition: An episodic event has a minimum interarrival time and an optional average rate.

Synchronisation is also an important property on a message that is interesting when designing a real-time system. In UML the synchronisation patterns that are defined are waiting, synchronous and call. Extensions have been made by Booch, according to [13], i.e. balking and timeout patterns. Balking is when the message is discarded if nobody is there

to take care of it. A timeout pattern do the same but waits for some time before proceeding.

Since old information generally is bad information in real-time systems, timing specification is important. In [13], this is handled by saying that the messages must have some parameters to specify timing. As mentioned above that is for periodic events a period and jitter. In [13] the events that affect the system are gathered in a table to show system response, response time and so on. Further it is said that “*a more complex timing needs a more complex modelling.*” This may be true, but this is quite domain specific concerns.

To define the timing behaviour, UML sequence diagrams can be used with additional timing constraints. It is possible to mark events by identifiers that then can be used in constraint expressions, i.e. timing marks. Timing marks and event identifiers are standard UML. In figure 17 the event identifiers can be used to specify the period of a sequence. By using constraints in timing marks this can be specified as $\underline{ab}=X\text{ ms}$, $\{\text{PERIOD}(ab)=X\text{ ms}\}$ or $\{a'-a=X\text{ ms}\}$, they all mean the same thing. Further, one constraint can be used to define a deadline of a sequence, i.e. $\{b-a=Y\text{ ms}\}$. Broadcast messages are sent to multiple object instances at the same time by the same object. In a sequence diagram this is shown by drawing multiple lines from the same object to more than one objects lifeline, see figure 17. State marks is another extension introduced in [13], i.e. ordinary state symbols from an UML statechart diagram that are included on the lifeline of an object. By this it is possible to follow the change in state by the participation in a certain interaction.

Definition: In UML an interaction is a specification of how messages are sent between instances to perform a specific task.

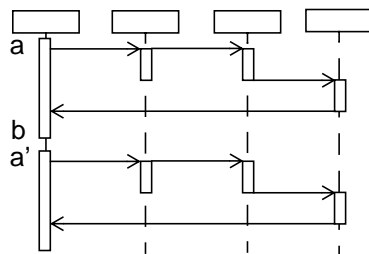


Figure 17. Example of sequence diagram with timing marks

Concurrent collaboration diagrams can be used to show how messages are sent between different threads or tasks. It is a collaboration diagram with active objects passing messages between their components. The messages are numbered both with a thread ID and a sequence number in that thread. A concurrent collaboration diagram is a collaboration diagram with composite active objects and thread numbered messages. Thread numbered messages in a collaboration diagram are standard UML.

To design the architecture for the system a deployment diagram with nodes, packages and active objects can be used. To represent tasks and showing task structures it is possible to use a class or object diagram and populate with active objects only. With the use of complex transitions in statecharts it's possible to model concurrent threads with roots in the active object[13].

6.2 The ACCORD approach

The ACCORD approach [18] uses UML to define a framework for designing real-time systems. The process they use is divided into analysis, design, implementation and deployment. In a real-time design step, extensions are added to UML. In ACCORD an active object is called a RealTimeObject given the stereotype <<RealTimeObject>>. In an active object operations are grouped into Readers and Writers operations. Readers operations can execute in parallel and a Writers operation must execute in sequence with other Readers or Writers operation. Any other operation can execute concurrently with other operations in the real-time object.

Some data objects may be shared among several active objects and thus have to be protected. When such objects are found they can be given the stereotype <<ProtectedPassiveObject>>, i.e. it specifies that concurrency control mechanism exist for data protection. Timing constraints are expressed in sequence diagrams. The timing constraints are attached to requests and specifies deadline of the execution of the operation. Periodic operations are expressed as cyclic transitions on Statechart diagrams.

Two types of StateMachines are defined in the ACCORD approach. One for classes and one for operation. The elements state and transition are redefined. A state with the <<A-State>> stereotype has among other things three associated events:

- **Deferred events** that are the same as in UML, i.e. it is put in a queue for handling later on.
- **Rejected events** are a list of events that are considered as faults and throws an exception when it is in the particular state.
- **Ignored events** are unexpected in the current state and is therefore lost. Ignored event is the default.

6.3 UML profile for Scheduling, Performance and Time

In a request for proposal [17] OMGs Real-Time Analysis and Design initiative some really interesting issues are posted. Below some of the points are listed and some of the requirements on a point is given. The list in this document is a subset of the list in [17].

Timing specifications:

- deadlines, periods, frequencies, jitter and its stochastic properties, intervals, duration and latency
- response times, response delay times and execution times for behaviours

Timing facilities and services:

- time resolution, jitter and stochastic properties
- explicit timer objects, clocks, clock synchronisation policies
- OS timing services

Resource-Related Semantics and Notation:

- modelling physical resources such as processors, networks and memory
- modelling non-physical resources such as buffers and semaphores
- representation of deployment of software components to physical resources
- specifying resource characteristics such as execution time, memory and bandwidth

Resource management policies:

- define represent and specify priority inversion bounding etc.

Concurrency:

- possibilities to show different models of concurrency, schedulable entities (e.g. process, threads)

Schedulers:

- represent and specify schedulable entities
- deployment to physical resources
- schedulers as controllable entities

Processors scheduling policies:

- processor scheduling policies such as rate monotonic scheduling, EDF and should support priority, priority inversion bounding
- distributed and local processor scheduling policies

Visually relating time and behaviour:

- should include visual representations for the mentioned policies

This is an interesting initiative taken by the RTAD of the OMG and it remains to see what it will look like when it is finished. According to the timetable there should be a voting for adopting the specifications in July 2000.

6.4 ARTiSAN Real-Time studio

Real-Time studio is a tool for making UML models and has made an approach towards a modelling environment with a focus on Real-Time systems design. This is a UML based modelling method with extensions to cover real-time issues.

What UML models are used

Some of the models and diagrams in Real-Time studio are standard UML diagrams and are basically used as they are supposed to be. Below there is a list with the supported UML diagrams.

- *Use case diagrams*
- *State Transition Diagram*, i.e. StateChart
This can be used both to express the behaviour of a class or it can be used to document the possible operational states of a system, in that case it is a *System Mode* diagram.
- *Class diagram*
- *Sequence diagram*
- *Collaboration diagrams*

What models are not standard UML

Some of the models used in the Real-Time studio are not in the set of models in standard UML. Although, some familiar UML notation is used in these diagrams(e.g. actors is used in some diagrams).

- *Context diagram*

To define the system scope a context diagram is used, this is made in a System Architecture Diagram. In a context diagram the system boundaries and system software boundaries are defined. Actors, representing people or other systems, are displayed outside the system boundaries. The actors in these diagrams must exist on either the use case diagram or a system architecture diagram.

- *State Event Matrix*
A State Event Matrix is used to document all possible state/event combinations. This matrix is of course very dependent on the State Transition Diagrams and so if one changes both have to keep consistency.
- *Constraints diagram*
A constraints model can be captured either in a table or in a diagram. This makes it possible to define constraints that can be applied on the system. It captures non-functional requirements. Constraints may be of different types, e.g. performance, throughput etc. Each requirement has a name, a description of how the measure will be made and a target value.
- *Concurrency diagram*
A concurrency diagram is used to document a multitasking architecture for a given processor. The scope of such a diagram is the tasks and intertask communication primitives for a single processor. In this diagram a node represents a task. A task can also be a thread or a process. Properties, e.g. priorities, can be added to tasks.
- *System Architecture Diagram*
The hardware architecture is modelled in this diagram. It captures all the hardware and communication links used in the system. It is a block diagram with nodes that represents, e.g. boards or subsystems. Connections are used to capture communication elements.
- *System Modes diagram*
The system mode diagram is expressed with a traditional statechart and is used to capture the different operational states a system can be in.

The tool includes a mentor function to help the user through the development. The method for developing a real-time system is called the Real-Time perspective [19] and is summarized below.

Real-Time Perspective

Real-Time Perspective (RTP) is a set of object-oriented development processes and techniques which is meant to help in all aspects of developing real-time systems. It's important to have a clear separation of concerns when a solution to a problem is to be found. RTP separates at a high-level the problem and the solution, in fact the *requirements* from the *system*. Further, in the solution there is a separation of concern, between objects that describe the system functionality, the software components that define concurrency and persistence and system components that define the hardware.

In the **Requirements Architecture** four models are used:

- System Scope
- System Constraints
- System Usage
- System Modes

The **Solution Architecture** is separated into three different layers:

- Object architecture
- Software architecture
- System architecture

In most object-oriented methods the focus is on conceptual level (on object). The link between the conceptual models and real-time implementation (system and software) architectures is often missing. In every architecture there are at least two important viewpoints. One static structure view that shows the static dependencies between entities in an architecture. But the behavioural aspects have to be understood as well. The dynamics is a really important topic in real-time systems.

The *object architecture* describes the solution to a problem without dealing with the technology. The software design is a set of objects collaborating to perform some activity. Each object is treated as an independent and concurrent software machine with responsibility of part of the overall functionality of the system. The communication between objects is done by message passing. The object architecture deals with the problem without implementation thoughts and is developed by the use of a set of techniques:

- **Define object domains**
The object architecture is partitioned into object domains, e.g. one application domain that describes the main part of system and several service domains. Service domains can for example be communication services or timing services.
- **Define initial object architecture**
Different techniques can be used to find candidate objects, e.g. from the requirements architecture. These objects can then be used to begin and populate scenarios and collaboration diagrams.
- **Define object message sequence**
This is used to describe how objects will collaborate to solve a requirement. This is done for every use case.
- **Extend object architecture**
In early stages the class diagram can change a lot, but as more and more use cases are analysed the class diagram get more mature.

The *software architecture* captures the specific multitasking aspects of the system. In a system, each processor might have different software architectures. The techniques for developing the software architecture are:

- **Initial software architecture choices**
The initial software architecture choices, e.g. Real-Time Operating System.
- **Define persistence model**
This model captures how persistent data is stored, e.g. in files or in a database.
- **Define concurrence model**
For each processor of the system, a part of the object architecture will be allocated to it. Communication between objects is mapped to communication primitives.

The *system architecture* captures how the system is implemented by using different types of building blocks. The building blocks can be computational elements, communication elements and different types of topologies. Topologies is the type of structure of the communication links that are used, e.g. a star or other type of network. A dynamic view cap-

tures the traffic between computational elements. The techniques for developing the system architecture are:

- **Define system architecture**
- **Map objects to system architecture**
- **Deploy system**

To help the development of a system, a process has been defined. This process is intended to be used with the different models that make up the Real-Time Perspective.

Real-Time Perspective Process

A process is made up of phases and stages, where a stage is broken into steps that's made up of tasks. The RTP Process is divided in three phases, that is Requirements Specification, System Development and System operation. System development is the main area for the RTP. An incremental delivery can improve the value of the system delivered to the user. This is made by for example feedback from user into the next increment delivery. The system development is made up of six stages:

- **Analyse and validate requirements.**
The deliverables from this stage are a formal description of each of the following, system scope, system's modes, systems usage and system constraints. Often the descriptions is supported by prototypes or animations that shows the interaction with the environment. If there is something that's inconsistent in this stage it may be fed back to the requirements specification.
- **Define technical architecture**
The deliverables from this stage is a high level system architecture, key software architecture decisions and initial partitioning of object architecture.
- **Plan increments**
Define the scope of a delivery. This might be a use case, a domain, a system constraint or a node.
- **Design and build increments**
At this stage the software for a particular increment is built and tested. The hardware engineers build and test the hardware for the increment and then the software is tested on it. If any changes to the requirements is discovered at this stage, it is fed back to requirements architecture and to the solution architecture.
- **Accept increments.**
At this stage the increment should be accepted by the project sponsor.
- **Deploy increments**
The increment is installed in the real system.

6.5 Discussion

The different approaches presented briefly here are all encompassing the need to tailoring the UML for real-time purposes. The Real-Time UML approach covered in [13] is a way of introducing object-oriented analysis and design into the real-time systems area. Some of the interesting points is presented in 6.1. One interesting point is the contribution of arrival patterns of messages instead of just showing the synchronisation pattern.

The approach made by ARTiSAN has a set of additional types of models to better support the design of real-time systems and this is implemented in Real-Time Studio. The concurrency diagram looks to be a good approach to capture the concurrency issues. In this type of diagram, priorities can be assigned to processes. This makes it different from the semantics of the ROOM language where priorities only can be assigned to events.

7 A basis for model evaluation

7.1 AIDA

The modelling framework of AIDA is made up of a set of views, e.g. structure and timing behaviour. The emphasis on the multidisciplinary in the design of machine embedded control system, makes it necessary to partition the views into a set of discipline domains, i.e. application models, computer hardware, software and mechanical models. On the other hand with the interdisciplinarity of this kind of system, there must be some links between the models from different domains.

A set of requirements of the models in the framework is formulated [1]:

- The models must support the multidisciplinary characteristics of the design problems where at least both computer (software and hardware) and control engineering is involved. To do this the modelling framework must provide suitable abstractions, views and links between these.
- The model abstractions should be on a relatively high-level to support ‘architectural decisions’ that directly affect the system structure and timing behaviour, and thereby a number of essential system properties such as control system performance (as a function of the system timing), extensibility, testability etc.
- The aim is to give support for the description of realistic control systems. This implies the need to be able to model time- and event-triggered, multirate, control systems with different modes of operation. These control systems are to be implemented on distributed heterogeneous hardware with both serial and parallel communication links interconnecting the processing elements. Furthermore the processing elements can be placed on different locations in the mechanical structure, hence the sites of sensors and actuators should be modelled as well as the assignment of processing elements to such sites. There is also a need for modelling the various system specific overheads, scheduling policies, error handling etc.
- Since design could be based also on existing systems or components, the models must allow description of earlier implementations with many already taken and hence fixed design decisions.

An overview of the framework is given in the table 1 below. In this work the main focus is on a subset of the models in this framework, i.e. the models that resides in the shaded regions of the table. That is because of the limited time and that those models are the easiest to adopt within the limited time of the M.Sc. project

Table 1. Overview of the models in the AIDA framework

| | Application models | Computer system models: Software and Hardware | Mechanical models |
|---|---|--|---|
| Overall behaviour | Modes of operation | | |
| Structure | Functional block diagram | Computer HW structure Process Structure Diagram | Location model Interface component placement |
| Timing Behaviour specification & implementation | Timing and Triggering Diagrams (TTD) Implementation TTD | Inter Process Communications: IPC model, Communication resources, Communication resource chains Processes: Process TTD and Process internal TTD Computer System: Operating system, Scheduling policy, Synchronisation | |
| Component Description | Detailed function description Detailed data flow description | HW characteristics: Processing elements, Communication links, Clocks SW characteristics: Implemented Processes | Interface component model Environment interfaces model |

To capture the structure of the application functionality in a system, in AIDA a functional block diagram is used. It is a common type of diagram with boxes, in this case circles, and lines connecting them, e.g. Simulink block diagram. A functional block diagram can be modelled at different levels of abstraction, one functional block can be decomposed into lower-level functional blocks. The lowest level of decomposition is made up of elementary functions, which are the lowest building units in AIDA terms.



Figure 18. Example of Functional block diagram(a) and TTD (b)

To describe the timing behaviour of the application, the elementary functions are grouped into activities. Activities in AIDA are defined to be a set of elementary functions that execute when triggered by explicit triggers or by precedence. In one activity all elementary functions execute with the same period or, if event triggered, are triggered by the same event and precedence relations. This is captured in a Timing and triggering diagram (TTD). The triggers in a TTD are specified by a set of attributes: $\{Type:Source, [<]Period:[>]Tau, [Tol]\}$. Three types of triggers exists in AIDA, they are:

- **Time triggers (TT)**
- **Event triggers (ET)**
- **Loop triggers (LOOP)**

Tau represents a phase from the period, i.e. a constant time delay between the execution

of two elementary functions. In figure 18b, an example of a TTD is shown. A rate interfacing function (RIF) is used to communicate between a time triggered activity and an event triggered activity or between two time triggered activities with different periods. This diagram specifies the requirements of the timing behaviour of the system.

To show what effect the implementation of the system has on the timing an Implementation-TTD can be used. It is a TTD but with included delays between triggering of subsequent elementary functions. The time delay can be of different types, i.e. constant or some other types. The triggers now shows what timing was achieved by the implementation, e.g. mean period times.

Elementary functions in an activity can be partitioned into processes. Currently in AIDA, no distinction is made between processes and threads. The structure of the processes and their communication, i.e. inter process communication(IPC), is described in a Process structure diagram. The timing of the processes is specified in a Process TTD. In the Process-TTD no rate interfacing functions are shown. To show the communication that is internal to the process a Process internal TTD is used. In a Process internal TTD (Pi-TTD) the communication resources are shown as blocks. The Pi-TTD in figure 19b could for

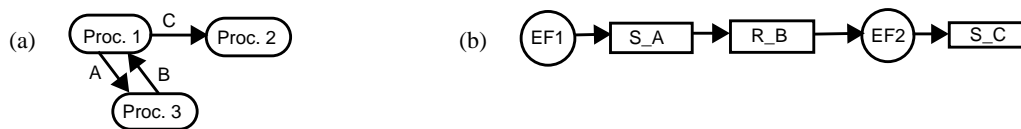


Figure 19. Process Structure diagram (a) and Process internal Timing and triggering diagram (b)

example represent the internals of process one where blocks for inter-process communication are included. For example in figure 19b, the sending of message A is shown as the block S_A and the reception of message B is represented by the block R_B. The blocks represents the interface for communication resources. A communication resource could for example be a message queue which is described separately.

Other models of the AIDA framework do not have a graphical notation. They are instead made up of a list with attributes for describing the model, e.g. for a processing element the attributes are such as speed, size of RAM and ROM, I/O connectors etc.

7.2 Object orientation for distributed real-time system

In ROOM the most general form of the object paradigm is used. That is, an object is defined to be a logical machine that can be incorporated in hardware, software or a manual procedure. A part of the object paradigm is the ability to connect logical machines to each other so it can fulfil the functionality of a whole system.

One approach to capture a relation between functional parts of a system is to see every functional block as an object and show their relations in a object diagram. This is okay to do because an object can represent “anything”, and an object does not have to represent an object defined by a class in some programming language. Instead it can be treated as a modelling element that exists only in the models. A functional block in a functional block diagram is something that has some sort of behaviour, i.e. it performs some actions on an input, and an object also has behaviour. By this an object can be a candidate for doing a

functional decomposition of a system. The functional block diagram also shows the structure of the functions in the system. One approach is to represent it by links that connects functional blocks, i.e. objects.

The objects can be decomposed by using aggregation or composition. Composition is a stronger form of aggregation. The decomposition can be done to an appropriate depth where the objects eventually represent elementary functions in AIDA notation. To see the object as an elementary function is in fact not so good. In terms of implementation, an elementary function is more like an operation or method of an object than an object itself. But treated as an analysis-time construction it's okay.

In 3.2 a set of advantages of object-orientation were listed. Two of them might be of certain interest for real-time systems [13]. They are:

- **Better support for reliability and safety concerns**

It is stated in [13] that with the better abstractions and encapsulation in object-orientation, the interaction between components can be limited to well defined interfaces.

The control of how components interacts increases reliability.

- **Inherent support for concurrency**

Object-oriented systems are inherently concurrent. For example task synchronisation can be represented by using orthogonal substates in a statechart, active objects and object message passing.

One of the stated advantages with object-orientation is the reuse capabilities. The architecture of a system can be built by the reuse of architectural design patterns. In [13] a list of design patterns particularly useful in the design of real-time systems is presented.

Traditionally, object-oriented methods are bottom-up, i.e. find data and operations that affect the data and then form objects. In the design of embedded real-time system with a lot of functionality the approach taken often is of top-down kind, i.e. functional decomposition.

In [26], the software development process for the Mars Pathfinder is described. It was first decided that an object-oriented method should be used. Experiences from this project was that it was hard to find all relevant objects when a true object-oriented process were followed. It wasn't so easy when the system became so large as this was. The textbook examples didn't scale up so well. Instead, a new method called the "meet-in-the-middle" method was developed. In this method they began by doing a functional decomposition, and then used object-oriented methods in the design of a decomposed functionality.

7.3 UML for Real-Time, a combination of UML and ROOM

This approach combines UML with some parts of the ROOM into something that is called UML for Real-Time[15, 16]. From UML the class model and the instance model, i.e. collaboration diagram, is taken. Another complementary view that comes from ROOM is the role model. A role model is a specialisation of a class model and a generalization of an instance model. It shows structural patterns of collaboration that a set of classes is part of,

i.e. shows the roles that classes plays in a collaboration. In figure 20, the role of controller

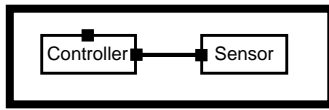


Figure 20. Controller and sensor roles

is to control something by the use of a sensor. The controller can be of different types in different applications as well as the sensor can be of different types. The role of sensor may be realized by an instance of a TemperatureSensor class or by a PulseSensor class. What the diagram says is that a controller always is connected to a sensor.

In UML for Real-Time some stereotypes are defined to extend the standard UML, e.g. capsule, port and connector. A capsule is the same as an actor in ROOM, i.e. possible active objects. Their structural decomposition is captured using collaboration diagrams. A capsule communicates through ports. A port is a realisation of a protocol. Connectors is the same as the ROOM concept of bindings, i.e. gives the relationship of communication between capsules. To avoid confusion, terms from ROOM have been changed to fit into UML, e.g. Message Sequence Chart - Sequence Diagram, Actor - Capsule and Data class-Class. An actor structure diagram in ROOM is a capsule collaboration diagram in UML. In a class diagram the Protocols of a capsule are listed in an additional compartment of the class symbol.

7.4 Criterias for evaluation

To be able to do some comparison of the models the scope has been limited to a subset of the models in the AIDA framework. Another limitation is that the concepts of AIDA is the central framework to be compared with and no detailed evaluation of the AIDA framework is done. But there is a need for a tool and framework to specify and fully describe a distributed embedded real-time system, at least in the application area of control engineering [1, 25].

This thesis is focused more towards how the AIDA frameworks fits in to the languages of ROOM and UML or in general terms how the concepts in AIDA can be ported to object-oriented modelling approaches. To be able to map the models from ROOM and UML on AIDA, some specific issues have to be looked at. In [1], a brief evaluation of different modelling approaches is done. The criterias for making that comparison of different modelling approaches is stated, they are:

- Timing behaviour
- Structure
- Resource Management
- Graphical notation
- Guidelines

In this thesis the criterias that are looked at is Timing behaviour, Structure and Graphical notation.

7.5 Approach for evaluation of models

The approach to the comparison is to look at some of the views in table 1 and try to map these models to UML and ROOM. The chosen views are:

- **Application model**
Structure view, i.e. functional block diagram.
Behaviour view, i.e. TTD and I-TTD.
- **Computer model**
Structure view, i.e. Computer hardware structure and process structure diagram.
Behaviour, only subset of the models, i.e. P-TTD and Pi-TTD.

The intents of this approach is to see how these views are modelled in UML and ROOM. To be able to do a mapping parts of the example system in [22] is used. It is a simple control system with feedback and feedforward functional blocks. The approach might be a bit naive and some of the models are used in a way so that they match the AIDA notation.

8 The AIDA framework vs. ROOM and UML

8.1 Mapping between AIDA and ROOM

As said before and from [5], the structural component in ROOM is the concept of an actor and the behaviour is described by ROOMcharts or by Message Sequence Charts. With those as the building blocks an attempt to a mapping between ROOM and AIDA is made.

Application model - structure view

In object-oriented modelling the composition and decomposition of objects is the main mechanism for model structure. A structure is made of parts that are connected to each other for the purpose to fulfil a functionality. Two important relations in structural modelling are [5], object communication and object containment relationships. The structural parts of the AIDA models are at the application level, see table 1, the Functional block diagram, i.e. blocks and arrows. The functional block diagram has its emphasis on de-

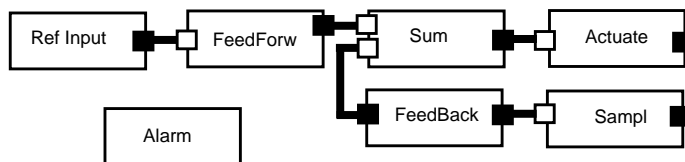


Figure 21. Actors representing functional blocks

scribing the flow of data between functional blocks. This flow of data might in ROOM be captured with bindings that connect actors. One possibility might be to model each function in a functional block diagram as an actor as in figure 21. This figure just shows that there are some connections between the functions. The possibility in ROOM to make use of recursive modelling, i.e. to go inside one block and add more details is exactly the same as what can be done in the AIDA functional block diagram or in an ordinary block diagram in e.g. Simulink. This can be done down to a level where each actor does not contain any more actors, i.e. a leaf actor. To conform to AIDA terms a leaf actor in ROOM is defined to be the same as an elementary function (EF). The use of an actor this way might not fit so well into the concept of actors as representing active objects, but the only way to show structure in ROOM is to use actors.

Application model - behaviour view

In ROOM the behaviour is a part of an actor specification and is captured using ROOMcharts. In AIDA the timing behaviour of the application is captured in a TTD. The functions are separated into activities, so that one activity only contains functions with the same period time. So how can the precedence relations and timing be captured using the ROOM notation? To model the interactions of the functions considered as actors it is necessary to compose them into another actor that can describe the overall behaviour by its behaviour component described with a ROOMchart. This is, because the whole ROOM approach is heavily influenced by the possibility to make executable models at all levels of the model. An alternative could be to use MSCs.

One approach for capturing the precedence relations of elementary functions, i.e. leaf actors in the fully decomposed application structural model, is to encapsulate them in a composite actor and use that actors behaviour component. This composite actor represents the

activity that those EFs are involved in. Assume that all the actors in figure 21 are leaf actors or EFs, then they can be separated into activities. In figure 22, four of the functions in figure 21 have been separated into an activity. The component actors contained in the actor ControlActivity are communicating directly with the behaviour of its containing actor. For the purpose of showing the behaviour in terms of timing and precedence relations, as is the meaning of a TTD, a ROOMchart might be useful. In that case a state represents the invocation of a leaf actor's functionality.

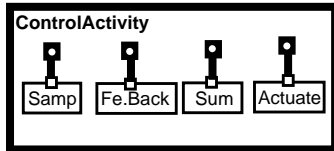


Figure 22. Leaf actors that constitute an activity

Explicit triggers can be shown on this diagram, e.g. TT or ET. To define a time triggered activity in ROOM, a period timer has to be started on the creation of that activity. This can be done by defining the initial transition to perform an action that sets the timer and goes into the idle state, see figure 23. An approach for defining periodic timers in ROOM is made in [7], where the definition of the periodic timer is given by $Tp:(T,D,C,A,X)$. T is the period for the timer, D is the deadline when the triggered function has to be finished, C is the maximum expected computation time for the triggered function, P is the priority of the signalled event and X is a data item returned with the timeout message. When such a timer is started by the Timing Service it continues to load after it expires until cancelled explicitly.

In an activity there is a possibility to have EFs that are triggered with a phase shift relative to the first triggered function of that activity. To realise this, another timer is started at the transition between the idle state and the execution of the first elementary function, i.e. the state corresponding to the first leaf actor to perform its functionality. This timer is not of periodic type but rather of oneshot timer type and therefore has to be loaded at the same transition every time the activity starts executing. The transition between the states Sum and Actuate in figure 23 is triggered by the event that occurs when the phasetimer times out.

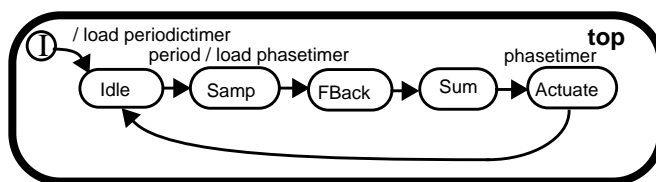


Figure 23. ROOMchart that captures precedence relations for EFs in the activity in figure 22

Compare the constructs above with the AIDA approach where the triggers and their specifications are given explicitly. A trigger in current AIDA notation is defined as $\{Type:Source,[<]Period:[<]Tau], [Tol]\}$, where Type is either TT, ET or LOOP. Source is the triggering event or a clock. Tau gives the possibility to specify a specific phase from the period Tol specifies the tolerance. Attributes like those are needed [1] to give a good description of embedded real-time system. So for the case of making a description or a specification of a embedded real-time system, the AIDA approach is more clear in showing the timing and precedence relations.

Another way to describe system behaviour is the use of message sequence charts. That 's not a big topic in current version of the ROOM language. In ROOM a MSC is used to capture a rather high level behaviour of a system. In fact it is used to show some scenarios that the system can be in. For the purpose of describing the behaviour as described in the parts above, its possible to model the timer as an actor and the message passed from it trigger the sequence of executing leaf actors, see figure 24. This can give a representation of

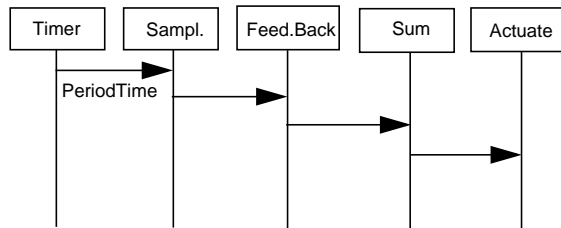


Figure 24. MSC that captures the execution order of EFs

the execution order of the leaf actors, but there is not much said about the possibility to add more explicit information in such a diagram, i.e. information like the one given in a TTD in AIDA. Of course that can be done by just adding textual notes giving the information, but that is rather a matter of tools and not of the language. Adding notes to a diagram is of course a good way to describing something in more detail, but that might of course insert some violations to the formal semantics. But that is another issue to discuss the formal meaning of text.

Computer model - structure view

In this view one of the models in AIDA captures the computer hardware structure of the system. In ROOM this can be done by using the structural concept of actors. By making a diagram with the physically distributed hardware components represented by actors, the structure is modelled. An actor can represent a physical object as well as software objects.

The functions in a functional block diagram in AIDA are partitioned into processes that can be allocated to processing elements. To use the concepts of ROOM this can be done by encapsulating the different function blocks/function actors into another actor. The question is what this means in matter of implementation. Every composition of actors has to be encapsulated by a top level actor. But actors are active objects with their own thread of

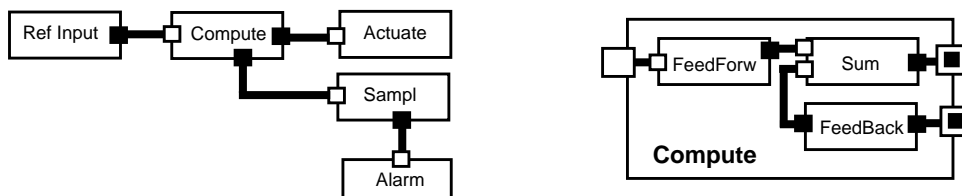


Figure 25. Grouping of leaf actors to processes

execution and the functions in a functional block diagram might be partitioned to processes that can be allocated to different physical processing nodes. Figure 25 gives the view of the functions partitioned into processes. Here every top-level actor represents a process, the actor Compute represents a process that is made up of the contained, in this case, leaf actors. The details of the components of process Compute does not have to be shown in

this view, so that can be omitted. Instead its details are shown in a more detailed definition of the Compute actor.

Computer model - behaviour view

In AIDA the behaviour view of the computer model is by part made up of process timing and triggering diagram. This can be described with the behaviour component of the containing actor for the actors at process level. There are, in the aspects of graphical notation, no difference between the “function level” or on the “process level”. It is just to encapsulate actors to form different views of the systems. The same approach, as for the application models, can be adopted here.

Discussion of the mapping between AIDA and ROOM

One of the main issues in ROOM is the ability to construct executable models and be able to generate code. It is not obvious how the approach to divide and partition the actors to different structural views, e.g. process versus activity, is treated when implemented from the tool. This might be a problem because the interobject behaviour modelling in ROOM is quite weak as it appears. It looks like that approach when using ROOM is not so good.

The approach suggested above is not so clear. It’s not a good solution to use a ROOMchart for specifying the behaviour of a system in terms of timing and triggering relations. It is hard to express requirements on a timing behaviour since a ROOMchart is based on events and not timing. Of course a timeout event can be generated, but this implies that a transition is taken when the timeout occurred and that may be too late in a real-time system.

It is a bit strained to do it the suggested way, but a ROOMchart is the only construct in ROOM that has the possibilities to show behaviour. ROOMcharts, or in general any type of state-machine, is normally used for capturing intraobject behaviour. This is mentioned in the foreword of [13]. The contrast to *intraobject* behaviour is *interobject* behaviour, and this can be captured with a type of diagram that depicts a sequence of messages passed in a collaboration.

To express the same things with ROOM as in a TTD in AIDA, it is necessary to make two diagrams in ROOM. First to group and encapsulate the actors that form an activity and then use a ROOMchart to describe its precedence relations. A TTD is both the grouping and the timing behaviour in the same diagram without cluttering the view.

The use of ROOM and its structural parts is okay. At least it looks okay in terms of just showing structure of system. It is possible to use it for a functional decomposition of the system, since it allows hierarchical structure. But as said before, an actor represents an active object and it can be a problem to treat an elementary function as an active object.

The semantics of actors and bindings can further be examined to see how that conforms with the AIDA approach. It might be possible, in a specification, to put some attributes to a binding and actors.

8.2 Mapping between AIDA and UML

UML is, as said before, a general modelling language with many different diagrams to

represent different views in a system. The models in the AIDA framework are at this moment not stated to be object oriented. In this part an approach to expressing AIDA models by using the notation available in UML is made. The semantics of the UML model elements might be violated in this approach, but the intents are to keep the approach as semantically correct as possible.

Application model - structure view

The structure view of the application model in AIDA is a functional block diagram. This is a kind of data flow diagram, but in UML there is no data flow diagram type. One approach can be to show the structural relationship between functions with an object diagram. An object in figure 26 doesn't have to represent an instance of a class in some

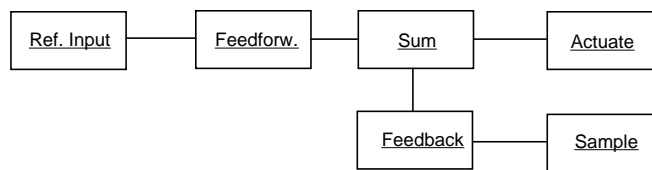


Figure 26. Structural relations between functions

programming language, it's just used for modelling. The Feedforw object has some link to the Sum object, this is the same as the links between functional blocks in a functional block diagram. It shows the structural relations between the objects.

Application model - behaviour view

When the functional model is decomposed, i.e. every block consists of elementary functions, the elementary functions are grouped into activities. The elementary functions for one activity are composed into an object with a stereotype <<activity>>¹. At this point the <<activity>> stereotype only means that this is an object that represents an activity at modelling time. This means that the elementary functions in an activity can be represented in more than one object, i.e. in different views of the system. This is okay if the objects are treated as a form of container for modelling constructs. In a Timing and Triggering Diagram the triggering and precedence relations of an activity are shown. So far the grouping of the elementary functions to an activity object just show the structure of the activity, i.e. just shows which objects participate in a particular activity. The possible approaches to show the behaviour of an activity with the UML are:

- **Activity diagram**
- **Sequence diagram**
- **Collaboration diagram**
- **Statechart diagram**

An *activity diagram* is like a statechart diagram but the states are activities that represent operations. In an activity diagram the transitions are made by completion of the operation

1. Introduced here.

in the preceding action state. The activity states in figure 27a represent the activation of

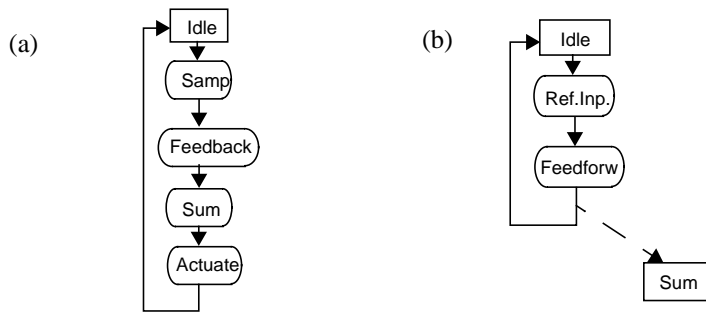


Figure 27. Activity diagrams showing the execution order of EFs

the elementary functions. The dotted arrow from the transition between feedforw and idle to the sum object is to show an object flow, i.e. a flow to an object that is an input or output to an activity. This type of objects can be treated as a rate interfacing function because usually such an object often is input to subsequent actions. The partitioning of the elementary functions in the activities to processes can be done by using swimlanes. Then a partitioning of the elementary functions in the activity in figure 27 can be like in figure 28. This shows three processes and parts of their contents. It shows how the activity in figure 27 is partitioned to these three processes. It is possible that the processes have other elementary functions that are parts of another activity. Each swimlane represents responsibility for part of the activity and may be implemented by one or possibly more than one object per swimlane. These objects are good candidates for active objects, i.e threads. So

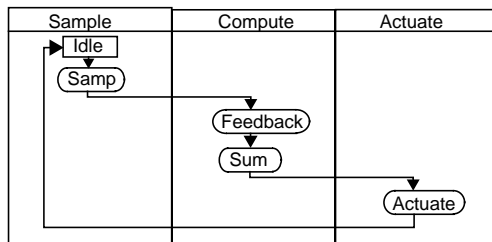


Figure 28. Process diagram

far this approach only shows in what orders a set of elementary functions are executed. The detailed triggering information that are given in a TTD has to be specified in some way. The easiest way to do this is to add a textual note to a state symbol, this is informal in the way that it doesn't show the mechanisms for the triggering of an activity. If this type of diagram is supposed to provide a description and a specification of the behaviour that might be enough. The AIDA framework is supposed to be used at high-level design of systems and with this in mind the use of textual notes for specification is alright.

A *sequence diagram* is another approach to show execution order of operations or functions. It is used so that messages are sent to trigger the invocation of a subsequent function. By populating such a diagram with the objects from an activity object the execution order of the elementary functions in that activity can be shown. To explicitly express the triggering information in this kind of diagram an approach is to define a trigger object. That is an object stereotyped with a <<Trigger>> stereotype and the properties on the

trigger given as tagged values. Figure 29 shows how a trigger object could look like. The

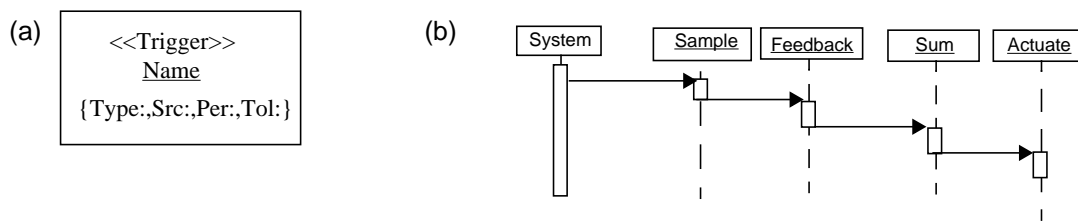


Figure 29. Trigger object (a) and a sequence diagram (b)

triggering specification is not a property of an object but rather on a message/event that triggers a function. When the models then are used for describing the system, the properties from the specification can be expressed by adding a textual note to that particular message. The use of a trigger object can be used for time-triggered activities, because a message must have a sender. It is although better to have the specification on the message that generates an event. It can give the wrong impression to use a special triggering object, i.e. an object that triggers a sequence of actions might have a lot of other functionality besides that of the triggering.

The approach to use a *statechart diagram* is the same as in the ROOM mapping. This is because there aren't any big differences between a statechart diagram in UML and a ROOMchart.

A *collaboration diagram* basically shows the same information as a sequence diagram and is therefore not examined further in this mapping. The differences are that a collaboration diagram shows the associations between objects while the sequence diagram do not and the sequence diagram includes the dimension of time while collaboration show the order in which messages are sent.

Stereotypes can be used on any kind of model elements and therefore it is possible to define the <<Trigger>> stereotype and the properties can be applied to a message. Compare this with the <<Signal>> stereotype which is a standard stereotype in UML. It is applied to a class that defines a signal that can be used to trigger transitions. Such a class has its parameters shown in the attribute compartment of the class definition. A signal is not allowed to have any operations. A trigger can be defined in the same way with the addition of the more detailed triggering information represented by the properties of the tagged values.

Computer model - structure view

This view shows the structure of the processes in the system. In the behavioural view of the application model an initial grouping of the elementary functions to processes are made. From this a set of candidate processes exist and their structural relations can then be shown using an object diagram. This was done in the functional decomposition of the application model and now the same notation can be used to represent the structure of the processes. In UML the stereotypes <<process>> and <<thread>> exist and can be used

in this model to point out that is the structure of the processes that are shown. Figure 30

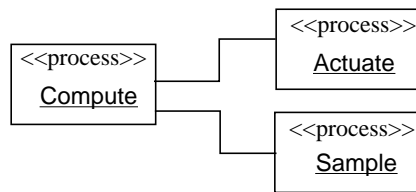


Figure 30. Process structure view for example system

shows an example of such a model.

Another aspect of the structural description of the computer system model is the computer hardware structure. This model shows the processing elements and their communication links and to which sensors and actuators a processing element eventually is connected to. This can be modelled by using a UML deployment diagram. It is made up of nodes and links that then can be stereotyped or other information can be added. The properties of a particular link can be added to the diagram by using a textual note. This is just a construction to be able to communicate the description of the system in a clear way. To be able to make an analysis with some back-end tool, the detailed description is also added to the modelling element in some appropriate way.

In figure 31, a possible deployment diagram is shown. It is made up of a Control node connected to a Mother node by a CAN communication bus. Some information about the CAN bus is given in a textual note. In another more elaborated deployment diagram the processes that are discovered in the application model can be added, by drawing the processes symbol inside the node symbol, to show allocation of processes to the different nodes. This is shown by d

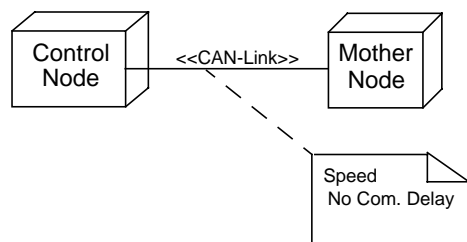


Figure 31. Deployment diagram with specification of the communication link

Computer model - behaviour view

The behaviour at this level is in AIDA expressed by a process timing and triggering diagram. The approach to make the model at this level is to use a sequence diagram showing the order of the activation of the processes. This approach is better suited for this type of specification. The alternative approach with the use of an activity diagram doesn't seem to be so well suited for this level of modelling. That is because an activity diagram is describing the behaviour of a class that is participating in a behaviour. In the process behaviour view a set of processes are interacting and if an activity diagram is to be used the processes have to be grouped to an object that is the owner of an activity diagram.

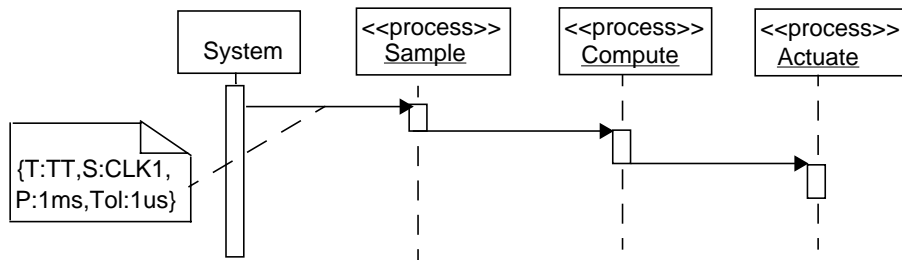


Figure 32. Process timing and triggering diagram with UML notation

In figure 32 an example of this model is shown and here the explicit triggering specification is given in a textual note. With this type of diagram the same type of information as a timing and triggering diagram is given. By using the UML notation it should then be easier to continue and refine the models towards an implementation.

The mapping between UML and AIDA

The Unified Modelling Language is a rich language with possibilities to extend it or make variants so it fits into a certain application field. Maybe it is too rich in notation and too weak in the semantics. It is pointed out in the foreword of [13], that UML in its current appearance is too massive, which makes it hard to understand all parts of it.

One problem with the UML approach to express the AIDA models is the way of representing elementary functions by objects in an object diagram. It might not be the most natural way in UML to make a functional decomposition to find the elementary functions. This is of course based on the method that is used. Therefore the approach can be okay if it is treated as a way of finding the elementary functions. On the other hand, of course other methods could be used or just an informal “box and line” model can be used to find elementary functions.

An object diagram with the `<<activity>>` objects show all the elementary functions and the composition to activities. Remember that the `<<activity>>` is defined to represent a modelling time construction. This solves the problem of showing the elementary functions when describing the system. Then the activity diagrams can be useful to describe the triggering and precedence relations of the elementary functions in one activity. To describe the partitioning of the elementary functions to processes with the use of swimlanes is a good approach, but the semantics of the transitions that cross swimlanes might require an extension to allow the swimlanes to represent active objects.

8.3 AIDA and the best of OO

There isn't an obvious way to incorporate object-oriented concepts, specifically UML and ROOM, to match the intents of the AIDA framework. But with a combination of the best of each, a step towards object-orientation of AIDA may be taken.

There are good possibilities to capture structural relations with object-orientated concepts. Functional decomposition with the use of the actor concept from ROOM or a capsule in UML-RT, match the functional block diagram of AIDA quite well in matter of showing hierarchical structure of the functions in an application. The problem is that an actor is an active object with its own thread of execution and this may put some restriction on how it can be used for functional decomposition.

The intraobject behaviour of an object can be captured with statecharts, for which the semantics is formal and quite well explored. Both the ROOMchart and the Statechart in UML are well suited for this. On the other hand for interobject behaviour there is not a lot of modelling capabilities in ROOM. UML has different kinds of diagrams to show this, i.e. collaboration diagram, sequence diagram and to some extent the activity diagram. For the purpose of giving explicit timing requirements, the best choice is to use a sequence diagram with timing marks. Hardware structure and mapping of threads to processors can be shown with deployment diagrams in UML. The interaction between processes can be shown either in a sequence diagram or in a collaboration diagram, but as with normal object interaction the choice of using sequence diagrams is better to capture the timing behaviour.

The best match of object-orientation and AIDA:

- **Structure**
 - Class and object diagrams (could be actors)
 - Hardware structure / mapping threads to processors shown with deployment diagram
 - Task interaction can be shown in collaboration diagrams with active objects

- **Behaviour**
 - Statemachines for intraobject behaviour
 - Sequence diagrams for interobject behaviour

9 Conclusions

9.1 The approach to the work done

In the beginning of the work behind this report the focus was on the graphical notation of the models in the AIDA framework. Instead the focus probably should have been on the contents of the models, i.e. what they represented, and from this viewpoint try to make it object-oriented. The approach to look at two different languages may have been a bad choice. It might have been better to combine the theoretical part with more practical work. Anyway, the somewhat naive approach taken have given some insight in the difficulties with different methods and models. Probably there are many other possibilities to do a mapping between the AIDA concepts and UML/ROOM.

9.2 Results

The result of this work is an overview of UML and ROOM and a mapping of them to the models in the AIDA framework. The results of the mapping is presented below.

Application Structure:

- From the mapping of ROOM to AIDA the conclusions are that the structural modelling capabilities in ROOM are good. It can be used for functional decomposition, but with the restriction that all actors represents an active object with its own thread of execution. At modelling time the graphical notation for an actor can be used as a “box and line” model. The question is then if it’s necessary to use the actor concept. The intention of ROOM is to make a formal language with executable models at all refinement levels.
- Conclusions drawn from the mapping between UML and AIDA is that structure can be shown in for example an object diagram. This can be used in the same way as an actor structure diagram in ROOM, with the difference that in UML the object doesn’t have to represent an active object. It is possible also in UML to decompose an object into component objects, so this may lead to a better approach than the ROOM concept of actor decomposition. Another point in this approach is a proposed stereotype, i.e. `<<activity>>`, representing a modelling time partitioning of elementary functions to activities. This approach is perfect if the functional decomposition of the objects is supported in a tool.

An alternative or perhaps a complementary view could be a collaboration diagram with messages but without the numbers. This could show that a function passes messages to another function.

Computer Structure:

- An actor can be used to group elementary functions/leaf actors into a composite object that represents a process. This is a perfect match for a process structure diagram since the actor represents an active object. One limitation might be that the component actors also are active. There isn’t any graphical notation for other types of objects in ROOM.
- In UML an object diagram or a collaboration diagram can be used to capture the process structure. This perfectly matches the intents of the process structure diagram. It’s possible to stereotype the object so it’s clear that it represents a process or thread. This

is better than the approach with ROOM, since it's possible to represent ordinary objects graphically in UML. It's also possible to show the allocation of elementary functions, represented by an object, to a process by aggregation or composition.

- Hardware structure and the mapping of threads to processors can be made in a UML deployment diagram. This is a perfect match to the intents of the description of the hardware structure.

Application Timing behaviour:

- The behavioural modelling in ROOM is made with *ROOMcharts* and the approach presented in this thesis is to load timers at specific states or transitions. It's a common opinion that statecharts are good for capturing event-driven behaviour and not so well suited for the modelling of requirements on timing behaviour. The loading of a timer and then a reaction to a timeout event is not good for capturing the type of requirement on timing behaviour that's needed in the description of an embedded distributed control system. Further, it's hard to model the behaviour of interacting objects with a ROOMchart, it's basically showing the intraobject behaviour and the reaction to events. A ROOMchart, or in general terms a statemachine, isn't good for capturing requirements on timing behaviour.

A complement or an alternative to a ROOMchart for expressing behaviour could be to use a *Message Sequence Diagram*. Even if the concept of MSCs isn't examined so much in ROOM it treated as an alternative for showing sequences of functions.

- It's possible to use the *activity diagram* in UML, where the activities represents elementary functions. Even if this approach matches the intents of a TTD better, there may still be problems in expressing timing requirements explicitly.

Sequence diagrams are good for capturing the sequence of executing elementary functions/leaf objects. It's possible to use timing marks on messages to express requirements on timing behaviour. This is the best identified alternative to capture the timing on interobject behaviour.

The same problems as with a ROOMchart applies to a *Statechart* in UML. It's good for reactive behaviour and not for requirements on timing behaviour.

Collaboration could be an alternative to a sequence diagrams but since the focus is on the order of the messages sent, it's not good for the timing behaviour.

Computer Timing behaviour:

- To show the interprocess behaviour, the approach with ROOM was the same as for the description at the application level. But the same conclusion is drawn here, that a statemachine isn't good for capturing timing and triggering requirements.
- To describe the timing and triggering information for the processes with UML, the approach is to use a *sequence diagram*. This is much the same approach a for the timing and triggering description at the application level, i.e. a TTD in AIDA. This is a good approach.

The best approach

As said in section 8.3 of this report the suggestion for the best mapping of object-oriented concepts and AIDA is:

- **Structure**
Class and object diagrams (could be actors)
Hardware structure / mapping threads to processors shown with deployment diagram
Task interaction can be shown in collaboration diagrams with active objects
- **Behaviour**
Statemachines for intraobject behaviour
Sequence diagrams for interobject behaviour

Graphical notation

- UML has a focus on graphical notation and it might be too massive in expression possibilities [13]. On the other hand, the possibility to extend the already massive language perhaps makes it useful in many application domains.

9.3 Suggestions for further work

Since there are some stated advantages when using object-oriented methods, e.g. reusability, abstraction and extension, it's necessary to try to incorporate this into a framework. If it's possible to use object-orientation throughout the process, discontinuities of different kinds are not introduced. There is although opinions that object-orientation lack possibilities to develop time-driven real-time systems. Therefore, one obvious suggestion for the future is to look at the upcoming UML profile for Scheduling, Performance and Time from the OMG.

A decision then has to be made if the framework should be based on UML, ROOM or any other object-oriented modelling language/method. In the case of choosing between UML and ROOM, the suggestion is to use UML. The same concepts that exist in ROOM, exist in UML for Real-Time [15], e.g. the concept of actors as active objects with an encapsulation shell that makes them highly reusable.

The work presented in this report has scratched the surface at many types of diagram. A suggestion is to choose the best candidate models and then examine the semantics more deeply.

Another direction for further work is to examine existing tools, e.g. ARTiSAN Real-Time Studio, and investigate the possibilities to make interfaces between different tools. One subject for further investigations is if the Case Data Interchange Format (CDIF) is something that can help in making these interfaces. As presented in [27] there are strong possibilities to exchange models between different domain specific tools.

One solution could be to use Simulink for making the functional decomposition of the application and then transfer these models to a tool where additional information on the blocks are added. For example the blocks could be represented by objects in a UML tool and the timing and triggering information could be added to a sequence diagram. Then this information is transferred to some analysis tool that can extract the additional information in an appropriate way.

10 References

- [1] Törngren M., Redell O., *A Modelling Framework to Support the Design and Analysis of Distributed Real-Time Control Systems*. To appear in Journal of Microprocessors and Microsystems, Elsevier., 2000
- [2] Fox A.M., Cooling J.E., Cooling N.S., (1999), *Integrated design approach for real-time embedded systems*, IEE Proc.-Softw., Vol. 146, No.2, April 1999.
- [3] Kruchten P., *The 4+1 view model of architecture*, IEEE Software Nov. 1995, 12(6), pp. 42-50.
- [4] Bortolazzi J., Hirth T., Raith T., (1998) *Codesign in Automotive Electronics*, Society of Automotive Engineers
- [5] Selic B., Gullekson G., Ward P.T., *Real-Time Object-Oriented Modeling*, John Wiley & Sons, 1994
- [6] Bourgoyne D.W., Gresham J.L., (1999), *Object-Oriented Modeling for Embedded Print Engine Control*, www.objecttime.on.ca/otl/products/modeling.pdf
- [7] Selic B., (1995), *Periodic tasks in ROOM*, www.objecttime.on.ca/otl/products/periodic.pdf
- [8] Rehtin E. and Maier M.W., *The Art of Systems Architecting*, CRC Press, N.W., 1997, ISBN 0-8493-7836-2
- [9] UML Summary, version 1.1, 1 September 1997., www.omg.org/docs/ad/97-08-03.pdf
- [10] UML Notation Guide, version 1.1, 1 September 1997., www.omg.org/docs/ad/97-08-05.pdf
- [11] UML Semantic, version 1.1., 1 September 1997, www.omg.org/docs/ad/97-08-04.pdf
- [12] Alhir S.S., *UML in a Nutshell*, O'Reilly & Associates, 1998, ISBN 1-56592-448-7
- [13] Douglass B.P., *Real-Time UML: Developing efficient objects for embedded systems*, Addison Wesley Longman, 1998, ISBN 0-201-32579-9
- [14] Bourdeau E., Lugagne P., Roques P., *Hierarchical Context Diagrams with UML: An Experience Report on Satellite Ground System Analysis*, Lecture Notes in Computer Science 1618, The Unified Modeling Language <<UML>> '98 : Beyond the Notation, ISBN 3-540-66252-9 Springer-Verlag Berlin Heidelberg New York
- [15] Lyons A., *UML for Real-Time Overview*, April 1998, www.objecttime.com/otl/technical/umlrt_overview.pdf
- [16] Selic B., Rumbaugh J., *Using UML for Modeling Complex Real-Time Systems*, March 11 1998, www.objecttime.com/otl/technical/umlrt.pdf
- [17] OMG, *RFP: UML Profile for Scheduling, Performance and Time*, www.omg.org/docs/ad/99-03-13.pdf
- [18] Lanusse A., Gerard S., Terrier F., *Real-Time Modeling with UML: The ACCORD Approach*, Lecture Notes in Computer Science 1618, The Unified Modeling Language <<UML>> '98 : Beyond the Notation, ISBN 3-540-66252-9 Springer-Verlag Berlin Heidelberg New York
- [19] Moore A., Cooling N., *Real-Time Perspective Foundation*, Version 1.1 1998, www.artisansw.com
- [20] Graham I., *Object-Oriented methods*, Addison Wesley, 1994, ISBN 0-201-59371-8
- [21] Douglass B.P., *UML Statecharts*, Embedded systems programming, January 1999
- [22] Redell O., Törngren M., (1998), *Preliminary Design of Models for the AIDA toolset*
- [23] www.damek.kth.se/~martin/aida.html
- [24] Cooling J.E., *Real-Time Software Systems*, International Thomson Computer Press, 1997, ISBN 1-85032-274-0
- [25] Hanselmann H., *Development Speed-Up for Electronic Control Systems*, www.dspace.de/ftp/papers/conv98_e.pdf
- [26] Stolper S.A., *Streamlined Design Approach Lands Mars Pathfinder*, IEEE Software, September, 1999
- [27] Burst A. et al., *A Rapid Prototyping Environment for the Concurrent Development of Mechatronic Systems*, www-itiv.etc.uni-karlsruhe.de