RWTH Aachen

Department of Computer Science

Technical Report

MeDUSA -

MethoD for UML2-based Design of Embedded Software Applications

Alexander Nyßen and Horst Lichter

ISSN 0935–3232 · Aachener Informatik Berichte · AIB-2007-07

RWTH Aachen · Department of Computer Science · May 2007

MeDUSA

MethoD for UML2-based Design of Embedded Software Applications

Alexander Nyßen und Horst Lichter

Research Group Software Construction
RWTH Aachen University, Germany
Email: {any, lichter}@cs.rwth-aachen.de

Abstract. MeDUSA (Method for UML2-based Design of Embedded Software Applications) is a model-driven software design method targeting the domain of small embedded systems, especially field devices.
Being Use Case-driven, MeDUSA systematically covers the software development lifecycle from the early requirements up to the late detailed design modelling. Models are successivly developed and employed throughout all activities.
By enforcing an object-based rather than an object-oriented design, a smooth transition of the resulting detailed design model towards an implementation in a procedural programming language is facilitated. This is essential, as procedural programming languages as the C language are still state-of-the-art in the regarded domain.
By leading to a component-based architectural design, MeDUSA explicitly addresses the reuse of components, something that is the prerequisite for the application of the method in a product-line setting. This has gained significant importance to the industrial practice in the last years.
MeDUSA was developed by the Research Group Software Construction of the RWTH Aachen University in close cooperation with the German ABB Research Centre in Ladenburg. It incorporates various practical experiences gained during the industrial development of embedded software in ABB Business Unit Instrumentation.

1 Introduction

1.1 Characterization of the Application Domain

Regarding its applicability, the domain covered by MeDUSA can be characterized as software development of small embedded devices. However, as this application domain is rather broad - and even if we think that MeDUSA would be applicable to quite a lot of its different sub domains - understanding the method and its characteristics can be best achieved by taking into consideration the domain MeDUSA was initially developed for, namely that of software development for field devices.

Field devices are rather small embedded systems that are integrated into an often large process automation plant. They are used across various industries such as food, chemicals, water and waste water, oil and gas, pharmaceutical, and others. Most of them occur in many different variants. Measurement devices for example, which are one sub category of field devices, occur in different product variants concerning the physical quantity they measure (temperature, pressure, flow), the measure principle applied, the communication capabilities offered, as well as the safety and reliability constraints accomplished.

1.2 Requirements and Objectives

All field devices do have in common that they can be characterized by rather strong resource constraints regarding memory, power consumption, and computing time. Thus, object-oriented programming languages are not yet the first choice and C is still the main implementation language in the regarded application domain. Any design method being applicable to the domain should therefore allow a smooth and rather direct transition from a detailed design into a procedural implementation in the C-language.

The large extent of variety, field devices occur in, does precipitate that the development of them is done - or is at least intended to be done - in a product line approach. The software of most measurement devices is for example developed on top of a common - product unspecific - platform covering basic services and hardware interfaces, into which product specific components regarding the measurement task, which is strongly dependent on the physical quantity to measure as well as on the measurement principle applied, have to be integrated. A method supporting the development of software for such devices should therefore be capable of supporting distributed development of software components, so that platform as well as product specific components can be developed in a distributed manner and can then be easily integrated.

The last basic requirement posed on any method targeting the regarded application domain is, that the notation employed is not proprietary but best based on an industry standard - or at least de facto standard. There are several reasons for this. First the development of software for safety critical application areas requires the application of standards wherever possible. Second, the application of a standard best enables the communication in a distributed development organization, as training of developers can be easily achieved. Last, a large number of standard-conformant tools is available from which a selection can be made when assembling a tooling infrastructure.

Having all that in mind, our intention was to develop a design method that fulfils all those requirements while also considering the expectations and practical experience

of application developers. Having already gained practical experience with the application of the object-oriented COMET method [Gom00] in the domain [NMSL04], we regarded it to be an adequate starting point. Thus MeDUSA tries to transfer all of COMET's advantages to the regarded application domain while trying to expunge most of its shortcomings identified during its practical application.

In detail, COMET

- does not reflect the rather manageable complexity of the regarded devices' software, which allows to model the run-time structure directly in terms of objects rather than indirectly in terms of classes.
- facilitates an object-oriented design, what deteriorates the transformation to a non object-oriented implementation in the C language.
- does not facilitate reuse of artifacts from prior development projects, as a systematic selection and integration of reusable components is not addressed.
- does not support UML2 notation, which is the current standard implemented by state-of-the-art modeling tools.

Further COMET introduces some overhead by dealing with aspects that are not applicable to all extend in the regarded application domain.

1.3 Characteristics of MeDUSA

Acting on the maxim that model-driven development for small embedded systems should allow a seamless transition from the design model to an implementation in the C-language MeDUSA was - unlike COMET - designed to be an object-based rather than an object-oriented method. That is inheritance and polymorphism are disregarded until the late detailed design, as the conception of classifiers is done not earlier than during this step (to be precise this holds for all objects forming the internal decomposition of a subsystem, as stated above). The application of generalization/specialization concepts is not enforced during earlier steps of the method. Therefore those concepts may even be omitted during detailed design to allow a more straightforward implementation of the detailed design model.

Taking into consideration that the run-time structure of software systems in the regarded domain is mostly rather small - being comprised of only a few subsystems and a manageable amount of objects - MeDUSA was designed to be an instance-driven method. That is, during all steps of the method, from the early analysis up to the late detailed design, the modeling of objects (or more precise classifier instances) rather than the modeling of classifiers is enforced. This allows the architectural design of the system to be directly captured in terms of the system's run-time structure rather in an abstracted classifier-based view on it and does - according to our practical experience - accommodate the intuitive understanding of the application designers and developers.

Due to the fact that the main focus of the method resides on modeling the run-time structure of the system rather than modeling the static classifier structure, the enhancements and additions the UML introduced with its new standard version 2 are quite beneficial [NL05]. The newly introduced composite structure diagrams for example are very well suited to cover the run-time structure of a system's subsystems. Because of this - and because of the tool landscape which is currently shifting to the new standard release - MeDUSA was conceptually designed to employ the latest UML version as its notation.

1.4 Applied Notation - Meta Model

The MeDUSA method definition is structured according to the UMA (Unified Method Architecture) meta model developed by IBM [Hau06]. UMA is an advancement to the OMG Standard SPEM 1.1 (Software Process Engineering Metamodel [SPE05]), which has found broad acceptance in practice. UMA is also IBM's and other OMG partners' candidate for the SPEM 2.0 standard of the OMG. As, at the time of this report's publication, the SPEM 2.0 standard has already reached the state of a Draft Adopted Specification ([SPE06]), we regard it as likely that the MeDUSA specification documented herein will conform - incorporating smaller corrections - to the new upcoming version of the standard.

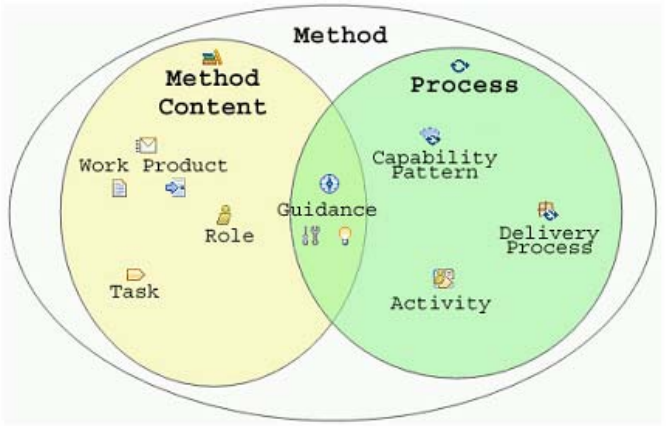


Fig. 1. UMA Terminology Overview (taken from [Hau05])

The main characteristic of the UMA (as well as SPEM) is the division of a method definition into (method) content and process, as denoted by Figure 1. The method content defines tasks, roles performing those tasks, work products serving as input or output of tasks as well as guidances, but does mask out how these tasks are executed over time. This is indeed specified by the process, which defines how the artefacts of the method content are employed in activities, iterations, phases, and processes. By its clear separation into method content - what to do - and process - when to do it - the UMA terminology does support the reuse of content for different processes, so that customized processes may be defined for each usage scenario and organizational setting.

1.5 MeDUSA Example System

To enhance the understandability of the report, a continuous example seems to be quite helpful. The system we will consider as a running example is of course a field device. To be more concrete, it is a small electromagnetic flow meter that is used to measure the flow rate of a liquid floating through a pipe. The physical measurement principle of such a device is rather simple. It is based upon the principle that an electric conductor,

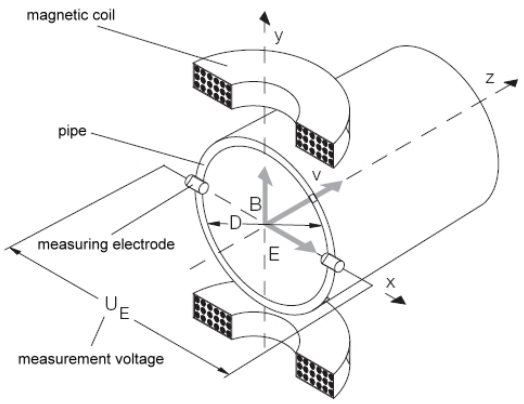


Fig. 2. Physical measurement principle of an electromagnetic flow meter (taken from [GHH+04])

being moved through a magnetic field, induces a voltage orthogonal to the direction of the magnetic field and the direction of its movement. The electromagnetic flow meter makes use of this *law of induction*, as it creates an electromagnetic field around the pipe, through which the measured liquid will flow, like shown in Figure 2. In case the liquid is a electric conductor, the induced voltage can be measured by electrodes. From the measured induced voltage, which is referred to as the *raw flow velocity*, the flow velocity (in m/s), and - having knowledge about the diameter of the pipe - the flow rate (in l/s) of the liquid can be computed.

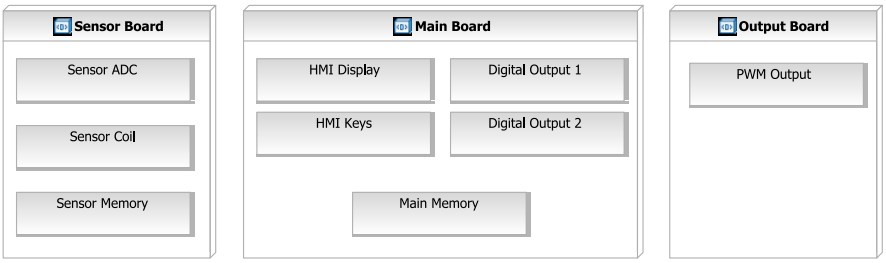


Fig. 3. MeDUSA example device's hardware

From a hardware viewpoint, the example measurement device was designed to be split into three distinct boards, as shown in Figure 3. The first board, the so called sensor board, is responsible of driving the coils which create the electromagnetic field needed for the measurement. It also measures the raw flow velocity with the help of two electrodes, connected to an ADC (analog digital converter). The main board is responsible of performing the signal processing, that is computing flow velocity and flow rate from the raw flow velocity provided by the sensor board. It controls the HMI (human machine interface), realized in form of some interaction keys and a small display, which is used to output the measured flow rate as well as alarms, which may occur during the measurment or signal processing, on an operator frame. The HMI is

further used for viewing and editing configuration parameters relevant to the device. Last two digital outputs, the device is equipped with, are controlled by the main board. They are responsible to transfer the measured flow rate (digital output 1) as well as alarms (digital output 2). Besides those two digital outputs the device is also equipped with an analog current output. It outputs either the measured flow rate or the most severe pending alarm in form of an analog electric current, which is generated by a PWM (pulse width modulation). The analog current output itself is not realized on the main board but instead on an output board equipped with an own microcontroller.

From a software viewpoint, the software fractions running on the three disctinct boards are indeed three software systems. We will concentrate on the software system running on the main board as the running example of this report. We will refer to it in the following as the *MeDUSA example (software) system* (MES).

1.6 Outline of report

Based on the central division into method content and process, this report is split into two major parts. In chapter 2, the method content is defined, structuring roles, tasks, work products, and guidelines into four disciplines. For each discipline, the definitions of all tasks belonging to the discipline are itemized, covering the role performing the task, work products being produced by the task, as well as guidelines that may support the execution of the task. Adjacent, chapter 3 defines the MeDUSA process in terms of four phases. Each phase consists of a number of iterations (being defined as capability patterns), whose definition is itemized one by another, specifying in terms of task descriptors, how the tasks of the method content are executed over time. Chapter 4 will then give a short summary and conclusion as well as an outlook on future work.

2 Method Content

The MeDUSA method content is defined - as specified by the Unified Method Architecture - in terms of roles, tasks, work products, and guidances.

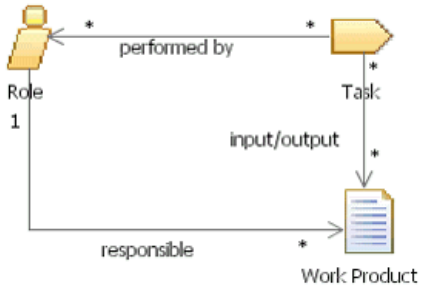


Fig. 4. UMA method content concepts (based on [Hau06])

As shown in Figure 4, tasks are performed by roles. They produce work products as outputs and may rely on other work products as inputs. While a task may be performed by multiple roles, of whom one may indeed be identified as the primary performer, a role might perform multiple tasks. The responsiblity for a single work product however lies within a single role.

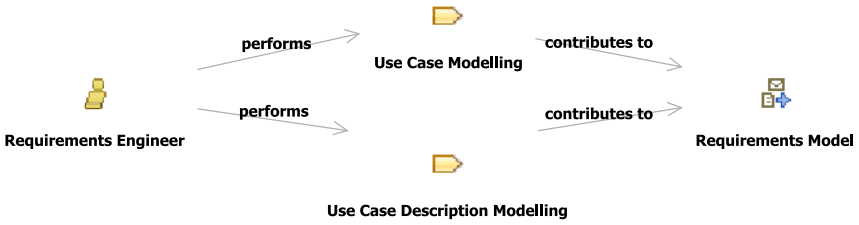
Guidances support the execution of tasks. According to the UMA terminology they can be devided into checklists, concepts, examples, guidelines, practices, reports, reusable assets, roadmaps, templates, term definitions, tool mentors, and whitepapers. Although we think that all other guidance types are important to a practical application of the method, we concentrate to provide only guidelines within this report. Further guidances should be included by a usable hypertext documentation of the method (see [MeD]).

The UMA terminology supports the categorization of roles, tasks, work products and guidances into so called content categories, to support the structuring of the method content definition. We make use of this concept by splitting the MeDUSA work products into four categories (called disciplines), namely *Requirements*, *Analysis*, *Architectural Design*, and *Detailed Design*.

Each discipline groups strongly related tasks (typically performed by one single role) and the work products being produced by these tasks. As MeDUSA method is characterized to be based on the UML notation, most work products are specified to be UML diagrams. As those UML diagrams have to be consistent with each other, which can only be achieved if they are based on an underlying UML model, the collaborative work product produced by the tasks of each discipline can therefore be understood as a model. Hence, MeDUSA defined a *Requirements Model*, an *Analysis Model*, *an Architectural Design Model*, and a *Detailed Design Model* respectively.

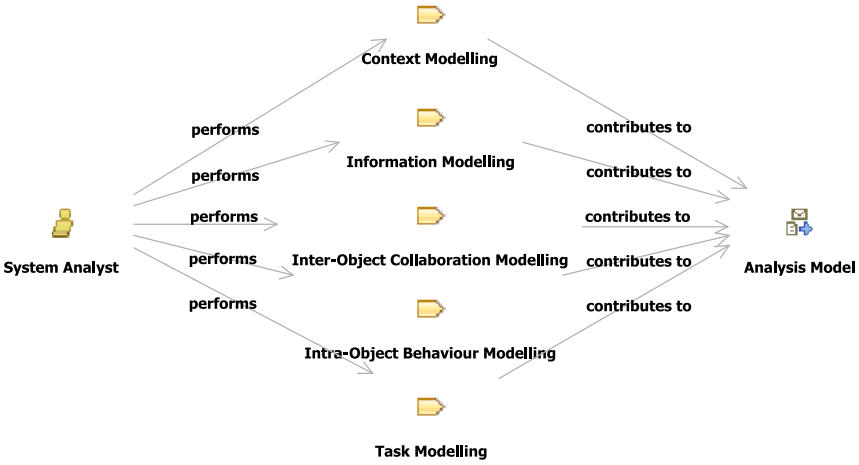
Before we give detailled information on the defined disciplines in the following chapters we briefly sketch their purpose and sturcture.

Requirements discipline



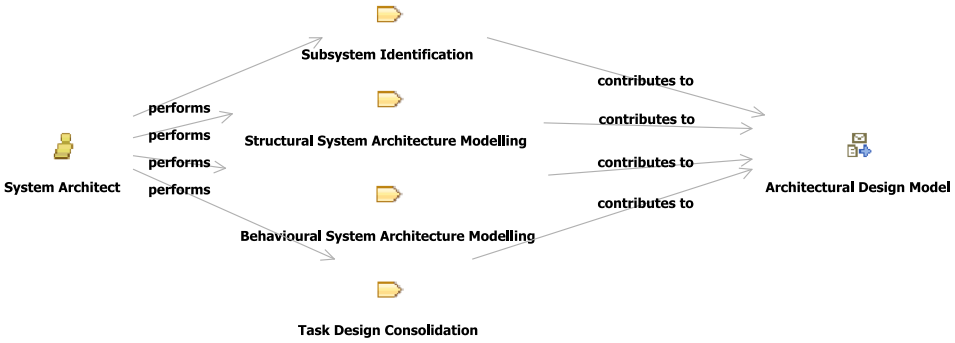
The *Requirements* discipline is concerned with understanding and capturing the functional requirements of the system under development, as well as the non-functional timing and concurrency concerns that constraint them. As MeDUSA is a use case-driven method, the *Requirements Model* is established in form of a UML use case model as well as a narrative model, specifying detailed narrative descriptions for each use case. Both tasks of the *Requirements* discipline are performed by the *Requirements Engineer*, which is also the single responsible for all work products produced.

Analysis discipline



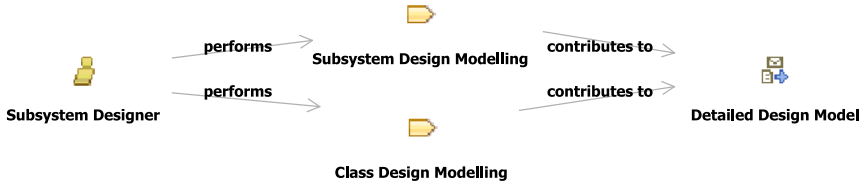
The *Analysis* discipline deals with understanding the problem domain. That is, the problem domain is modelled in terms of analysis objects who collaboratively perform the use cases captured in the *Requirements Model*. Thereby a detailed understanding of the problem domain is gained. Further, the concurrent tasks inherent to the system under development are identified (indeed they can be inferred from the requirements) and the overall schedulability of those concurrent tasks is evaluated. The tasks comprised by the *Analysis* discipline are performed by the *System Analyst*, who is also responsible for the produced work products.

Architectural Design discipline



The *Architectural Design* discipline is concerned with the specification of the system architecture. That is, based on the analysis objects captured in the *Analysis Model*, a system architecture is defined in terms of subsystems, which can be understood as groups of objects, with clearly defined interfaces. The system architecture is not only defined from a structural viewpoint, specifying how the subsystems are structurally interconnected via their interfaces, but also from a behavioural viewpoint, specifying the inter-subsystem communication. Further the design for all data types that are needed in the signatures of the subsystems' interfaces as well as the overall task design have to be specified. All tasks in the *Architectural Design* discipline are performed by the *System Architect*, who is also responsible for all work products being produced.

Detailed Design discipline



The *Detailed Design* discipline groups those tasks, which are related to the design of individual subsystems. As the externally visible interfaces of all subsystems are defined already as part of the system architecture, the tasks belonging to the *Detailed Design* discipline are concerned with designing the internal decomposition of each subsystem. Further a detailed class design has to be developed here, as the produced *Detailed Design Model* is the direct input for the implementation tasks. All *Detailed Design* tasks are performed individually for each subsystem by the respective *Subsystem Designer*. He is responsible of work products related to the subsystem he is responsible for.

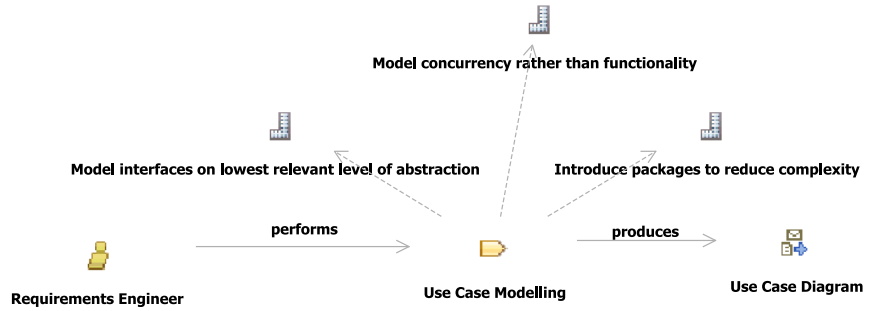
2.1 Requirements Discipline

The *Requirements Discipline* is concerned with eliciting and understanding the functional requirements of the software system under development by capturing them in a *Requirements Model*. It has to be pointed out that in the domain of real-time systems besides functional requirements also non-functional requirements (timing and concurrency constraints) play an outstandingly important role, as they may have severe impact on the later overall system design. This is why *Requirements Modelling* in the context of real-time systems has to also deal with capturing those non-functional constraints.

The *Requirements Model* is developed in terms of use cases and narrative use case descriptions. This is why the *Requirements Modelling* discipline is broken down into the following two tasks:

1. *Use Case Modelling*: Develop one or more use case diagrams to depict the essential use cases of the software system under development and to understand how the system interacts with its environment to fulfil those.
2. *Use Case Description Modelling*: Document each use case in a narrative use case description, to capture the detailled flow of events of each use case and to document pre- and post-conditions as well as other valuable information.

Use Cases Modelling



Use Case Modelling deals with the development of a use case model in terms of external actors, use cases, and their relationships. Use cases describe sequences of interaction between the software system under development and the external actors. They have the objective of accomplishing a certain goal, which is usually of value to one of the external actors. Actors trigger the execution of use cases inside the system (primary actor) and take part in the interaction with the system (secondary actor). The software system is treated as a black-box in this context, meaning that no assumptions about the internal structure of the software system are made. As use case modelling is a quite common technique of software engineering, we will skip to give a detailed introduction here. The reader may refer to [JCJv92], [Coc01], or [Wal07] to get a basic introduction into use case modelling.

As already anticipated, in the domain of real-time systems not only functional requirements have to be regarded, but non-functional timing and concurrency concerns are of major importance. They have severe impact on the design of the software system under development and therefore have to be investigated and understood as early as possible. That is why we want to address them very explicitly already during use case modelling.

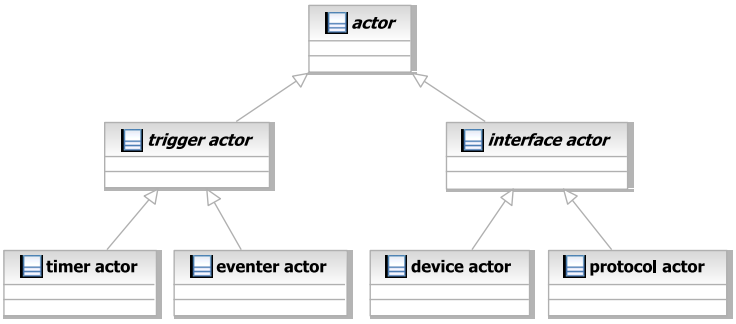


Fig. 5. MeDUSA Actor Taxonomy

To be able to capture those non-functional timing and concurrency concerns apart from functional requirements normally captured in a use case model, we propose to use *timer* and *eventer* actors. Those actors represent sources of either periodic (timer

actor) or aperiodic (eventer actor) events and occur as triggers for the execution of use cases. In fact, we cast timer and eventer actors to be the only primary actors (those triggering the execution of a use case) and consider all other actors to be secondary actors, so called interface actors, uncoupling any timing and concurrency concerns from them. The complete taxonomy of actors is defined as shown in Figure 5. According to this besides the classifiction of trigger actors into timer and eventer actors, which was motivated before, interface actors are also further devided into device actors (representing an external hardware device) and protocol actors (representing an external software system).

That is, if an external hardware device or software system does also trigger the execution of the use case, it should be represented by two actors, a device or protocol actor representing the communication interface and a timer or eventer actor representing the event source triggering the use case execution. If for example an external input device delivers data to the system in an aperidioc manner and notifies the system about the arrival of such data by using a hardware interrupt or any other mechanism, we propose to introduce two actors to the use case model; one eventer actor representing the event source (i.e. the hardware interrupt) and one device actor representing the interface used to obtain data from the device. Using such timer and eventer actors, concurrent execution of use cases can then be explicitly expressed by associating use cases to different timer or eventer actors, dependent on whether the use cases are performed in a periodic or aperiodic manner.

One may notice that in such a setting, it may occur, that the concurrent execution of a use case is indeed not triggered by a periodic or aperiodic event source from outside the system but from inside it. This is most likely the case if a use case is periodically executed from within the system and does not correspond directly to a periodic event source that resides outside the system. Although Jacobson and Overgaard ([JCJv92]) state that “the essential thing is that actors constitute anything that is external to the system we are to develop” we propose to model internal timer and eventer actors to capture such a situation.

To determine the use cases in a systematic manner, it might be reasonable to start with identifying the external actors. As stated above, they may represent external hardware devices and external software systems, as well as external timers or eventers. We tend to not represent human users as actors, as they communicate with the system not directly but always indirectly via hardware and software interfaces. As in the real-time domain those interfaces are normally no standard interfaces, the hardware or software interface is more of interest than the user communicating over it. After having identified the actors, the next step is to regard what behaviour the primary actors (those who trigger execution of use cases) initiate in the system. This leads to a first set of use cases directly associated to the primary actors. Analyzing those use cases in terms of similar interaction subsequences might lead to the identification of new use cases encapsulating that behaviour. New use cases and relationships between use cases are successively identified this way. Regarding timing and concurrency concerns of the identified use cases might lead to extracting further interactions into own use cases (if they are executed concurrently) and might also lead to the identification of further (internal) actors.

WORK PRODUCTS

– *Use Case Diagram* The results of the use case modelling task are captured in a use case diagram. An example is shown in Figure 6.

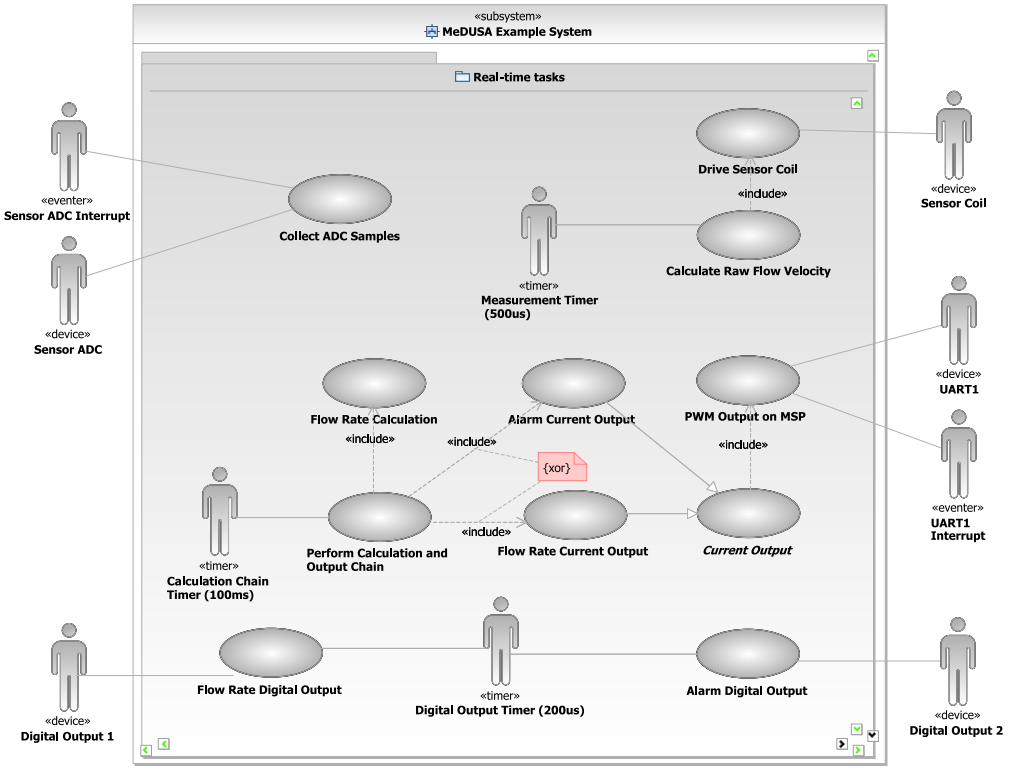


Fig. 6. Example: MES Use Case Diagram (excerpt)

It captures the functional and the non-functional timing constraints in terms of

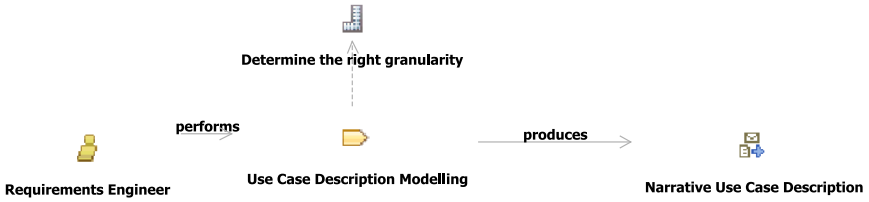
- the system boundary,
- the use cases (inside the system boundary)
- the internal and external timer and trigger actors
- external (hardware) device or (software) system actors,
- relationships between use cases (generalization, include, extend),
- relationships between actors (generalization), and
- relationships between use cases and actors (associations)

GUIDELINES

– *Introduce packages to reduce complexity*: If the use case granularity is satisfactory and still a large number of use cases occur, packages may be introduced to group use cases into functional areas. It might also be helpful in use case models having a smaller amount of use cases, to group use cases regarding certain aspects, e.g. if they are performed cyclically and can be regarded as belonging to the real-time related tasks of the system.

– *Model interfaces on lowest relevant level of abstraction* From our lessons learned, a major modelling problem one often has to deal with is that an actor has interfaces to the system on different layers of abstraction. This might for example be the case if communication to another software system is established via a hardware communication interface or via an underlying software interface (if for example the communication service is provided by an underlying operation system). In such a case, the software under development has interfaces to its surrounding environment on different levels of abstraction.
  Consider as a concrete example that the PWM (pulse width modulation) needed for the analog current output is realized on a separate output board, which is accessed from the software system under development via an asynchronous UART communication interface. The question that arises is whether the software protocol on the higher level of abstraction (which we will refer to as PWM protocol), the underlying UART device interface (or respective operating system communication protocol), or both should be reflected in the use case model.
  The UML does not provide sufficient support to model interfaces on different levels of abstraction with the help of actors. Our advice to this modelling problem is that one should indeed represent the interface(s) on the lowest relevant level of abstraction. That is, if the software system under development is responsible of controlling the UART device interface itself, it should be represented as a device actor (and a corresponding eventer actor representing the interrupt source inherent to the UART). If the functionality of communicating via UART is realized by an underlying operation system, a protocol actor should indeed be introduced to represent the operation system communication facility.
  The PWM output protocol on the higher abstraction level might also be represented (by an additional protocol actor, being linked to the underlying interface actors via a dependency), as it is a relevant interface to the software system under development, but it may not be the only interface represented. If indeed, the UART device actor (and UART interrupt eventer actor) or the operating system communication protocol actor would be omitted, the underlying direct communication interfaces the software system under development has to interface to, would not be represented. Indeed, in case of the software system having to control the UART device directly, the concurrency needed to react to the UART interrupt would also not be reflected in such a case, which would have a significant impact on the later task design.
– *Model concurrency rather than functionality*: As timing and concurrency concerns are modelled apart from the functional requirements using timer and eventer actors, a question might arise on whether two use cases which are functionally related but executed concurrently are modelled independently, being associated to different internal timer or eventer actors or related to each other using include or extend relationships. We propose to give precedence to the concurrency concerns in such a case, as timing and concurrency concerns are of outstanding importance for real-time systems and are thus more heavyweight compared to the functional dependencies that might be identified.

Use Case Description Modelling



Based on the use case modell a narrative description of each use case modelled has to be developed. This way details about the interaction sequences (main interaction sequence, alternative sequences) as well as other valuable information like pre- or post-conditions, which cannot be captured graphically, can be recorded.

To capture the narrative description of use cases, we recommend a notation developed at the Research Group Software Construction [Wal07]. It was designed to capture narrative use case descriptions in a concise and understandable form that remains in line with the semantics of use cases as defined by the UML. The notation was designed inspired by the notation presented in [BS02] around the concept of “flow of events”. Each use case is understood as one or more flow of events, where an event represents an atomic part of a system-actor interaction. Besides the main flow of events, a concrete use case should always have (either directly or indirectly by inheriting it from a general use case), a use case may also have alternative flows of events to capture exceptual behaviour or error handling. Inclusion and extension of other use cases is also expressed in terms of depencendies between their respective flows of events. Even generalizations between use cases is transferred into the domain of flows, as spezialization is understood in terms of specializations between flows. Taking the flow as the central concept around which textual use case descriptions are defined, a consistent and understandable notation of use cases can be created, which is very much in line with the common understanding of use cases as interaction sequences. Another advantage of the rather semi-formal notation is, that consistency with the UML use case model (like developed in the *Use Case Modelling* task) can be easily validated. To gain further understanding and a detailed introduction into the developed notation, we propose to refer to [Wal07].

WORK PRODUCTS

– *Use Case Description:* A detailed narrative description for each use case should be developed, as exemplarily shown in Figure 7 for some of the use cases graphically modelled in Figure 6. It uses the notation presented in [Wal07].

GUIDELINES

– *Determine the right granularity:* Finding the right granularity is often difficult when identifying and modelling use cases. If use cases are modelled too fine-grained, a lot of trivial use cases are modelled. In such a situation, a lot of associations, include, exclude, and generalization relationships are also modelled in consequence, so that the overall use case model gets rather complex. If use cases

Use Case Current Output

Main Flow	
Start	
1	Alternative Extension Point : Choose Between simulation and calculation
2	Specialization Extension Point : Calculate actual current
3	Alternative Extension Point : Current stored
4	Validate that current does not exceed span limits.
5	Alternative Extension Point : Current validated
6	Calculate PWM output signal.
7	Normalize.
8	Include Use Case Output PWM signal on MSP.
9	Alternative Extension Point: End
End	

Alternative Flow Simulate Current	
Start	At Choose Between simulation and calculation, if simulation mode has been set
1	Store simulated current value as current value.
End	Continue at Current stored

Alternative Flow Raise ”Limits exceeded” alarm.	
Start	At Current validated, if the current exceeds span limits
1	Raise ”Limits exceeded” alarm.
End	Continue at End

Use Case Alarm Current Output

Specialization Flow Calculate alarm current	
Start	At Calculate actual current
1	Calculate actual current from alarm value.

Use Case Flow Rate Current Output

Specialization Flow Calculate flow rate current	
Start	At Calculate actual current
1	Calculate actual current from flow rate value.

Use Case Perform Calculation And Output Chain

Main Flow	
1	Include Use Case Flow Rate Calculation
2	Include Use Case Flow Rate Totalizing
4	Validate that no alarm has been raised
6	Alternative Extension Point : Alarm state validated
6	Include Use Case Flow Rate Output
7	Alternative Extension Point: Analog output done
8	Include Use Case Flow Rate Digital Output

Alternative Flow Output Alarm.	
Start	At Alarm state validated, if any kind of alarm has been raised
1	Include Use Case Alarm Output.
End	Continue at Analog output done

Fig. 7. Examples of Use Case Descriptions

are modelled too course-grained, they tend to be internally complex (lots of instructions and alternative branches) what makes their narrative descriptions difficult to handle.

We propose to consult the narrative use cases descriptions as a guidance for determining the right granularity of use cases, as the internal flow of events captured in a narrative use case description does support the appraisal of a use case's complexity far more than what can be inferred from the use case diagram. We noticed that beginners tend to often model too fine-grained. Often use cases that represent just single steps are modelled. Sequences of such "single instruction" use cases are then combined together by including them by another use case, which represents no own functionality but mere control logic.
Good guidance to determine the right granularity of a use case can be taken from the narrative description developed for it. If the narrative description of a use case does consist of only one or two steps this might indicate that the use case is modelled to fine-grained. If the description gets rather complex (lots of steps and branches) this is a good indicator that the use case model is indeed to course-grained. For the application domain regarded, a rule of thumb might be that a good granularity is achieved if a narrative use case description consists of about 5 to 15 steps.

2.2 Analysis Discipline

The *Analysis Discipline* is concerned with the development of an *Analysis Model* that helps to understand the problem domain in terms of objects, whose collaborative behaviour performs the use cases of the software system. Construction of the *Analysis Model* can therefore conceptually be broken down into three main objectives:

- Identifying all objects needed to perform the use cases.
- Capturing the inter-object behaviour of the identified objects.
- Capturing the intra-object behaviour of the identified objects.

Identifying objects is of course a quite complicated task that has to be broken down into handable units to get manageable. MeDUSA addresses the identification of objects successively during three activities of the *Analysis Modelling* discipline by regarding objects of different categories during each task.

The categories used to support the identification process are defined by the MeDUSA object taxonomy. It was designed following the analysis object taxonomy of the COMET [Gom00] and is shown in Figure 8.

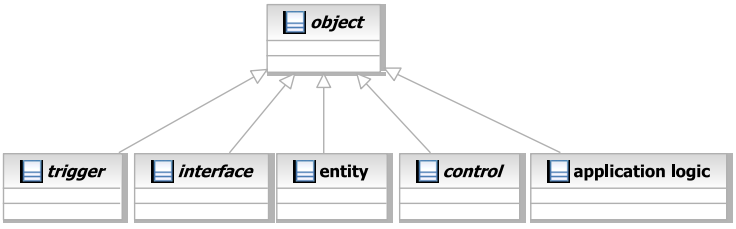


Fig. 8. MeDUSA Object Taxonomy

According to it, analysis objects can be classified into *trigger*, *interface*, *entity*, *control*, and *application-logic* objects.

- *trigger objects* represent periodic or aperiodic sources of events external to the software system under development.
- *interface objects* represent hardware or software interfaces towards the external environment of the software system under development.
- *entity objects* represent long-living data the software system under development has to keep track of.
- *control objects* represent control-flow logic needed to coordinate between other objects or to encapsulate state-dependent behaviour.
- *application-logic objects* represent self-encapsulated pieces of application-logic like an algorithm or an application-domain specific functionality (which is neither control-flow and is therefore not encapsulated into a control object, nor functionality related to the long-living data of a single entity object and is therefore not encapsulated into the respective entity object).

The different tasks of the Analysis discipline aim at identifying objects of different categories each.

- *Context Modelling* supports the identification of interface and trigger objects by questioning what interfaces from the software system under development towards its external environment have to exist and what external event sources the system under development has to correspond to. Interface objects may represent interfaces to external hardware devices, or software protocols. Trigger objects do represent external sources of periodic and aperiodic behaviour (which might be adherent to the interfaces).
- *Information Modelling* helps to identify which entity objects are needed to store data, which has to be handled by the system.
- *Inter-Object Collaboration Modelling* takes into consideration the use cases identified during *Requirements Modelling*. It supports the identification of objects that make up application-logic or control-flow-logic by questioning, which additional objects are needed to perform each use case. It combines the identification of the application-logic and control objects with the capturing of inter-object behaviour that results from performing each use case.
- *Intra-Object Behaviour Modelling* deals with specifying the intra-object behaviour by synthesizing the partial behaviour each identified object shows in the collaborations it takes part in.

After all relevant analysis objects have been identified and the structural and behavioural relationships between them have been specified, the concurrent behaviour sequences triggered by the active trigger objects - we refer to as task candidates - have to be analyzed, regarding whether they are capable to meet their individual timing constraints. A first impression on the overall system schedulability has also to be gained. Both is done during *Task Modelling*.

Context Modelling - Identify Interface and Trigger Objects

During *Context Modelling* all hardware and software interfaces of the software system under development towards its surrounding environment as well as all sources of periodic and aperiodic events, the system has to deal with, are regarded. A *System Context Diagram* is developed that captures the gathered information in terms of a UML object diagram that shows the software system under development as well as *interface* and *trigger* objects aggregated by it.

– *Trigger objects* are modelled if the software system under development needs to keep track of time or has to react to aperiodic events. Trigger objects are categorized as shown in Figure 9 dependent on whether they represent a periodic or aperiodic event source. While periodic event sources are represented by *timer objects*, aperiodic event sources are represented by *eventer objects*. That is trigger objects are directy inferred from the trigger actors captured in the requirements model.

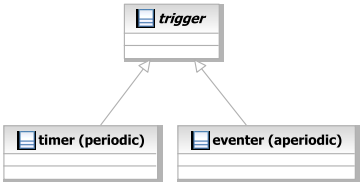


Fig. 9. MeDUSA Trigger Object Taxonomy

– *Interface objects* serve as interaction points for incoming or outgoing communication of the system under development towards its external environment. As shown in Figure 10, interface objects are categorized into hardware and software interfaces.

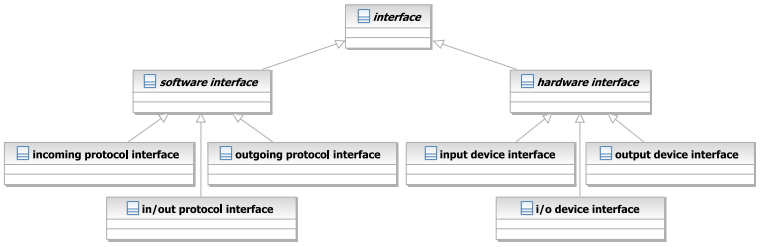


Fig. 10. MeDUSA Interface Object Taxonomy

Identifying interface and trigger objects is done by inferring them from the actors identified in the *Requirements* discipline. While trigger actors are directly mapped to trigger objects, interface actors will of course lead to corresponding interface objects. It may however be the case that an interface actor leads to multiple interface objects if they are categorized under different aspects.

WORK PRODUCTS

– *Context Diagram:* The results of context modelling are captured in a *Context Diagram*, which is developed in form of a UML object diagram as depicted in Figure 11.

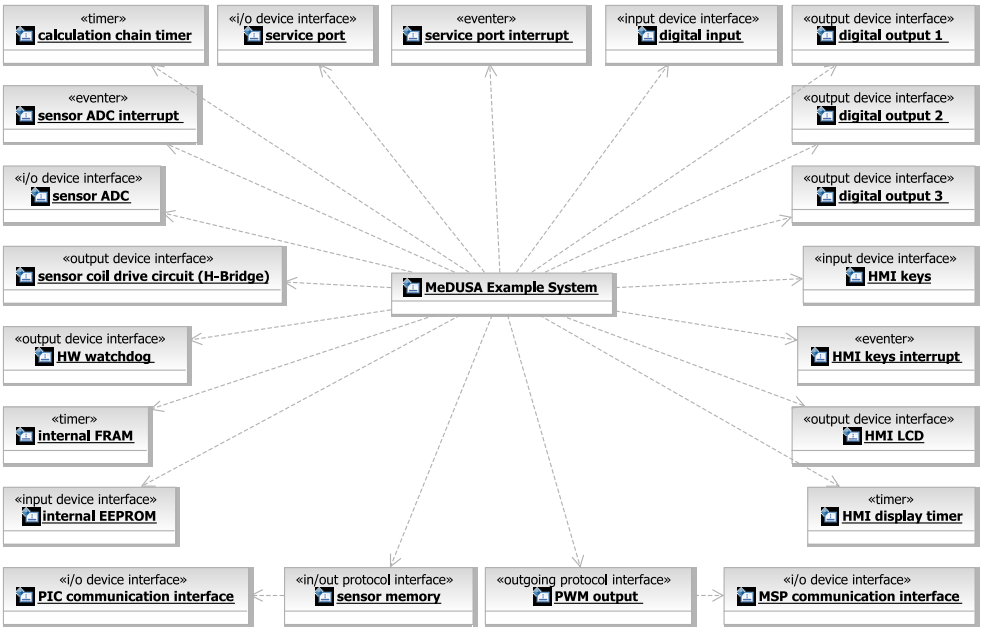
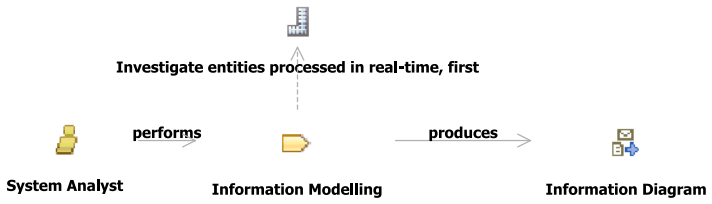


Fig. 11. Example: MES System Context Diagram

It shows the software system under development as an aggregate object that composes interface objects, which are used to interact with the external environment of the software system, as well as trigger objects representing sources of periodic or aperiodic events.
Depending on the characteristics of the identified objects they are categorized (by using stereotypes) into one of the following categories as specified by the MeDUSA Interface Object Taxonomy shown in Figure 10. Trigger objects are stereotyped accordingly as specified by the MeDUSA Trigger Object Taxonomy shown in Figure 9. The composition of the interface objects by the system aggregate object is modelled using links (instances of associations, respectively aggregations) or dependencies.

Information Modelling - Identify Entity Objects



Information Modelling is done to capture the data-intensive objects of the problem domain - the so called entity objects - as well as relationships between them. Entity objects store data that is long lasting and often accessed by several use cases. Entity objects may represent measured physical quantities, real world objects, abstract concepts or any other data as constraints, configuration, or calibration information. Entitiy objects may contain slots (instances of attributes) representing the properties of the entity. Those properties might also be physical quantities, as e.g. the `bore` has an `innerWidth` and a `nominalWidth` property.

WORK PRODUCTS

– *Information Diagram:* The results of the information modelling task should be captured in an *Information Diagram*, which is developed in form of a UML object diagram as shown exemplarily by Figure 12.

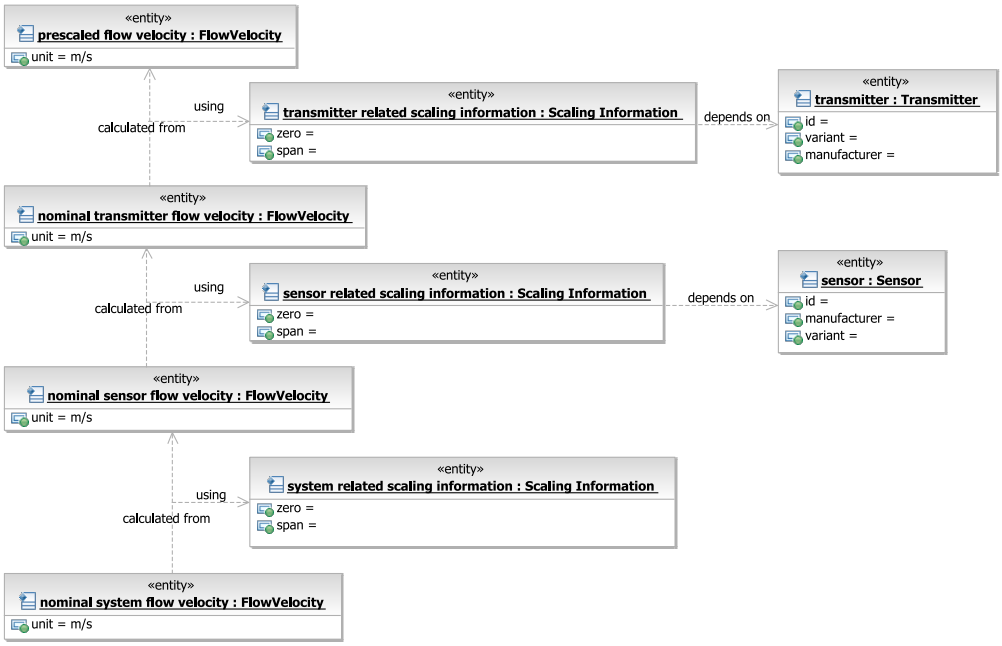


Fig. 12. Example: Information Diagram (excerpt)

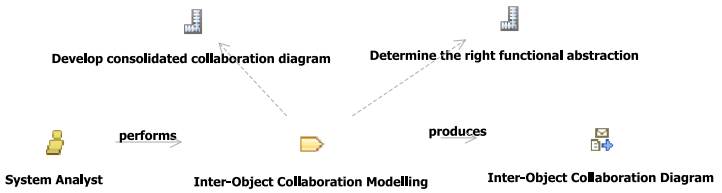
It shows the entity objects linked to each other using links or just dependencies (less formal). If supported by the tool, modelling n-ary links or dependencies re-

lating on other dependencies may be useful in some cases, e.g. when an value entity is calculated from another using the information stored in a third data entity.

GUIDELINES

- *Investigate entities processed in real-time, first*: As the domain od MeDUSA is much focused on value processing, identifying the relevant entity objects is most easily done by first identifying the relevant physical quantities involved into the real-time tasks of the device, e.g. the `flowVelocity` or `volumeFlow`. As those entities are often intertwined (they most often get calculated from each other), other entities may be identified next, as they are needed for the translation/calculation steps. For example, the density of the medium flowing through the device is needed to calculate the mass flow, leading to an entity object called medium having a property/slot of name density.

Inter-Object Collaboration Modelling - Identify Control and Application-Logic Objects



In this step for each use case identified during requirements modelling a collaboration of objects is identified, whose collaborating behaviour fulfils the goal of the use case.

As a starting point to identify the objects performing a use case collaboratively, the object initiating the execution of the use case behaviour has to be identified. Indeed, unless the use case is included by another use case or extends another use case, this always has to be one of the trigger objects identified during the *Context Modelling* task, as these are the objects that are directly inferred from the actors identified during *Requirements Modelling* (they together with the interface objects also deferred from the actors are indeed the only objects that manifest interaction with the external environment of the software system under development). In case the use case is included by another use case or the use case extends another use case, the object triggering the use case will be one belonging to the collaboration performing the including respectively extended use case (most likely it will be a control object).

Next the interface and entity objects involved in the use case, which were identified during System Context Modelling and System Information Modelling have to be identified. Having found them, the main flow of events of the use case has to be investigated and additional control and application-logic objects have to be identified, which are needed in addition to perform the use case:

– *Application logic objects* encapsulate functionality relevant to the regarded application domain. This may for example be an algorithm or some business-logic that accesses more than one entity object or is likely to be changed and is therefore encapsulated into an own object.
– *Control objects* are meant to encapsulate control flow logic. As prescribed by the *MeDUSA Control Object Taxonomy* shown in Figure 13, control objects can be further classified into *coordinator* and *state-dependent-control* objects. While state-

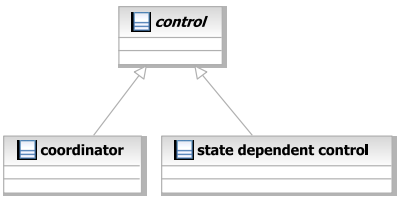


Fig. 13. MeDUSA Control Object Taxonomy

dependent-control objects encapsulate state-dependent behaviour, coordinator objects encapsulate some non-state-dependent coordination between other objects.

Having identified the necessary control and application-logic objects, the use case main flow of events can be described in terms of messages between the identified objects to gain an understanding on how the collaborative interplay of the identified objects performs the use case main flow of events. Last, alternative flow of events have to be considered. This may lead to identification of additional control and application-logic objects, it may also just lead to additional messages being sent between already identified objects.

After *Inter-Object Collaboration Modelling* has been performed, all necessary objects should be identified and it should be understood how these objects collaboratively work together to perform each use case identified during Requirements Modelling.

WORK PRODUCTS

– *Inter-Object Collaboration Diagram*: The results of *Inter-Object Collaboration Modelling* should be captured in one or more *Inter-Object Collaboration Diagram(s)* for the use cases identified during *Requirements Modelling*. Those *Inter-Object Collaboration Diagams*, are developed in form of UML2 communication or sequence diagrams like shown exemplarily by Figure 14 and 15. The decision whether to use a communication diagram or a sequence diagram to depict the collaborative behaviour depends on whether the emphasis is placed more on showing the objects and their structural relationships (communication diagram) or on the flow of messages (sequence diagram).

  We propose to model at least the main flow of the use case in a communication diagram that shows all objects participating in the collaboration (also those not involved in the main sequence) to show the identified objects and their structural relationships. All alternative flows of the use cases should in our eyes be modelled by an additional sequence diagram, as it better supports the modelling of optional or alternative messages by the use of fractions. It may however be reasonable to just have a single communication diagram (if there are no alternative flows or if they are trivial) or just a single sequence diagram, if the number of objects is quite low.

  Even if stated so above, it may not always be necessary or reasonable to have *Inter-Object Collaboration Diagrams* for each individual use case identified during *Requirements Modelling*. If for example a use case is included by another use case, it might be reasonable to integrate the collaboration for the included use case into the *Inter-Object Collaboration Diagram* of the including one. It may however - even in such a case - be reasonable to have separate diagrams for both use cases (if for example the included use case is also included by anther use case or if the number of objects or messages grows to large if combining the two). The same holds for a use case extending another use case. Also in this case it might be reasonable to handle the extending use case within the *Inter-Object Collaboration Diagram* for the extended one.

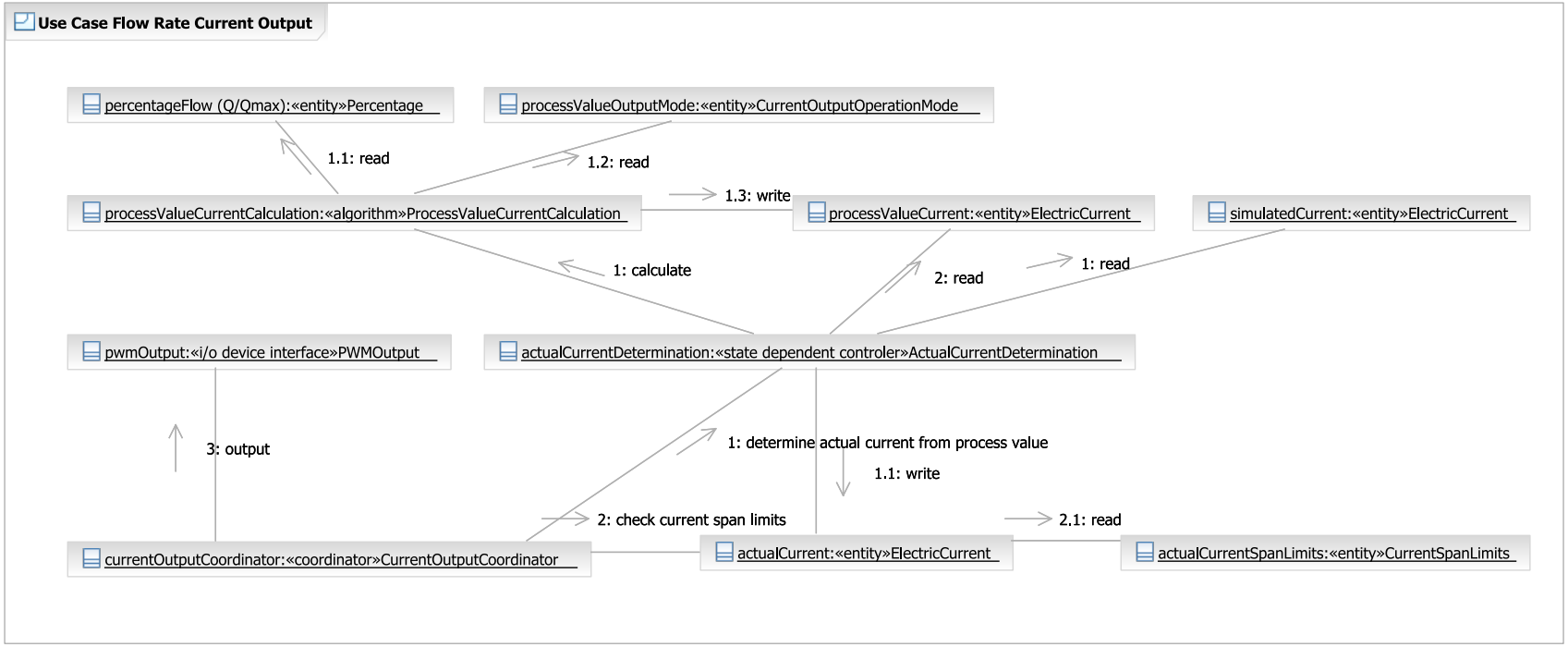


Fig. 14. Example: Inter-Object Collaboration Diagram (Communication)

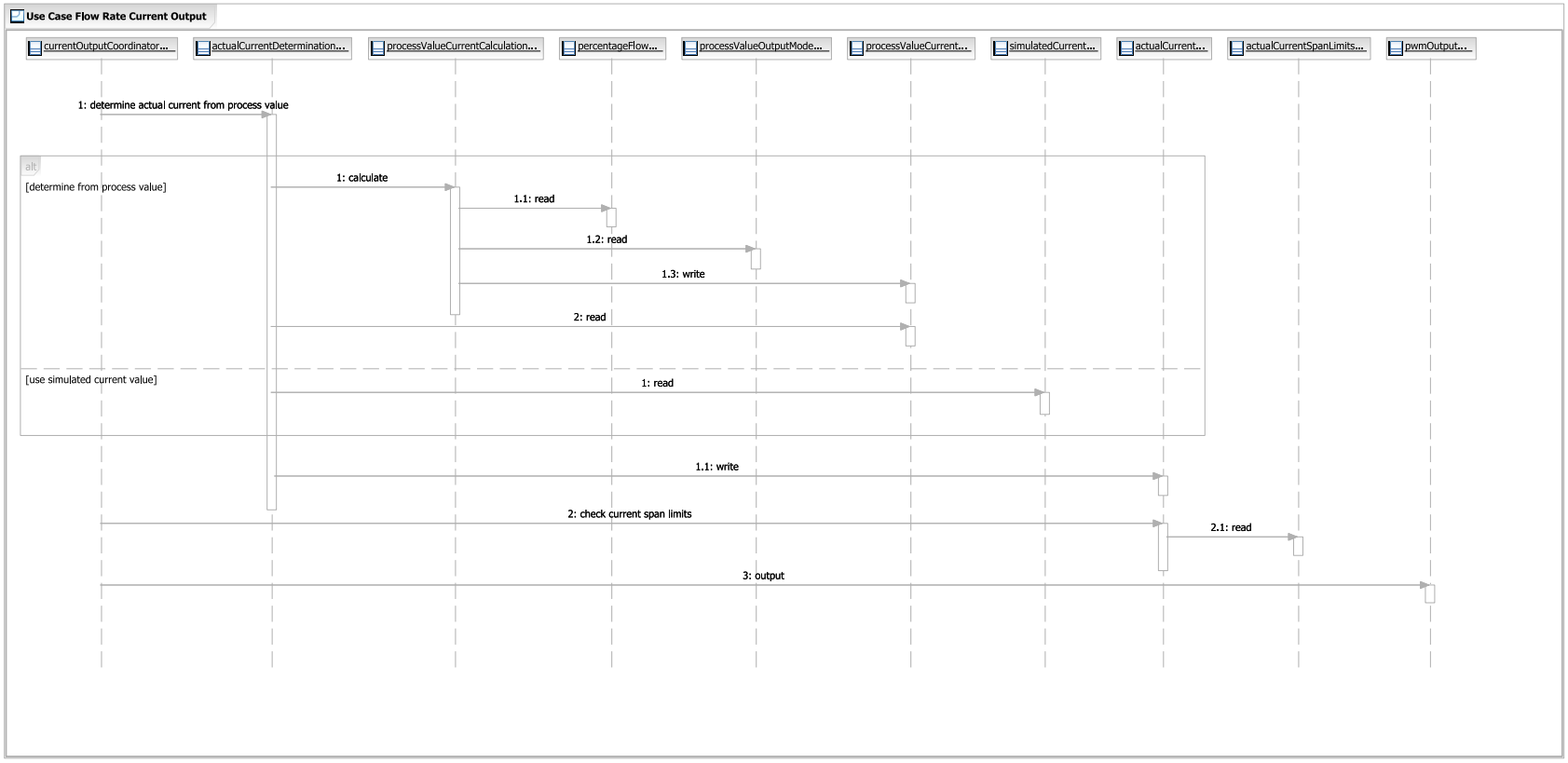
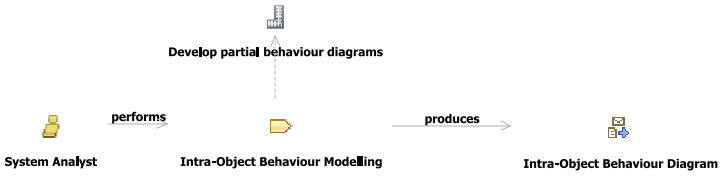


Fig. 15. Example: Inter-Object Collaboration Diagram (Sequence)

GUIDELINES

- *Develop consolidated collaboration diagram*: As it might be rather hard to retrieve information about functional coupling of the objects from the communication and sequence diagrams developed during *Collaboration Modelling* (as an object often participates in more than one collaboration and also a single collaboration is often modelled in several diagrams to show all alternative flows), it may be reasonable to develop a consolidated communication diagram to support the succeeding activities. This is basically done by merging all communication and sequence diagrams of the identified use case collaborations together. More information can be found in [Go00] in chapter 12.4 (Consolidated Collaboration Diagrams).
- *Determine the right functional abstraction*: One question that often arises when identifying application-logic objects is whether a business-specific function or control-logic is best modelled by an application-logic object and when it is just a function of entity object (i.e. it is modelled as simple message). According to [Gom00] the question can be best answered by looking at how many entity objects would have to be accessed by the control or application-logic objects to execute. If more than one entity object is envolved, encapsulating the function or algorithm by an application-logic object is the better choice. If just one entity object is envolved it might usually be better to use a simple function in the respective entity object.

Intra-Object Behaviour Modelling - Model internal object behaviour



After having identified all needed application objects (trigger, interface, entity, control and application-logic), and having modelled how these objects collaboratively perform the identified use cases, the internal behaviour of all objects has to be modelled, where it is not trivial.

For all state-dependent control objects, which were identified during modelling of the system collaborations, the state-dependent behaviour has to be documented by a state machine diagram. If the state-dependent control object takes part in more than one of the collaborations, the state-dependent behaviour of that object has to be synthesized from the partial use case based behaviour of the object in all collaborations it participates in.

It may also be useful to capture the behaviour of coordinator objects, if the behaviour can not be extracted easily from the communication/sequence diagrams of the different collaborations it participates in. We propose to model a sequence diagram showing the overall behaviour of such a coordinator object in such a case.

Similar to specifying the behaviour of the identified control objects, it may be reasonable to also describe the behaviour of the identified application-logic objects, if the algorithm or business-logic encapsulated is rather complex. We propose to use an activity or state machine diagram for such a case.

WORK PRODUCTS

– *Intra-Object Behaviour Diagram*: As shown exemplarily in Figure 16, the internal behaviour of each non-trivial object should be captured by an *Intra-Object Behaviour Diagram*.

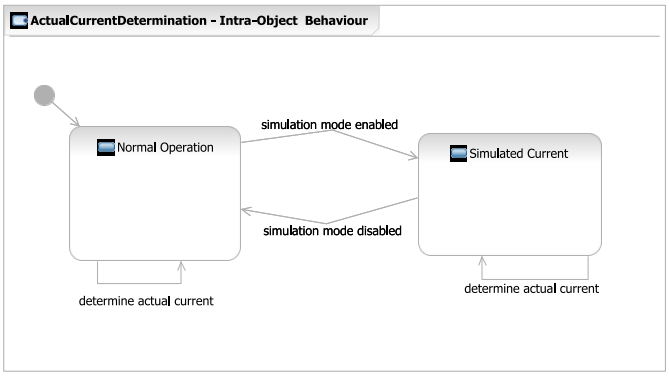


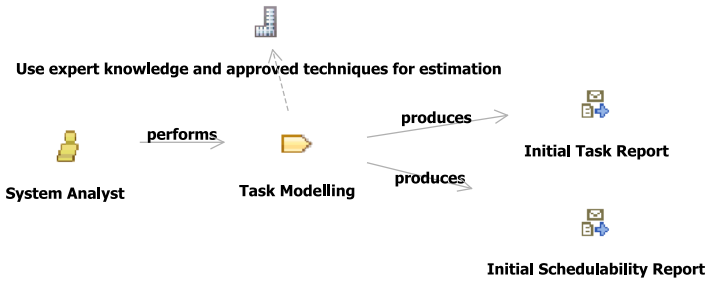
Fig. 16. Example: Intra-Object Behaviour Diagram - ActualCurrentDetermination

In case of a state-dependent application object this is done in the form of a state-machine diagram. For coordinator objects, sequence diagrams may be most appropriate. The internal behaviour of application-logic objects will most likely be best captured by using an activity diagram. However, other diagrams may be employed if applicable.
Further working products are the activity and/or state machine diagrams showing the behaviour of coordinator and application logic objects with non trivial behaviour.

GUIDELINES

– *Develop partial behaviour diagrams*: If the synthesizing of the partial state dependent behaviours of a state-dependent control object gets too complex to be managed, the process of synthesizing may be supported - if necessary - by developing a separate state machine diagram for the partial state dependent behaviours of the object in all collaborations first, and using them as input for synthesizing.

Task Modelling



As timing and concurrency concerns are of outstanding importance for real-time systems, identification of performance problems has to be done as soon as possible. Having identified the sources of concurrent behaviour during early *Use Case Modelling* by using trigger actors and having identified during *Inter-Object Collaboration Modelling* how each such concurrent behaviour - referred to as a task candidate[1], that is after the system architecture has been defined - is established in terms of messages between objects participating in collaborations identified for each use case, an early estimation can be done, on whether each such task is able to hold its deadline, and on whether the overall system is at all schedulable. Even if the overall task design is not established yet, valuable information can be inferred from such an early performance analysis, as potential problems can be inferred about individual task candidates likely to miss their deadline as well as on the overall system performance. Further, valuable information can be inferred to may be taken into account in the *Subsystem Identification* task, as one major criteria for partitioning of objects into subsystems is the task allocation.

Task analysis is started by identifying the period of each concurrent task candidate. For task candidates originating from timer objects, the period can be inferred directly from the timer period. For eventer objects a worst case assumption has to be made about the interarrival time of two asynchronous events. Having identified the task candidate's period, the CPU utilization of each task candidate has to be estimated. This can be done by estimating the time for message processing consumed by each object participating in the object-collaboration performing the respective task candidate, adding an additional overhead for the message communication itself.

Having gained an estimation for the CPU utilization and the period of each task candidate, it can be inferred if each task candidate is able to hold its individual deadline and if the overall system would be schedulable by applying real-time scheduling theory or event sequence analysis. We propose to refer to [Gom00] to get additional information of practices applicable.

[1] we refer to them as task candidates and not tasks, as the final task design cannot be defined earlier than during *Task Design Consolidation*

WORK PRODUCTS

– *Intitial Task Report*: The results of the task modelling should be captured in a *Initial Task Report*. It should list for each task candidate the following information:
  - the periodic and aperiodic event source from which the task candidate originates
  - a description of the task canidate's purpose
  - the task candidate's frequency (timer period or worst case interarrival time in case of eventer)
  - an estimation of the task candidate's CPU consumption time
  - the CPU utilization, computed from the task candidate's frequency and the CPU consumption
  - a target priority the task candidate should be assigned

  An example of an *Initial Task Report* can be seen in Figure 17.

Task Report

#	Trigger Object	Description	Frequency (T_i)	CPU consumption (C_i)	Utilization (U_i)	Priority (P_i)
t_1	sensorADCInterrupt	Collect and preprocess ADC samples from sensor	25 μs	5 μs	0.2	HIGH (1)
t_2	measurementTimer	Calculate Raw Flow Velocity from ADC samples	500μs	70 μs	0.14	HIGH (2)
t_3	calculationChainTimer	Calculate Flow Velocity, Volume and Mass Flow from Raw Flow Velocity and output them by PWM	100 ms	14.5ms	0.145	MED (4)
t_4	digitalOutputTimer	Output Process Value on Digital Output	200μs	3μs	0.015	HIGH (3)
…	…	…	…	…	…	…

Fig. 17. Example: MES Initial Task Report (excerpt)

– *Initial Schedulability Report*: The results of the schedulability analysis that has to be performed based on the estimations of the task candidate's frequency and CPU consumption should be captured in a *Initial Schedulability Report*. The form of the report depends on the concrete type of selected schedulability analysis, which is not prescribed by MeDUSA. We propose to refer to [Gom00] to get an overview of available practically approved techniques. However, independent on the applied technique, the *Initial Schedulability Report* should provide an estimation about the individual task candidates and the overall system schedulability. We will elaborate this on an example based on the **Generalized Utilization Bound Theorem** as introduced in [Gom00]. A detailed introduction into the applied principles of **real-time scheduling theory** and **event sequence analysis** can be found in chapter 17 of [Gom00] and will be omitted here due to lack of space.
  Let us assume that an example system consists of only the four tasks candidates listed in Figure 17. The overall CPU utilization can be computed as the sum of

the individual CPU utilizations to $0.2 + 0.14 + 0.145 + 0.015 = 0.5$, which is well below the worst-case utilization bound of 0.69, which is the upper utilization bound for a unrestricted number of tasks (compare [Gom00]). The priorities assigned to the task candidates were not based on rate monotonic scheduling (i.e. the task priorities were not assigned inversively to the task periods), as the `measurementTimer` task candidate was decided to get a higher priority than the `digitalOutputTimer` task candidate, although if it has the longer period. Therefore, each task candidate has to be analysed individually.

Schedulability Report

- **Task** t_1 is an aperiodic, interrupt-driven task with a worst case interarrival time of $T_1 = 25\mu s$ and a CPU consumption time of $C_1 = 5\mu s$. It has the highest priority.
  1. **Preemption time by higher priority tasks with periods less than** t_1. There are no tasks with periods less than t_1.
  2. **Execution time** C_1 **for task** t_1. Execution time is $5\mu s$ what leads to a utilization of $5\mu s/25\mu s = 0.2$.
  3. **Preemption by higher priority tasks with longer periods**. No tasks fall into this category.
  4. **Blocking time by lower priority tasks**. Task t_2 may block task t_1 because it accesses the ADC samples collected by task t_1. We assume that the blocking time (needed to read out the ADC samples) can be estimated to $4\mu s$, which leads to a blocking utilization during period T_1 of $4\mu s/T_1 = 4\mu s/25\mu s = 0.16$.

  The worst case utilization of task t_1 can thereby be computed as execution utilization + blocking utilization = $0.2 + 0.16 = 0.36$, which is well below the utilization bound of 0.69, so task t_1 will meet its deadline.

- **Task** t_2 is a periodic task with a period of $T_2 = 500\mu s$ and a CPU consumption time of $C_2 = 70\mu s$. It has the second highest priority.
  1. **Preemption time by higher priority tasks with periods less than** t_1. Task t_2 could be preempted by task t_1, which has a shorter period but a higher priority. The preemption utilization of task t_2 is 0.2
  2. **Execution time** C_2 **for task** t_2. Task t_2 has an execution time of $70\mu s$, which leads to a CPU utilization of 0.14.
  3. **Preemption by higher priority tasks with longer periods**. No tasks fall into this category.
  4. **Blocking time by lower priority tasks**. Task t_3 may block task t_2 because it accesses the raw flow velocity calculated by task t_2. We assume that the blocking time (needed to access the flow velocity) can be estimated as $3\mu s$, which leads to a blocking utilization during period T_2 of $3\mu s/T_2 = 3\mu s/500\mu s = 0.006$.

  The worst case utilization of task t_2 can thereby be computed as $0.2 + 0.14 + 0.006 = 0.346$ which is below the utilization bound of 0.69, so task t_2 will also meet its deadline.

- **Task** t_3 is a periodic task with a period of $T_3 = 100ms$ and a CPU consumption time of $C_3 = 14.5ms$. It has the lowest priority of the four regarded tasks.
  1. **Preemption time by higher priority tasks with periods less than** t_3. Task t_3 could be preempted by tasks t_1, $t2$ and $t4$, which all have a shorter period and a higher priority. The summarized preemption utilization of these tasks is 0.355
  2. **Execution time** C_3 **for task** t_3. Task t_3 has an execution time of $14.5\mu s$, which leads to a CPU utilization of 0.145.
  3. **Preemption by higher priority tasks with longer periods**. No tasks fall into this category.
  4. **Blocking time by lower priority tasks**. Task t_3 has the lowest priority of the regarded tasks, so no tasks fall in this category.

  The worst case utilization of task t_3 can be computed as $0.355 + 0.145 = 0.5$, which is below the utilization bound of 0.69, so task t_3 will also meet its deadline.

- **Task** t_4 is a periodic task with a period of $T_4 = 200\mu s$ and a CPU consumption time of $C_4 = 3\mu s$. It has the third highest priority of the regarded tasks.
  1. **Preemption time by higher priority tasks with periods less than** t_4. Task t_3 could be preempted by task t_1, which has a shorter period and a higher priority. The preemption utilization of task t_1 is 0.2.
  2. **Execution time** C_4 **for task** t_4. Task t_4 has an execution time of $3\mu s$, which leads to a CPU utilization of 0.015.
  3. **Preemption by higher priority tasks with longer periods**. Task t_4 can be preempted by task t_2, which has a higher priority and a longer period. Preemption utilization of task t_2 is 0.14.
  4. **Blocking time by lower priority tasks**. Task t_4 may be blocked by lower priority task t_3 when it tries to obtain the next process value to be outputted on the digital output. As t_3 does need to block the process value for exlusive write access, we assume that blocking time will be around $5\mu s$, so a blocking utilization during period T_4 of $5\mu s/T_4 = 5\mu s/200\mu s = 0.025$ does result.

  The worst case utilization of task t_4 can therefore be computed to $0.2 + 0.015 + 0.14 + 0.025 = 0.38$, which is below the utilization bound of 0.69, so also task candidate t_4 will meet its deadline.

Fig. 18. Example Initial Schedulability Report

GUIDELINES

- *Use expert knowledge and approved techniques for estimation*: To analyze the schedulability of individual task candidates and the overall system, the CPU utilization of the task candidates has to be estimated. Without a good estimation of the CPU utilization, a significant statement about the schedulability of an individual task candidate or even the overall system can not be achieved in most cases.
  To obtain a meaningful estimation of the CPU utilization of each task candidate, expert knowledge of experienced designers is one of the most valuable input. Another possibility that can also be taken into consideration, is the development of a rapid prototype to measure the execution time of functions or algorithms that are hard to estimate. Also some theoretical approaches to estimate the CPU utilization based on formal reasoning have been developed by the research community. However, as neither of those has been able to prove its practical applicability yet, we propose to stick to use expert knowledge and rapid prototypes to obtain valid estimation data.

2.3 Architectural Design Discipline

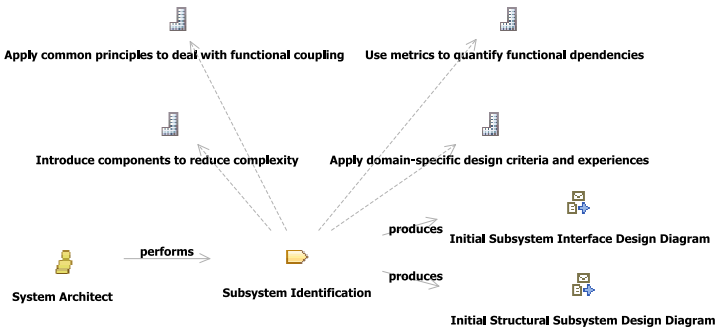
While the *Analysis* discipline emphasized on breaking down the problem domain, *Architectural Design* can be seen more as composing a solution. The central goal of the tasks comprised by the *Architectural Design* discipline is to develop the system architecture, which specifies subsystems, as well as the structural and behavioural relationships between them.

The internal decomposition of each identified subsystem is not regarded in detail during *Architectural Design* , as this is the objective of the succeeding *Detailed Design* discipline. What is done here, however, is the definition of the initial decomposition of the subsystems in terms of objects distributed amongst them.

In detail, the *Architectural Design* discipline is comprised of the following tasks:

- *Subsystems Identification* is done by grouping together the objects of the *Analysis Model* into groups of objects, denoted as subsystems to reduce the overall complexity. Each subsystem should show a high internal cohesion of the composed objects, while the overall system partition should establish a loose coupling between the identified subsystems.
- *Structural System Architecture Modelling* is done by defining required and provided interfaces for each subsystem, inferred them form the inter-object relationships of the analysis model. The structural system architecture is then constituted by the subsystems and the structural relationships between them, established via their required and provided interfaces.
- *Behavioural System Architecture Modelling* is done by regarding, how the use cases, which span more than one subsystem, are performed by the collaborative behaviour of the identified subsystems. Communication between subsystems can in this context be only established via the structural relationships identified in the previous step.
- *Task Design Consolidation* is done by clustering together active objects. This task may indeed be necessary if the overall task design, which can be inferred from the partitioning of the active analysis objects among the subsystems, is not feasable or its schedulability cannot be guaranteed. It is as well performed to exploit optimization potentials, so that more leeway for the later detailed design of the identified subsystems is gained.

Subsystem Identification



After having identified all application objects in the *Analysis* discipline and having specified their internal and external behaviour, it is now necessary to perform the first architectural design step, namely to group the identified trigger, interface, entity, control and application-logic objects into subsystems. According to Jacobson ([Ja92]), "the task of subsystems is to package objects in order to reduce the complexity."

Jacobson denotes two major principles that should be regarded during division of objects into subsystems (compare [Ja92]):

- Locality in changes: "If the system is to undergo a minor change, this change should concern no more than one subsystem. This means that the most important criterion for this subsystem division is predicting what the system changes will look like, and then making the division on the basis of this assumption."
- Functional coupling: "The division into subsystems should also be based on the functionality of the system. All objects which have a strong mutual functional coupling will be placed in the same subsystem [. . .]."

We want to extend the list by adding the following two major principles that will also have to be regarded:

- Task coupling: Jacobson also states that "another criterion for the division is that there should be as little communication between different subsystems as possible". We want to go further and want to emphasize that in the domain we are targeting, not only the pure amount of communication between different subsystems may be a criterion for the division, but also how the message sequences originating from the trigger objects - the tasks - are allocated to the subsystems. A central guide should therefore be that the allocation of tasks is done so that as few tasks as possible span subsystem boundaries. It should also be a general goal to reduce the synchronization overhead, which arises from a subsystem being affected by more than one task.
- Reuse: Another criterion that should be taken into consideration is the reusability of already existing subsystems. Analysis objects might be grouped together so that the functionality matches that of an already existing subsystem (maybe smaller changes have to be implemented), so that no new subsystem has to be developed but the already existing can be integrated instead. This may be most likely the case for basic service subsystems that do not provide domain-specific application-logic but deliver system-level services, like network communiation management or storage management.

After a group of objects has been decided to form a subsystem, the interaction points of the subsystem to its surrounding environment have to be defined. They can be determined by looking at the behavioural inter-object relationships of the analysis objects. Where communication between objects partitioned into different subsystems takes place, this communication has to enter or leave the subsystem via a defined interaction point, which we refer to as a port. Further, by differentiating on whether a message enters or leaves the subsystem, and by grouping them together, required and provided interfaces can be derived, which allow to detail the interaction established via each port.

WORK PRODUCTS

– *Initial Structural Subsystem Design Diagram*: The internal structure of each subsystem that has been obtained by grouping together objects from the analysis model is captured in a corresponding *Initial Structural Subsystem Design Diagram*. An example is shown in Figure 19.

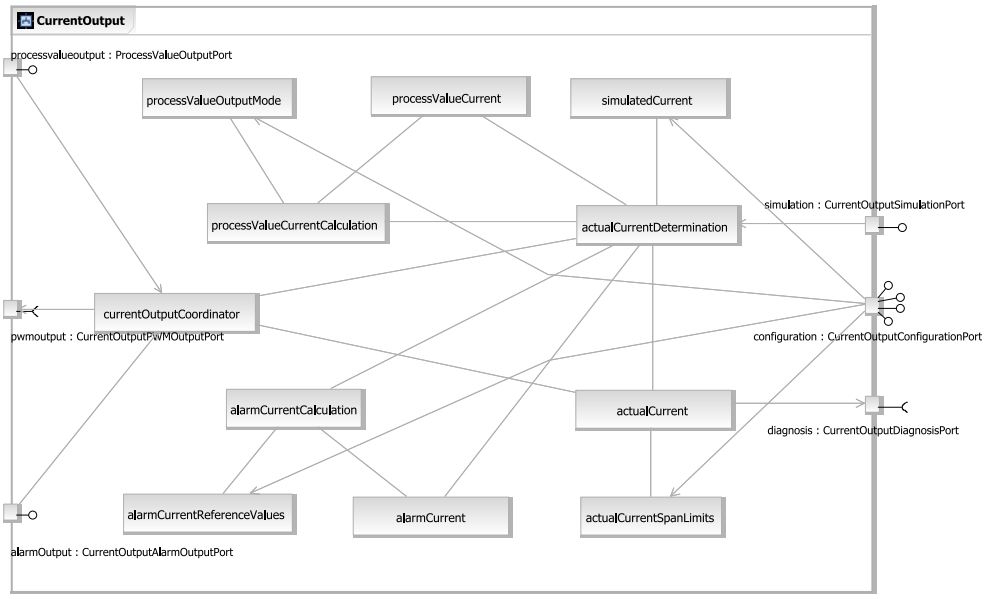


Fig. 19. Example: Intial Structural Subsystem Design Diagram

It is developed in terms of a UML2 composite structure diagram having the subsystem as the structured classifier with ports defining the interaction points of the subsystem towards its external evironment. The provided and required interfaces aggregated by each port are denoted by the so called *ball* and *socket* notation, which shows the interface in a symbolized form of a ball or a socket depending on whether it is a provided or required interface. The internal composite structure of the subsystem is modelled in terms of parts representing the trigger, interface, entity, control and application-logic objects partitioned into it. Relationships between the objects are modelled using assembly connectors. Where objects do have

relationships to external objects (which are partitioned into other subsystems) delegation connectors can be modelled to the port of the subsystem that encapsulates the interaction point towards this other subsystem. The diagram is denoted as *initial*, as the internal structure is obtained by just partitioning the analysis objects and be inferring the structural relationships from the analysis model, not being regarded further during this task. Indeed a consolidation of the internal subsystem design is addressed by the *Detailed Design* discipline.

– *Initial Subsystem Interface Design Diagram* As the *Initial Structural Subsystem Design Diagram*, which shows the external interaction points only in terms of ports, denoting their provided and required interfaces as stylized balls and sockets, an initial version of the externally visible provided and required interfaces' signatures has to be defined in an *Initial Subsystem Interface Design Diagram*. It is developed in form of a UML class diagram showing the required and provided interfaces grouping the messages entering and leaving the subsystem in the form of simple methods without return or call parameters.

Fig. 20. Example: Intial Subsystem Interface Design Diagram

GUIDELINES

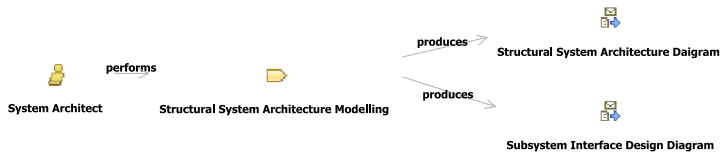
– *Apply common principles to deal with functional coupling*: Additional to the major design principles, Jacobson mentions some more concrete guidelines that can be applied to decide whether to place two objects into the same subsystem or not. For example, the following questions can be considered (compare [JCJv92]):

- Will changes of one object lead to changes in the other object?
- Do they communicate with the same actor?
- Are both of them dependent on a third object, such as an interface or entity object?
- Does one object perform several operations on the other?

Generally, our advice is to begin by placing a trigger or control object in a subsystem, and then place strongly coupled interface, application-logic and entity objects in the same subsystem.

- *Use metrics to quantify functional dependencies*: It may also be reasonable to be guided by metrics to determine the coupling between the objects (and cohesion of object clusters) during execution of this task. The number and frequency of messages exchanged between two objects could for example be an indicator to decide if those objects should reside inside one subsystem or could be separated into distinct ones. Other metrics are imaginable.
- *Apply domain-specific design criteria and experiences*: Further guidance to support the identification of subsystems may be inferred from domain-specific design principles. It may for example be common practice to introduce a central coordinator subsystem that takes care of coordinating the subsystems contributing to the most severe real-time tasks. Or it may be reasonable to group all objects handling the user interface together into a single user interface subsystem. Besides such common practices a standard architecture defined for the application-domain may be taken as a guidance for grouping objects into a prefined scheme. Reuse of existing subsystems may be named as a further source for inferring domain-specific design-criteria.
- *Introduce components to reduce complexity*: If an identified subsystem seems to be quite complex and it is reasonable to not split it into several subsystems, its internal decomposition should be designed in a hierarchical form. That is the internal decomposition is described in terms of component instances rather than objects. The components forming the subsystem decomposition are in turn formed by grouping together functionally related subclusters of the analysis objects partitioned into the subsystem.

Structural System Architecture Modelling



After having identified subsystems by grouping together the analysis objects, and after having defined interaction points (ports) with the required and provided interfaces of each subsystem, the next step is to describe how the subsystems are structurally related via those interfaces. It has be ensured that the subsystems' interfaces are designed so that they fit together. Therefore additional detail in form of method parameters has to be added to the initial subsystem interface definitions that were created by the previous task.

WORK PRODUCTS

– *Structural System Architecture Diagram*: The results of this task should be captured in one or more *Structural System Architecture Diagrams*, which are developed in terms of UML composite structure diagrams showing the subsystems, their provided and required interfaces and the structural relationships established via those interfaces. An example is shown in Figure 21.

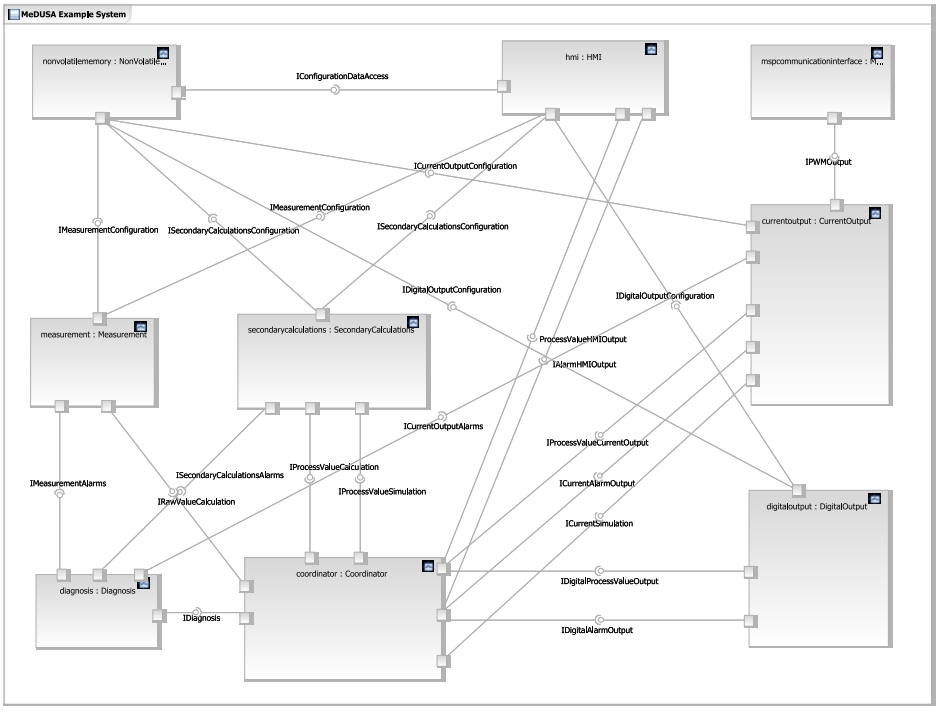


Fig. 21. Example: Structural System Architecture Diagram

– *Subsystem Interface Design Diagram*: Further, for each subsystem, the signature of the provided (and required) interfaces has to be defined in a *Subsystem Interface Design Diagram*, as shown in Figure 22. It is an advancement of the *Initial Subsystem Interface Design Diagram* developed during the*Subsystem Identification* task, where method parameters and their data types are additionally defined.

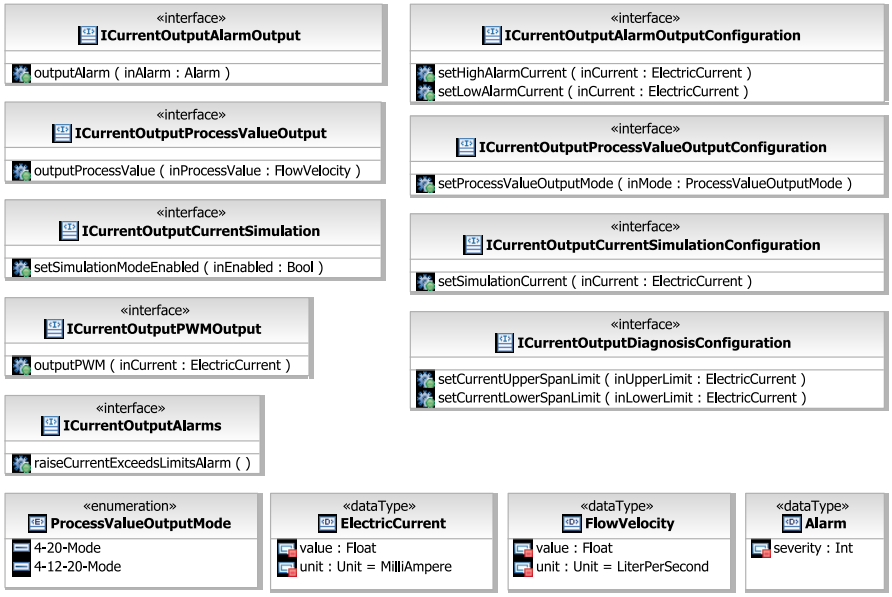


Fig. 22. Example: Subsystem Interface Design Diagram

It has to be pointed out that - even if we want to perform detailed class design as late as possible - for those classes and data types occuring in the method signatures, detailed class design has indeed to be done here. This is necessary, as distributed development of subsystems can only be done against well-defined interfaces, which includes that all data types passed via those interfaces are also well defined.

Behavioural System Architecture Modelling

After having the subsystems structurally integrated into the overall system architecture, it is necessary to describe, how system-wide use cases (that is affecting more than one subsystem) affect the subsystems via their provided/required interfaces. This is done by investigating how the use cases identified during requirements modelling are collaboratively performed by the subsystems on a system level (i.e. taking all use cases into account, that span more than one subsystem).

WORK PRODUCTS

– *Behavioural System Architecture Diagram*: The results of the *Behavioural System Architecture Modelling* task are captured in a sequence diagram for each use case identified during requirements modelling phase that is system-wide. Where include or extend relationships exist between use cases, interaction occurrences can be used in the including or extended use case, so that the included or extending use case can be modelled in a separate sequence diagram. As an example, Figure 23 shows a combined *Behavioural System Architecture Diagram* for the use cases `FlowRateCurrentOutput` and `AlarmCurrentOutput`.

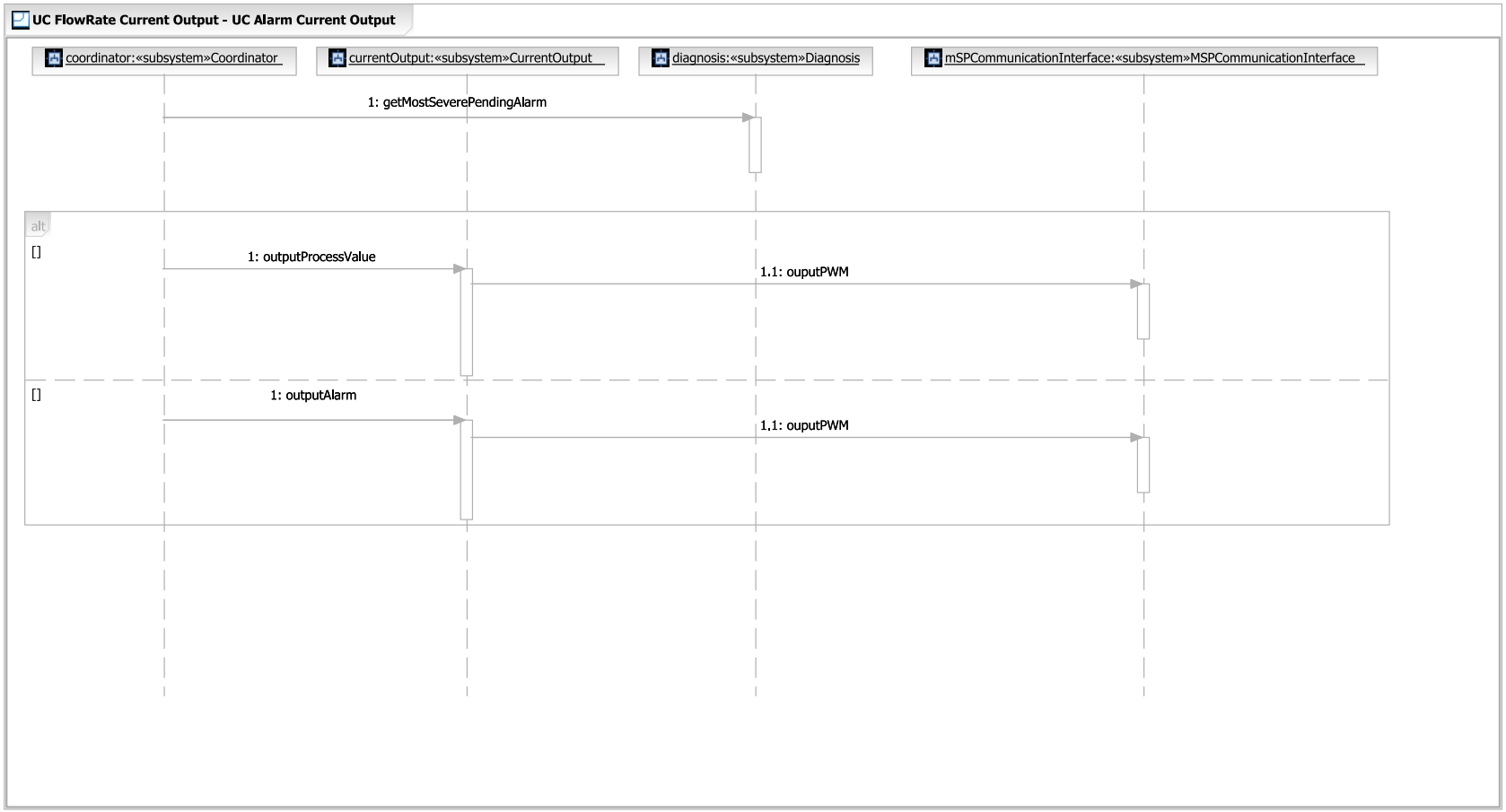


Fig. 23. Example: Behavioural System Architecture Diagram

Task Design Consolidation - Consolidate active objects

After the system architecture has been defined in terms of subsystems and their structural and behavioural relationships, the task architecture has to be consolidated. That is, optimization potentials have to be identified and exploited to give more leeway to the subsystem designer when elaborating the detailed design of each subsystem. That is, it has to be evaluated if trigger objects can be clustered together in order to reduce the overall number of tasks and thereby the inherent task overhead. Timer objects can for example by clustered together if they have the same period (or periods with a greatest common divisor greater than one) of it. Eventer objects may be clustered together if the events are indeed not truly concurrent events but do occur indeed in a sequential or mutually exlusive manner. We propose to refer to the task clustering criteria described in [Gom00] to get an inspiration on optimization potentials exploitable by clustering together trigger objects.

As an example consider that the `Calculation Chain Timer (100ms)` actor and the `Digital Output Timer (200`μ`s)` actor shown in the Use Case Diagram in Figure 6 have been transferred into respective `calculationChainTimer` and `digitalOutputTimer` analysis objects, who have been partitioned into two different subsystems. The `digitalOutputTimer` was of course partitioned into the `digitalOutput` subsystem, which is responsible of putting out the calculated process value or any pending alarms on the two digital outputs. The `calculationChainTimer` was put into the `secondaryCalculations` subsystem, which is responsible of calculating a process value from the raw value delivered by the `measurement` subsystem. As the periods of the two timers have a greatest common divisor greater than one and as the output of the digital process value depends of course on the calculation of the process value done by the `secondaryCalculations` subsystem, the decision could be taken to merge these two timer objects together into a `calculationAndOutputChainTimer` who resides in a third subsystem that coordinates the execution of the process value calculation and output. This way the task switching and synchronization overhead necessary for the two tasks could be economized and the overall performance could be improved.

WORK PRODUCTS

– *Consolidated Task Report*: After consolidation of the active objects has been done, the final task architecture can be inferred from the task candidates and the knowledge about which task candidates have been clustered together for optimization purposes. The results should be captured in a *Consolidated Task Report*, which is identical to the *Initial Task Report* produced during *Task Modelling* with the exception that it now indeed lists the tasks inherent to the system and no longer task candidates.

– *Consolidated Schedulability Report*: Besides the *Consolidated Task Report* capturing the individual tasks inherent to the system, a *Consolidated Schedulability Report* has to be developed, which demonstrates the overall schedulability of the system and indeed specifies a potential schedule. Its format is the same as that of the *Initial Schedulability Report* produced during *Task Modelling*.

2.4 Detailed Design Discipline

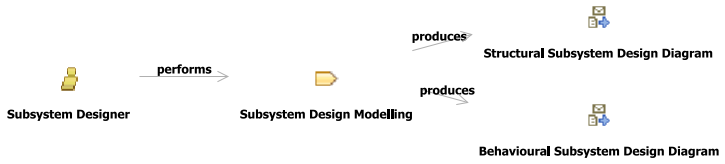
While the externally visible interfaces of each subsystem have been completely defined and the active objects partitioned into each subsystem have been consolidated regarding performance considerations, the intial subsystem design resulting from the partitioning of passive objects has still to be consolidated under design considerations. This is the first task of the *Detailed Design* discipline. Second, a detailed class design has to be developed that can be taken as direct input for implementation.

As Jacobson states, “Subsystems may also be used as handling units in the organization”. Taking this statement literally, it has to be pointed out that while the tasks of all other disciplines are indeed performed within the scope of the overall system, *Detailed Design* is performed distinctly for each subsystem, thus allowing an independent and potentially concurrent processing of each.

Detailed Design comprises the following tasks:

- *Subsystem Design Consolidation* deals with the consolidation of the internal subsystem decomposition under design considerations. That is, all passive objects (the active ones have already been consolidated during *Task Design Consolidation*) have to be examined regarding how they can be reasonably mapped to a detailed class design. The aim is, to modify the initial object collaboration - if necessary - so that it can serve as a reasonable basis for implementation.
- *Class Design Modelling* deals with developing a detailed class design for the internal subsystem decomposition, which is established in terms of objects.

Subsystem Design Modelling - Consolidate passive objects



The analysis objects themselves as well as the collaborations of those that were grouped together to form subsystems during *Architectural Design* were of course not identified from a design but from an analysis viewpoint. Therefore, most likely they have to be consolidated, so that a detailed class design can be reasonably applied to them.

In detail, during consolidation it should be checked, if the objects have to be removed from the internal subsystem decomposition, or if they have to be split or merged together. Removing an object from the internal subsystem decomposition may be reasonable when for example an entity object is indeed not decided to be stored inside the subsystem but is just passed into it, out of it or between two objects of its' internal decomposition as a mere parameter. Splitting an object may be appropriate where a weak cohesion of the resulting class is likely. Merging objects may be necessary where a very strong cohesion between the resulting classes would result.

WORK PRODUCTS

– *Structural Subsystem Design Diagram*: A *Structural Subsystem Design Diagram* has to be developed for each subsystem, which is an advancement to the *Initial Subsystem Design Diagram* developed during *Subsystem Identification*. Like the *Initial Subsystem Design Diagram* it is developed in form of a UML composite structure diagram, showing the subsystem as the structured classifier having ports aggregating the required and provided interfaces, via which communication with the external environment is established, and having an internal structure in terms of interconnected parts representing the composed objects.

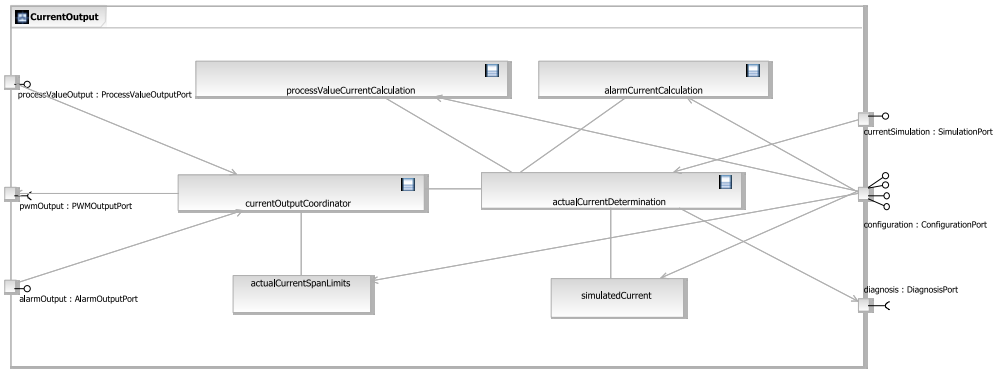


Fig. 24. Example: Structural Subsystem Design Diagram

– *Behavioural Subsystem Design Diagram*: Changes to the structural subsystem design will imply changes to its behavioural design as well. Therefore, one or more *Behavioural Subsystem Design Diagrams* should be developed in terms of a UML sequence or communication diagram that show how the internal object decomposition collaborative performs if either triggered internally by a trigger object or externally via one of the ports of the subsystem. An example for a *Behavioural Subsystem Design Diagram* is shown in Figure 25.

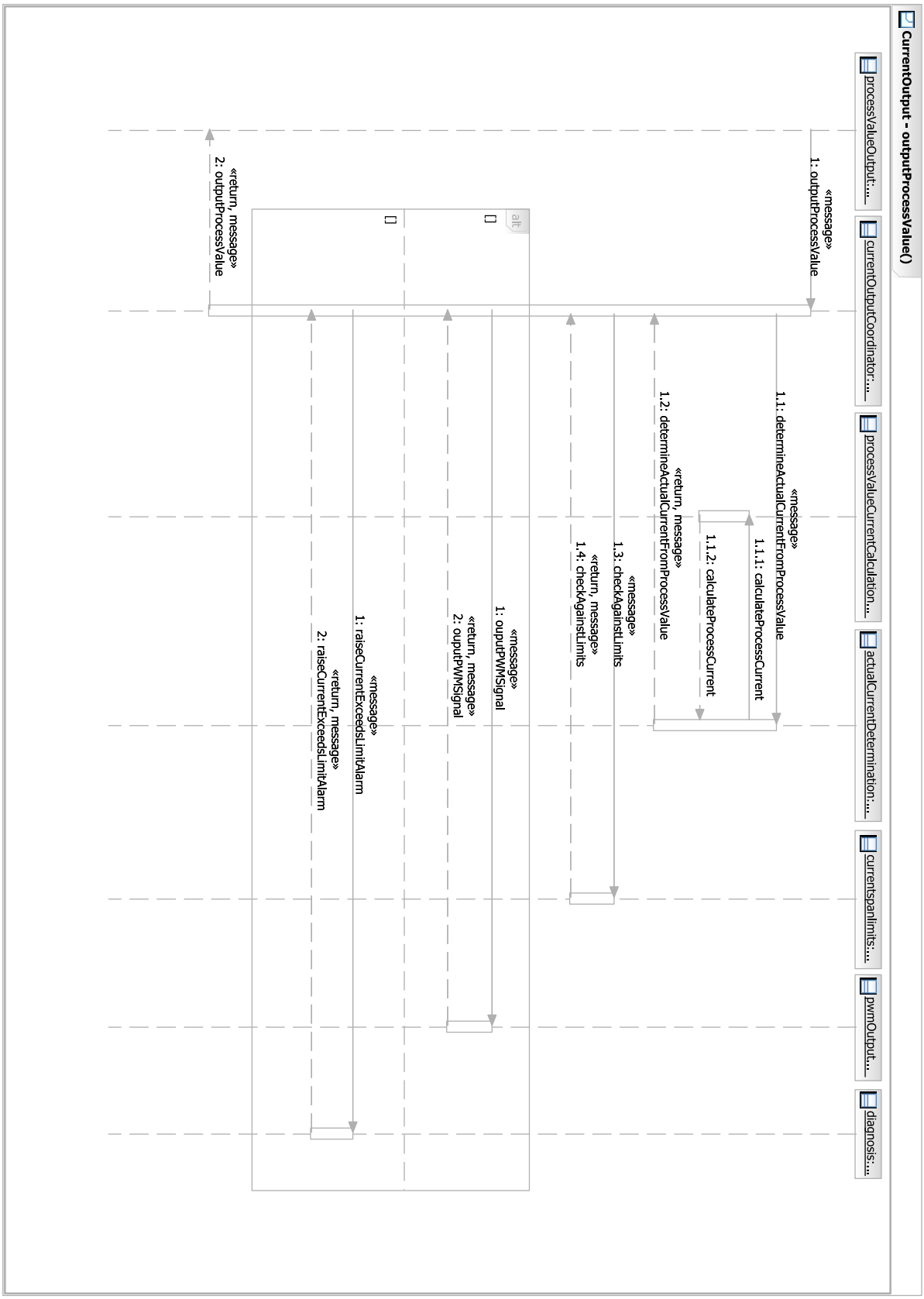
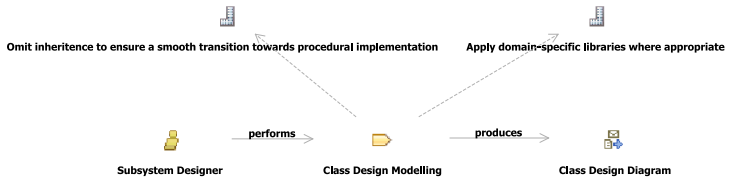


Fig. 25. Example: Behavioural Subsystem Design Diagram

Class Design Modelling



After the object collaboration forming the subsystem has been consolidated, the detailed class design for all objects composed by the subsystem has to be developed. That is, classes have to be designed for the identified objects, having attributes corresponding to the slots of the objects and operations corresponding to the messages the object receives. Where objects are connected, associations have to be designed on the class level.

WORK PRODUCTS

– *Class Design Diagram*: The results of the *Class Design Modelling* should be documented in UML class diagrams, which show the classes with their attributes and operations as well as associations between them. The class diagram should be so detailed that it can be taken as a building plan for implementation, meaning that attribute types, as well as the types and names of operation parameters are included. An example for the detailed class design is shown in Figure 26.

Fig. 26. Example: Class Design Diagram

GUIDELINES

- *Omit inheritance to ensure a smooth transition towards procedural implementation*: As the implementation languages of the regarded application domain are procedural languages (C-language or one of its derivates) MeDUSA aims at developing a design model that can be easily transferred to such a procedural implementation model (probably tool-supported). This is why we designed the method to be object-based rather than object-oriented; the instance-driven nature of the method causes that inheritance and related polymorphism mechanisms are not regarded, as up to now instances (objects) rather than classes were modelled. Therefore we propose to omit the application of inheritance and related polymorphism concepts also during this last step. However, if it seems reasonable to apply inheritance mechanisms during detailed class design modelling, this can be done. The only thing we want to emphasize is that all mappings from object-oriented design models to procedural implementation languages tend to cause the source code to be not adequately readable and also tend to harden traceability between the design model and the source code.
- *Apply domain-specific libraries where appropriate*: As the class design is most often developed in a distributed manner - and rather late - it may happen that certain functionality that is needed inside several subsystems is designed plural and would therefore also be implemented plural. If this affects objects that are exchanged between subsystems (i.e. the object is exchanged via provided/required interfaces of the subsystem) synchronization has in either case to be guaranteed to ensure that subsystems may be interconnected. If it affect objects that are not visible to other subsystems, this is not necessarily to be ensured. However, it would be desirable to reuse such functionality regarding detailed class design as well as implementation. In either case, from the experience gained, such multiple occurrence of functionality is most often the case for such objects that are rather application-domain specific than device-specific, like e.g. in case of physical quantities. We propose to build up a class library to achieve best reuse in such a situation.

3 Process

Expressed in the terminology of the *Unified Method Architecture*, a process describes "how the method content defined with tasks, roles, and work products is sequenced" over time. That is, like denoted by Figure 27, it is defined at which point in time and in which order the tasks are performed by their respective roles.

Fig. 27. UMA process concepts (copied from [Hau06])

Basically, the definition of a process is - according to UMA - based around the concept of *Activity*. Activities contain references to the method content, can be nested to define a hierarchical breakdown structure, and can be related to each other. UMA defines two major kinds of processes, namely *Capability Patterns* and *Delivery Processes* (which are themselves activities). A *Delivery Process* defines a complete and integrated process spanning a complete project lifecycle. *Capability Patterns* define processes (or process fractions) that are related to a key area of interest such as a discipline or a best practice. They can be directly used but can also be employed as building blocks to define a delivery process.

The definition of the MeDUSA process is based on four such basic capability patterns that define the sequencing of the tasks belonging to the four disciplines of the method content definition. We refer to them as *Requirements Modelling*, *Analysis Modelling*, *Architectural Design Modelling*, and *Detailed Design Modelling*. The MeDUSA delivery process itself is defined in terms of four phases, namely *Requirements Modelling Phase*, *Analysis Modelling Phase*, *Architectural Design Modelling Phase*, and *Detailed Design Modelling Phase* as shown in Figure 28.

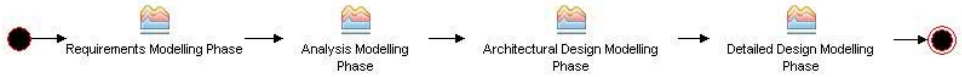


Fig. 28. MeDUSA Delivery Process

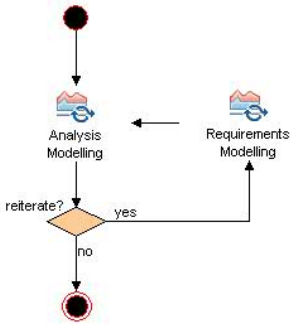
All phases are self-contained units, concerned primarily with the iterative execution of the activities defined by the homonymous *Requirements Modelling*, *Analysis*

Modelling, *Architectural Design Modelling* and *Detailed Design Modelling* capability patterns. The goal of each phase is to develelop a consise model (i.e. a *Requirements Model* at the end of the *Requirements Modelling Phase*, an *Analysis Model* at the end of the *Analysis Modelling Phase* and so on). Besides the primary activities, belonging to the homonymous capability pattern, each phase allows the iterative execution of other preceeding activities if changes to previous models are needed. That is, if for example in the *Architectural Design Modelling Phase* a change resulting in the *Analysis Model* is noticed while performing *Architectural Design Modelling*, this iteration can be skipped an the *Analysis Modelling* activities can be reiterated.

Requirements Modelling Phase

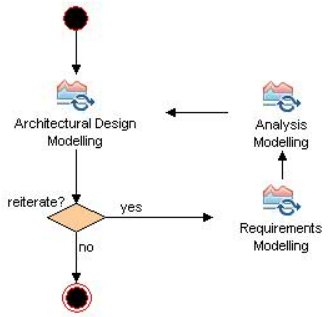
The *Requirements Modelling Phase* is indeed concerned with executing the tasks as executed by the *Requirements Modelling* capability pattern. That is, all activities comprised by the *Requirements Modelling* capability pattern are executed iteratively, as long as a thoroughly defined *Requirements Model* has been developed and the next phase can be entered.

Analysis Modelling Phase



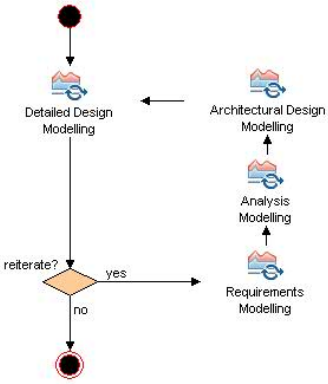
The *Analysis Phase* is concerned with producing a consise *Analysis Model*. Here the activities defined by the *Analysis Modelling* capability pattern are executed iteratively. It may happen, that by executing the *Analysis Modelling* activities, awareness about weaknesses or necessary changes to the *Requirements Model* may arise. Therefore the *Analysis Modelling Phase* allows to again iterate through the *Requirements Modelling* activities as well.

Architectural Design Modelling Phase



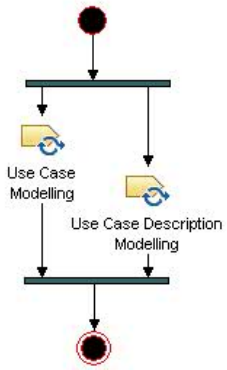
The *Architectural Design Phase* deals with the definition of a system architecture. That is, the *Architectural Design Modelling* activities are iteratively performed to construct a consise *Architectural Design Model* that defines the system architecture in terms of structurally and behaviourally related subsystems. As changes to the *Analysis Model* or even the *Requirements Model* may be necessary, backflows to *Analysis Modelling* and *Requirements Modelling* are permitted.

Detailed Design Modelling Phase



The *Detailed Design Phase* forms the last phase of the MeDUSA delivery process. Its objective is to deliver a *Detailed Design Model* that can be taken as input to the implementation. Therefore, the *Detailed Design Modelling* activities are performed within it. However, in distinction to the previous phases, *Detailed Design Modelling* is not performed at once for the overall system, but may be performed individually for each subsystem identified during the previous *Architectural Design Modelling Phase*. Therefore, if changes to the *Architectural Design Model* (or even *Analysis Model* or *Requirements Model*) are determined, and the *Architectural Design Modelling* (and possibly *Analysis Modelling* and *Requirements Modelling*) activities are performed again to incorporate these changes, the following iteration of *Detailled Design Modelling* has to be executed for all subsystems that are affected by those changes and not only for the one subsystem, during whose *Detailed Design Modelling* the changes were noticed.

3.1 Requirements Modelling

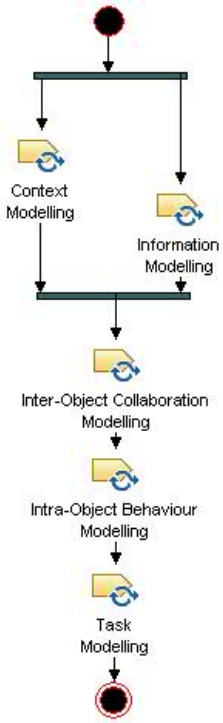


Requirements Modelling is concerned with the construction of a *Requirements Model*, which is indeed a use case model that manifests itself in a UML use case model and a narrative use case model. As UML model and narrative model have to be consistent to each other and as either of them may serve as valuable input for the construction of the other - e.g. when trying to determine the right granularity - the two activities concerned with the construction of the two model fractions are executed more or less in parallel. Often, one starts to construct the UML use case by constructing an initial version of a *Use Case Diagram* first. Then, a *Use Case Description* for each initially identified use case may be started, which may lead in turn to modifications inside the *Use Case Diagram(s)* or use case model respectively, as additional use cases or relationships between use cases may be identified. This way, use case model and narrative use case model are developed in parallel, until they are consistent to each other and their quality is satisfying to proceed with the *Analysis Modelling* activities.

It has to be pointed out here again that MeDUSA is a use case driven method, meaning that the use cases identified during *Requirements Modelling* play a very central role throughout all following activities of the method. From the identification of analysis objects during *Analysis Modelling*, up to the *Architectural Design Modelling*, use cases are the central artefacts around which the activities of the method are organized.

Therefore *Requirements Modelling* is a very essential activity of the MeDUSA method, which has to be performed very thoroughly. Defects and misunderstandings not resolved here can cause costly fixes in later activities. Especially misunderstandings and mistakes related to the concurrency concerns, which are of major interest already during these early activities, may have severe impact on the later on developed *Analysis Model* and *Architectural Design Model* if not regarded thoroughly. It has be pointed out therefore that even if the regarded domain is already well understood or similar products in the regarded domain have already been developed, it is essential to perform this step in the described detail.

3.2 Analysis Modelling

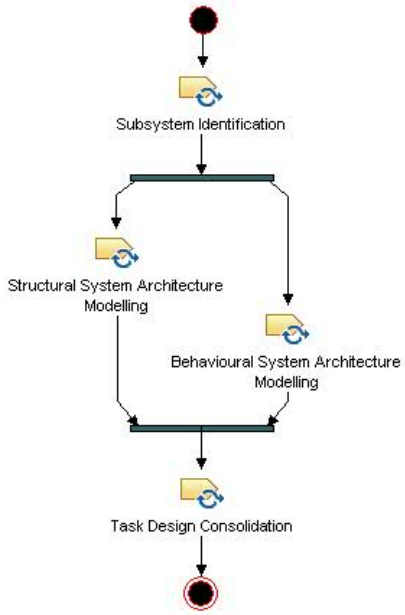


After having investigated functional and the non-functional timing and concurrency requirements during *Requirements Modelling*, *Analysis Modelling* is concerned with the construction of an *Analysis Model*. That is, a thorough understanding of the problem domain has to be achieved by modelling an object collaboration for each use case identified during *Requirements Modelling*. The starting point for the identification of the objects that collaboratively perform the identified use cases is the identification of trigger, interface and entity objects, which is regarded during *Context Modelling* and *Information Modelling*. As both activities are concerned with different aspects - the one with the interfaces to the external environment, the other with internal data - they can be performed very much in parallel. After trigger, interface and entity objects have been defined, the remaining activities are concerned with identifying additional control and application-logic objects needed to execute the use cases, and to specify their internal behaviour (in case it is not trivial). Last, a first impression on the capability of the software system to meet its performance constraints has to be gained by an intial task schedule and schedulability report.

After having performed the last *Analysis Modelling* activity, all interface, trigger, entity, control and application-logic objects that build up the *Analysis Model* should be identified and their collaborating behaviour for each of the use cases identified during *Requirements Modelling* should have been captured. Also, the internal behavior of each object should be captured where it is not trivial, and the tasks (candidates) originating from the active trigger objects should have been analyzed. That is, with the end of *Analysis Modelling* activities, a thorough understanding of the problem domain should have been achieved, so that *Architectural Design* can be started.

What has to be pointed out is that especially *Analysis Modelling* does have a very iterative nature. That is, several iterations through all of its activities will be needed to develop an appropriate *Analysis Model*. Often, entity objects are not directly identified during *Information Modelling* but not earlier than during *Inter-Object Collaboration Modelling*, when the object collaborations performing the use cases are developed. It may also be the case that during the activities of the *Analysis Modelling* interface or trigger objects are identified that were not already determined during *Requirements Modelling*. This is where the iteration has to be interrupted (respectively cancelled) and a backflow to those earlier activities is needed.

3.3 Architectural Design Modelling



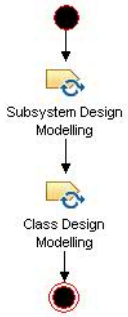
The objective of the *Architectural Design Modelling* activities is the specification of the software system's architecture. That is, subsystems have to be identified during *Subsystem Identification* by grouping together objects from the *Analysis Model*. An initial version of each subsystems' required and provided interfaces can also be inferred from the messages exchanged between objects partitioned into different subsystems. After that, the structural and behavioural relationships between the identified subsystems can be designed. This is done during *Structural System Architecture Modelling* and *Behavioural System Architecture Modelling*. The definition of structural and behavioural relationships is strongly intertwined. Of course, one might start with defining an initial version of the structural relationships first, as the behavioural relationships have to reside on them. However, while modelling the behavioural relationships, changes to the structural relationships are likely to occur, so that from the initial defintion of the structural relationships onwards both activities will be executed very much in parallel.

After the system architecture in terms of subsystems and their structural and behavioural relationships has been defined, the task allocation has to be reflected. That is, during *Task Design Consolidation*, the distribution of the active trigger objects among the subsystems has to be regarded to investigate optimization potential and to review the feasibility of the chosen task allocation. After all optimization potentials have been exploited - which might make it necessary to start over with some of the preceeding activities - the overall schedulability of the final task design has to be proved with the help of a consolidated task and schedulability report.

It has to be pointed out that *Architectural Design Modelling* is rather iterative. First in itself, as the activities may have to be gone through in several iterations until a feasible system architecture has been specified. Second in a sense that most likely changes to the *Analysis Model* will be noticed while developing the *Architectural De-*

sign Model. An example provided by Jacobson might help to demonstrate this (compare [JCJv92]): “ When the division into subsystems is made, in some cases it may also be desirable to modify the analysis objects also. This may be the case, for instance, when an entity [or application-logic] object has separate behavior that is functionally related to more than one subsystem. If this behaviour is extracted, it may be easier to place the entity object in a subsystem.”

3.4 Detailed Design Modelling



After having performed the *Architectural Design Modelling*, all externally visible required and provided interfaces of the identified subsystems are clearly specified and the system behaviour that occurs over these interfaces is well understood. *Detailled Design Modelling* is now first of all concerned with designing the internal subsystem decomposition, which was up to now only expressed in terms of analysis objects partitioned into the subsystem. After that, during *Class Design Modelling*, a class design has to be created that can taken rather seamlessly as input to the succeeding implementation activities.

Of course, as with the other capability patterns, *Detailed Design* is an iterative activity. That is, several iterations will be needed until the developed subsystem decomposition will be transferrable in a feasible class design. Further, changes to the *Architectural Design Model* may be necessary if the consolidation of a subsystem might lead to the insight, that the system architecture is indeed not adequate. Severe changes on the *Architectural Design Model* are rather unlikely as the externally visible interfaces of each subsystem have been thoroughly defined and validated during *Architectural Design Modelling*, but are indeed not impossible. As with all other activities, the iterative nature of the method manifests itself here, as well.

4 Summary & Conclusion

Having started as a mere advancement to the COMET method developed by Hassan Gomaa ([Gom00]), following the goal to overcome those shortcomings the COMET method showed during its practical application in ABB Business Unit Instrumentation, MeDUSA has undergone several changes and has in the last two years grown into an independent and self-contained design method.

As we stated in the introduction, MeDUSA was initially designed having in mind three major design objectives resulting from the requirements of the application domain it was intended for, namely

- to regard the non-functional constraints related to memory and power consumption and computing time, as well as the real-time constraints small embedded software systems have to meet.
- to support distributed development of reusable components.
- to support a standard-based notation that allows to choose from a broad set of market-available modelling tools.

MeDUSA faces the first objective by its object-based and instance-driven nature. That is, as class design is done quite late, a seamless transition into C procedural implementation language can be easily achieved. Further, non-functional timing and concurrency constraints are regarded right from the beginning, performance problems, as well as optimization potentials are investigated as soon as possible.

The second objective is addressed by supporting the distributed development of subsystems, which are regarded as the major reusable assets. Having split the design activities into an *Architectural Design Modelling* and a *Detailled Design Modelling* part, MeDUSA forces the distributed development of subsystems, while trying to ensure that those distributedly developed subsystems, as well as existing ones being selected for reuse, can be easily and seamlessly integrated.

The last objective is met with the decision to choose the UML as the underlying notation. This decision was in our eyes quite natural, as the UML is indeed the only notation standard that has found an acceptance, broad enough to deserve that denomination. However, the UML notation also showed some shortcomings that resulted from its application to the domain of embedded systems. As an example for this consider that an adequate representation of interfaces, residing on different levels of abstraction, cannot be modelled in use case diagrams. We will have to investigate such problems in the future.

While the decision to use an industry standard for the notation does allow to use market-available modelling tools to support the execution of the method, the full potential of the MeDUSA method can of course only be unleashed if customized support is delivered. This is the reason why we are currently developing a research prototype [ViP] - named ViPER - to demonstrate the benefits of a specially customized MeDUSA tool support.

As already stated before, MeDUSA was not designed “in the open countryside” of university research but in close cooperation with industrial practitioners. Therefore, development of MeDUSA does not stop with the publishing of this report. As more and more experience from its practical application can be gained, it will likely change in the future - as it has done in the past. We will therefore publish updates to the

method - as well as to this report - on the MeDUSA project web page [MeD], which also contains a hypertext documentation (HTML) of the method as well as further supporting material.

References

[BS02] Kurt Bittner and Ian Spence. Use Case Modeling. Addison Wesley, 2002.

[Coc01] Alistar Cockburn. Writing Effective Use Cases. Addison Wesley, The agile software development series, 2001.

[Fow03] Martin Fowler. UML Distilled - Third Edition. Addison Wesley Object Technology Series, 2003.

[GHH+04] H. Grothey, C. Habersetzer, M. Hiatt, W. Hogrefe, M. Kirchner, G. Lütkepohl, W. Marchewka, U. Mecke, M. Ohm, F. Otto, K.-H. Rackebrandt, M. Schönsee, D. Sievert, A. Thöne, and H.-J. Wegener. Praxis der industriellen Durchflussmessung. ABB Automation Products, Werk Göttingen, 2004.

[Gom00] Hassan Gomaa. Designing Concurrent, Distributed, and Real-Time Applications with UML. Addison Wesley, Object Technology Series, 2000.

[Hau05] Peter Haumer. IBM Rational Method Composer - Part 1: Key Concepts. Rational Edge, www.ibm.com/developerworks/rational/library/dec05/haumer/index.html, 2005.

[Hau06] Peter Haumer. IBM Rational Method Composer - Part 2: Authoring method content and processes. Rational Edge, www.ibm.com/developerworks/rational/library/jan06/haumer/index.html, 2006.

[JCJv92] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Övergaard. Object-Oriented Software Engeneering - A Use Case Driven Approach. 1992.

[JRH+04] Mario Jeckle, Chris Rupp, J"urgen Hahn, Barbara Zengler, and Stefan Queins. UML2 glasklar. Carl Hanser Verlag, Wien, 2004.

[MeD] MeDUSA project site. http://www.medusa.sc.

[NLS+05] Alexander Nyßen, Horst Lichter, Jan Suchotzki, Peter Müller, and Andreas Stelter. UML2-basierte Architekturmodellierung kleiner eingebetteter Systeme - Erfahrungen einer Feldstudie. volume TUBS-SSE-2005-01, 2005.

[NMSL04] Alexander Nyßen, Peter Müller, Jan Suchotzki, and Horst Lichter. Erfahrungen bei der systematischen Entwicklung kleiner eingebetteter Systeme mit der COMET-Methode. Lecture Notes in Informatics (LNI) Modellerierung 2004, P-45:229–234, 2004.

[SPE05] Software Process Engineering Metamodel Specification - Version 1.1. http://www.omg.org/cgi-bin/doc?formal/2005-01-06, 2005.

[SPE06] Software & Systems Process Engineering Metamodel, OMG Draft Adopted Specification. http://www.omg.org/cgi-bin/doc?ad/06-06-02, 2006.

[ViP] ViPER project site. http://www.viper.sc.

[Wal07] Andreas Walter. Ein Use Case-Modellierungswerkzeug für die ViPER-Plattform. diploma thesis, RWTH Aachen University, 2007.

Aachener Informatik-Berichte

This list contains all technical reports published during the past five years. A complete list of reports dating back to 1987 is available from http://aib.informatik.rwth-aachen.de/. To obtain copies consult the above URL or send your request to: Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen, Email: `biblio@informatik.rwth-aachen.de`

2001-01 * Jahresbericht 2000

2001-02 Benedikt Bollig, Martin Leucker: Deciding LTL over Mazurkiewicz Traces

2001-03 Thierry Cachat: The power of one-letter rational languages

2001-04 Benedikt Bollig, Martin Leucker, Michael Weber: Local Parallel Model Checking for the Alternation Free mu-Calculus

2001-05 Benedikt Bollig, Martin Leucker, Thomas Noll: Regular MSC Languages

2001-06 Achim Blumensath: Prefix-Recognisable Graphs and Monadic Second-Order Logic

2001-07 Martin Grohe, Stefan Wöhrle: An Existential Locality Theorem

2001-08 Mareike Schoop, James Taylor (eds.): Proceedings of the Sixth International Workshop on the Language-Action Perspective on Communication Modelling

2001-09 Thomas Arts, Jürgen Giesl: A collection of examples for termination of term rewriting using dependency pairs

2001-10 Achim Blumensath: Axiomatising Tree-interpretable Structures

2001-11 Klaus Indermark, Thomas Noll (eds.): Kolloquium Programmiersprachen und Grundlagen der Programmierung

2002-01 * Jahresbericht 2001

2002-02 Jürgen Giesl, Aart Middeldorp: Transformation Techniques for Context-Sensitive Rewrite Systems

2002-03 Benedikt Bollig, Martin Leucker, Thomas Noll: Generalised Regular MSC Languages

2002-04 Jürgen Giesl, Aart Middeldorp: Innermost Termination of Context-Sensitive Rewriting

2002-05 Horst Lichter, Thomas von der Maßen, Thomas Weiler: Modelling Requirements and Architectures for Software Product Lines

2002-06 Henry N. Adorna: 3-Party Message Complexity is Better than 2-Party Ones for Proving Lower Bounds on the Size of Minimal Nondeterministic Finite Automata

2002-07 Jörg Dahmen: Invariant Image Object Recognition using Gaussian Mixture Densities

2002-08 Markus Mohnen: An Open Framework for Data-Flow Analysis in Java

2002-09 Markus Mohnen: Interfaces with Default Implementations in Java

2002-10 Martin Leucker: Logics for Mazurkiewicz traces

2002-11 Jürgen Giesl, Hans Zantema: Liveness in Rewriting

2003-01 * Jahresbericht 2002

2003-02 Jürgen Giesl, René Thiemann: Size-Change Termination for Term Rewriting

2003-03 Jürgen Giesl, Deepak Kapur: Deciding Inductive Validity of Equations

2003-04 Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, Stephan Falke: Improving Dependency Pairs

2003-05 Christof Löding, Philipp Rohde: Solving the Sabotage Game is PSPACE-hard

2003-06 Franz Josef Och: Statistical Machine Translation: From Single-Word Models to Alignment Templates

2003-07 Horst Lichter, Thomas von der Maßen, Alexander Nyßen, Thomas Weiler: Vergleich von Ansätzen zur Feature Modellierung bei der Softwareproduktlinienentwicklung

2003-08 Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, Stephan Falke: Mechanizing Dependency Pairs

2004-01 * Fachgruppe Informatik: Jahresbericht 2003

2004-02 Benedikt Bollig, Martin Leucker: Message-Passing Automata are expressively equivalent to EMSO logic

2004-03 Delia Kesner, Femke van Raamsdonk, Joe Wells (eds.): HOR 2004 – 2nd International Workshop on Higher-Order Rewriting

2004-04 Slim Abdennadher, Christophe Ringeissen (eds.): RULE 04 – Fifth International Workshop on Rule-Based Programming

2004-05 Herbert Kuchen (ed.): WFLP 04 – 13th International Workshop on Functional and (Constraint) Logic Programming

2004-06 Sergio Antoy, Yoshihito Toyama (eds.): WRS 04 – 4th International Workshop on Reduction Strategies in Rewriting and Programming

2004-07 Michael Codish, Aart Middeldorp (eds.): WST 04 – 7th International Workshop on Termination

2004-08 Klaus Indermark, Thomas Noll: Algebraic Correctness Proofs for Compiling Recursive Function Definitions with Strictness Information

2004-09 Joachim Kneis, Daniel Mölle, Stefan Richter, Peter Rossmanith: Parameterized Power Domination Complexity

2004-10 Zinaida Benenson, Felix C. Gärtner, Dogan Kesdogan: Secure Multi-Party Computation with Security Modules

2005-01 * Fachgruppe Informatik: Jahresbericht 2004

2005-02 Maximillian Dornseif, Felix C. Gärtner, Thorsten Holz, Martin Mink: An Offensive Approach to Teaching Information Security: "Aachen Summer School Applied IT Security"

2005-03 Jürgen Giesl, René Thiemann, Peter Schneider-Kamp: Proving and Disproving Termination of Higher-Order Functions

2005-04 Daniel Mölle, Stefan Richter, Peter Rossmanith: A Faster Algorithm for the Steiner Tree Problem

2005-05 Fabien Pouget, Thorsten Holz: A Pointillist Approach for Comparing Honeypots

2005-06 Simon Fischer, Berthold Vöcking: Adaptive Routing with Stale Information

2005-07 Felix C. Freiling, Thorsten Holz, Georg Wicherski: Botnet Tracking: Exploring a Root-Cause Methodology to Prevent Distributed Denial-of-Service Attacks

2005-08 Joachim Kneis, Peter Rossmanith: A New Satisfiability Algorithm With Applications To Max-Cut

2005-09 Klaus Kursawe, Felix C. Freiling: Byzantine Fault Tolerance on General Hybrid Adversary Structures

2005-10 Benedikt Bollig: Automata and Logics for Message Sequence Charts

2005-11 Simon Fischer, Berthold Vöcking: A Counterexample to the Fully Mixed Nash Equilibrium Conjecture

2005-12 Neeraj Mittal, Felix Freiling, S. Venkatesan, Lucia Draque Penso: Efficient Reductions for Wait-Free Termination Detection in Faulty Distributed Systems

2005-13 Carole Delporte-Gallet, Hugues Fauconnier, Felix C. Freiling: Revisiting Failure Detection and Consensus in Omission Failure Environments

2005-14 Felix C. Freiling, Sukumar Ghosh: Code Stabilization

2005-15 Uwe Naumann: The Complexity of Derivative Computation

2005-16 Uwe Naumann: Syntax-Directed Derivative Code (Part I: Tangent-Linear Code)

2005-17 Uwe Naumann: Syntax-directed Derivative Code (Part II: Intraprocedural Adjoint Code)

2005-18 Thomas von der Maßen, Klaus Müller, John MacGregor, Eva Geisberger, Jörg Dörr, Frank Houdek, Harbhajan Singh, Holger Wußmann, Hans-Veit Bacher, Barbara Paech: Einsatz von Features im Software-Entwicklungsprozess - Abschlußbericht des GI-Arbeitskreises "Features"

2005-19 Uwe Naumann, Andre Vehreschild: Tangent-Linear Code by Augmented LL-Parsers

2005-20 Felix C. Freiling, Martin Mink: Bericht über den Workshop zur Ausbildung im Bereich IT-Sicherheit Hochschulausbildung, berufliche Weiterbildung, Zertifizierung von Ausbildungsangeboten am 11. und 12. August 2005 in Köln organisiert von RWTH Aachen in Kooperation mit BITKOM, BSI, DLR und Gesellschaft fuer Informatik (GI) e.V.

2005-21 Thomas Noll, Stefan Rieger: Optimization of Straight-Line Code Revisited

2005-22 Felix Freiling, Maurice Herlihy, Lucia Draque Penso: Optimal Randomized Fair Exchange with Secret Shared Coins

2005-23 Heiner Ackermann, Alantha Newman, Heiko Röglin, Berthold Vöcking: Decision Making Based on Approximate and Smoothed Pareto Curves

2005-24 Alexander Becher, Zinaida Benenson, Maximillian Dornseif: Tampering with Motes: Real-World Physical Attacks on Wireless Sensor Networks

2006-01 * Fachgruppe Informatik: Jahresbericht 2005

2006-02 Michael Weber: Parallel Algorithms for Verification of Large Systems

2006-03 Michael Maier, Uwe Naumann: Intraprocedural Adjoint Code Generated by the Differentiation-Enabled NAGWare Fortran Compiler

2006-04 Ebadollah Varnik, Uwe Naumann, Andrew Lyons: Toward Low Static Memory Jacobian Accumulation

2006-05 Uwe Naumann, Jean Utke, Patrick Heimbach, Chris Hill, Derya Ozyurt, Carl Wunsch, Mike Fagan, Nathan Tallent, Michelle Strout: Adjoint Code by Source Transformation with OpenAD/F

2006-06 Joachim Kneis, Daniel Mölle, Stefan Richter, Peter Rossmanith: Divide-and-Color

2006-07 Thomas Colcombet, Christof Löding: Transforming structures by set interpretations

2006-08 Uwe Naumann, Yuxiao Hu: Optimal Vertex Elimination in Single-Expression-Use Graphs

2006-09 Tingting Han, Joost-Pieter Katoen: Counterexamples in Probabilistic Model Checking

2006-10 Mesut Günes, Alexander Zimmermann, Martin Wenig, Jan Ritzerfeld, Ulrich Meis: From Simulations to Testbeds - Architecture of the Hybrid MCG-Mesh Testbed

2006-11 Bastian Schlich, Michael Rohrbach, Michael Weber, Stefan Kowalewski: Model Checking Software for Microcontrollers

2006-12 Benedikt Bollig, Joost-Pieter Katoen, Carsten Kern, Martin Leucker: Replaying Play in and Play out: Synthesis of Design Models from Scenarios by Learning

2006-13 Wong Karianto, Christof Löding: Unranked Tree Automata with Sibling Equalities and Disequalities

2006-14 Danilo Beuche, Andreas Birk, Heinrich Dreier, Andreas Fleischmann, Heidi Galle, Gerald Heller, Dirk Janzen, Isabel John, Ramin Tavakoli Kolagari, Thomas von der Maßen, Andreas Wolfram: Report of the GI Work Group "Requirements Management Tools for Product Line Engineering"

2006-15 Sebastian Ullrich, Jakob T. Valvoda, Torsten Kuhlen: Utilizing optical sensors from mice for new input devices

2006-16 Rafael Ballagas, Jan Borchers: Selexels: a Conceptual Framework for Pointing Devices with Low Expressiveness

2006-17 Eric Lee, Henning Kiel, Jan Borchers: Scrolling Through Time: Improving Interfaces for Searching and Navigating Continuous Audio Timelines

2007-01 * Fachgruppe Informatik: Jahresbericht 2006

2007-02 Carsten Fuhs, Jürgen Giesl, Aart Middeldorp, Peter Schneider-Kamp, René Thiemann, and Harald Zankl: SAT Solving for Termination Analysis with Polynomial Interpretations

2007-03 Jürgen Giesl, René Thiemann, Stephan Swiderski, and Peter Schneider-Kamp: Proving Termination by Bounded Increase

2007-04 Jan Buchholz, Eric Lee, Jonathan Klein, Jan Borchers: coJIVE: A System to Support Collaborative Jazz Improvisation

2007-05 Uwe Naumann: On Optimal DAG Reversal

2007-06 Joost-Pieter Katoen, Thomas Noll, and Stefan Rieger: Verifying Concurrent List-Manipulating Programs by LTL Model Checking

* These reports are only available as a printed version.

Please contact `biblio@informatik.rwth-aachen.de` to obtain copies.