RWTH Aachen

Department of Computer Science
Technical Report

The MeDUSA Reference Manual, Second Edition

Alexander Nyßen and Horst Lichter

ISSN 0935–3232 · Aachener Informatik Berichte · AIB-2008-07

RWTH Aachen · Department of Computer Science · July 2008

The publications of the Department of Computer Science of RWTH Aachen University are in general accessible through the World Wide Web.

http://aib.informatik.rwth-aachen.de/

MeDUSA

MethoD for UML2-based Construction of Embedded & Real-Time SoftwAre

Version 2.0

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 Construction of Embedded & Real-Time Software) is a model-based software construction method targeting the domain of small embedded & real-time software. MeDUSA was developed by the Research Group Software Construction of the RWTH Aachen University in close cooperation with the German ABB Corporate Research Centre in Ladenburg. It incorporates various practical experiences gained during the industrial development of embedded software in ABB Business Unit Instrumentation.

Being Use Case-driven, MeDUSA systematically covers the software construction lifecycle phase from the early requirements up to implementation. Models are successively developed and employed throughout all activities. By enforcing a class-based rather than an object-oriented design (compare classification according to [Weg87]), a smooth transition of the resulting 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 domain of small embedded & real-time software.

Table of Contents

List of Figures

1 Introduction

1.1 Changes from First Edition to Second Edition

Having been initially published in May 2007 [NL07a], the method has since undergone several changes to incorporate further experiences and lessons learned. As this did not only comprise minor bug fixes and corrections, but indeed affected the overall design of the method, it thus seems to be a natural approach to publish a second revision of the method's reference document at this point.

The first obvious change to the method is a change in the interpretation of the acronym MeDUSA. From its initial meaning *Method for UML2-based Design of Embedded Software Application* it was altered to *MethoD for UML2-based Construction of Embedded & Real-Time Software*. This reinterpretation was done to emphasize mainly two aspects.

The first one is a change in the denomination of the method as a construction method rather than a design method, to indicate that now, all constructive activities of the software development lifecycle are indeed covered (and not only the design related ones). While a seamless transition of the detailed design to source code was already formulated as a central goal for the first edition of MeDUSA, this was actually not intensively reflected by the method. In its second revision, MeDUSA was therefore enhanced by an *Implementation* discipline and a respective phase to explicitly reflect this.

The second is a reformulation of the target domain from *Embedded Software Applications* into *Embedded & Real-Time Software*, to indicate a broader scope of the method. That is, not only application software, but also system software is regarded to be in the scope of MeDUSA (in fact, in the domain of embedded & real-time software, one often focuses a mixture of both). The extension of *Embedded* into *Embedded & Real-Time* may be first and foremost understood as a clarification of terminology (indeed real-time systems were already addressed before. However, this also emphasizes that the method now explicitly addresses real-time related constraints by performing a continuous real-time analysis throughout the *Requirements*, *Analysis*, and *Architectural Design* phases. The corresponding tasks have been grouped into a new *Real-Time Analysis Discipline* to denote that they are closely related to each other.

Besides these, a major change incorporated into the current method definition is related to the detailed design of subsystems and the consolidation of active objects (tasks). While version 1.0 of the method subsumed the consolidation of passive (analysis) objects under the *Detailed Subsystem Design* task, being performed by the *Subsystem Designer* during the *Detailed Design Modeling* phase, and thus split it conceptually from the consolidation of active objects, which was performed by the *System Architect* as part of *Task Modeling*, these two tasks are now merged into a single *Subsystem Consolidation* task, performed by the *System Architect* during *Architectural Design Modeling*.

There are two major reasons for this. The first is that the very late consolidation of passive objects - while somehow emphasizing the distributed aspect of the method - seemed to be overall unpractical and overloaded. In fact, most of the consolidation process is done successively after having identified the subsystems during *Subsystem Identification*, and before modeling the overall system architecture in terms of *Struc-*

tural System Architecture Modeling and *Behavioral Architecture Modeling*. The second reason is related to *Task Modeling*, which comprised not only the consolidation of active objects, but also a real-time analysis. As these two aspects should logically be split into two tasks, and because the consolidation of active objects after the conception of the overall system architecture would always cause a revision of the architecture, merging the consolidation of active and passive objects together seemed to be logical (the former splitting seemed to be artificial).

Besides these major novelties, the current revision of course incorporates several bug fixes, corrections, and improvements in its text and figures. This also comprises some research results, gathered since the first publication of the method, that we incorporated into the method, in particular related to the modeling of use cases in the context of embedded & real-time systems [NL07b] as well as on modeling narrative use case descriptions [WNHL08].

Additionally, the report now also covers a detailed specification of the structure of the employed MeDUSA UML models (instances), an outline of MeDUSA UML profiles, which cover MeDUSA's taxonomies, as well as a code generation schema to generate ANSI-C code from a MeDUSA *Design UML-Model* (compare Appendixes A to C).

Last, the new method definition has been updated to be based on the current SPEM 2.0 Beta Specification [OMG07a]. It also has been translated to American English, as this seems to be more of a standard, internationally seen.

1.2 Characterization of the Application Area

Regarding its applicability, the domain covered by MeDUSA can be characterized as software development of small embedded & real-time software. 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 application area, MeDUSA was initially developed for, namely that of software development for field devices as they can be found in the industrial automation.

Field devices are rather small embedded real-time 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 category of field devices, occur in different product variants concerning the physical quantity they measure (temperature, pressure, flow), the measurement principle applied, the communication capabilities offered, as well as the safety and reliability constraints accomplished.

1.3 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,

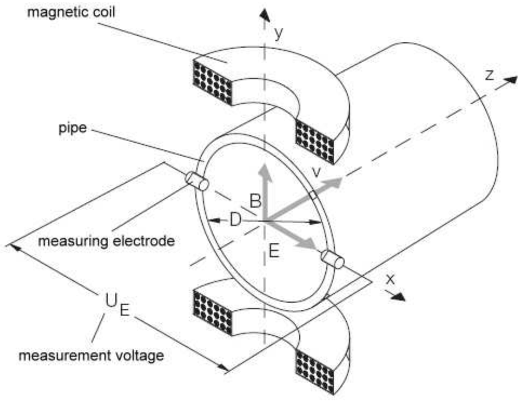


Fig. 1: Physical measurement principle of an electromagnetic flow meter (taken from [GHH+04])

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 1. 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.

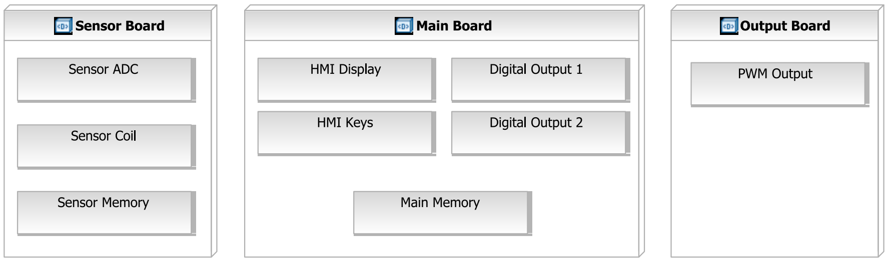


Fig. 2: MeDUSA example device's hardware

From a hardware viewpoint, the example measurement device was designed to be split into three distinct boards, as shown in Figure 2. 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 measurement 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 of transferring 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 electric current, which is generated by a PWM (pulse width modulation). The 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 distinct 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*.

1.4 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 the procedural C-language 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 [1].

Besides those strong technical constraints that have to be addressed, a second basic requirement posed on any method targeting the regarded application domain is, to properly address the organizational and economical constraints that are faced in the domain. One aspect related to this 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. Another aspect is that the method has to be practically applicable by the people responsible of software development in the respective domain, who do not always have to be educated software engineers.

Having all that in mind, our intention was to develop a method that fulfils all those requirements. As we had already gained practical experience with the application of the object-oriented COMET method [Gom00] in the domain [NMSL04], we took it as a starting point to develop MeDUSA. Since its initial publication in 2007 [NL07a] however, MeDUSA can be regarded as a fully self-contained method. The changes incorporated into this revision may be taken as another indication on our ambition to make the method best applicable and understandable.

[1] While this was not covered in its initial publication [NL07a], the current revision of MeDUSA documented herein was enriched by an *Implementation* discipline and phase respectively, thus documenting that MeDUSA is not a mere design method but indeed has to be understood as a construction method

1.5 Characteristics of MeDUSA

Acting on the maxim that model-based software development for small embedded & real-time systems should allow a seamless transition from the design model to an implementation in the C-language, MeDUSA is - unlike COMET - designed to be a class-based rather than an object-oriented method. That is, the application of object-oriented concepts as inheritance and polymorphism is not enforced during all steps of the method. Therefore those concepts may even be omitted during detailed design to allow a straightforward implementation of the detailed design model in a procedural implementation language as C.

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. To offer the flexibility and customizability, which is not least needed, because MeDUSA is embedded into a larger system engineering process and thus has to be aligned with hardware development, the method was furthermore designed to be iterative.

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 [NLS+05]. 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.

1.6 Applied Notation - SPEM 2.0

The MeDUSA method definition is structured according to the current SPEM 2.0 Beta Specification [OMG07a], which is OMG's upcoming standard for method/process definitions. The main characteristic of SPEM 2.0 is the division of a method/process definition into so called *Method Content* and *Process*, as illustrated by Figure 3.

The *Method Content* defines the static content of a method or process, basically in terms of *Task Definitions*, *Role Definitions*, and *Work Product Definitions*. The relationships between those content elements are defined as depicted by Figure 4, i.e. 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 be identified as the primary performer, a role might perform multiple tasks. The responsibility for a single work product however lies within a single role.

Grouping of related *Method Content* elements is supported by so called *Categories*, which are referred to as *Disciplines*, *Role Sets*, or *Domains*, dependent on whether they group related *Task Definitions*, *Role Definitions*, or *Work Product Definitions*.

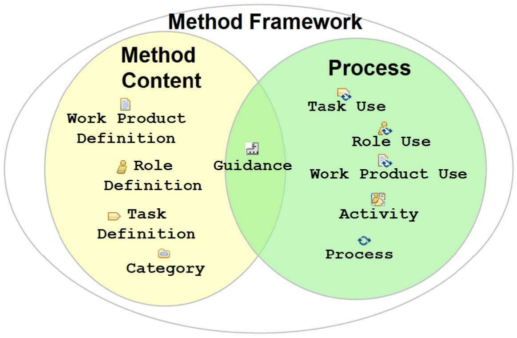


Fig. 3: SPEM Terminology Overview (taken from [OMG07a])

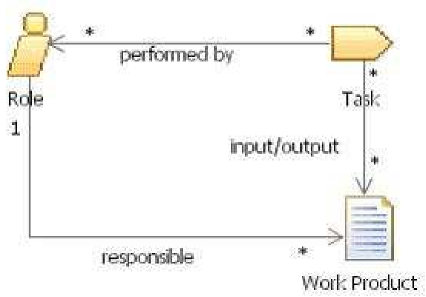


Fig. 4: SPEM method content concepts (based on [Hau06])

The *Method Content* does however not specify how the defined tasks are executed over time. This is defined by the *Process*, as depicted by Figure 5, in terms of a breakdown structure, consisting of *Task Uses*, referencing the *Task Definitions* of the *Method Content*, and *Activities*, which may be arbitrarily nested, so that hierarchical structures can be build.

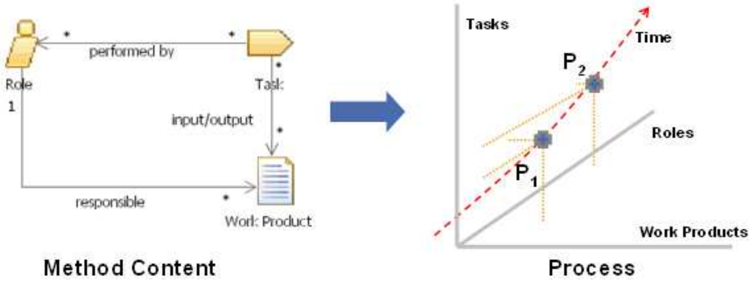


Fig. 5: UMA process concepts (copied from [Hau06])

SPEM supports *Activities* of different kinds, namely *Iterations*, and *Phases*, as well as *Process Patterns* and *Delivery Processes*. While *Process Patterns* are reusable process building blocks, which can be used to build up more complex structures, a *Delivery Process* is understood to be a full end-to-end lifecycle process.

SPEM further offers to specify *Guidances*, which play some hybrid role, belonging neither directly to the *Method Content* nor to *Process*, as indicated by their exposed representation in Figure 3. *Guidances* of several different kinds are supported by SPEM, namely *Templates*, *Estimation Considerations*, *Examples*, *Checklists*, *Guidelines*, *Concepts*, *Estimates*, *Practices*, *Term Definitions*, *Reports*, *Tool Mentors*, *Supporting Materials*, and *Whitepaper*.

Table 1 lists the important SPEM elements, being used in this report, together with their iconified representations.

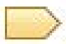
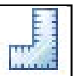
Method Content		**Process**	
Role Definition		Role Use	
Task Definition		Task Use	
Work Product Definition		Work Product Use	
		Iteration	
		Phase	
		Process Pattern	
Guidance		Delivery Process	

Table 1: SPEM Terminological Legend

1.7 Outline

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 the covered task definitions into six *Disciplines* (i.e. categories). Adjacent, chapter 3 defines the *Process* in terms of five *Process Patterns* (referred to as *MeDUSA Workflow Patterns* in the following), which specify how the *Task Definitions* of the *Method Content* are executed in the different constructive lifecycle phases of development. The *MeDUSA Workflow* is then defined in terms of a *Delivery Process* consisting of five *Phases*, where each *Phase* specifies a set of *Iterations* of the respective *Workflow Pattern* for the respective lifecylce phase, as well as re-iterations of earlier ones. Chapter 4 provides a short summary and conclusion.

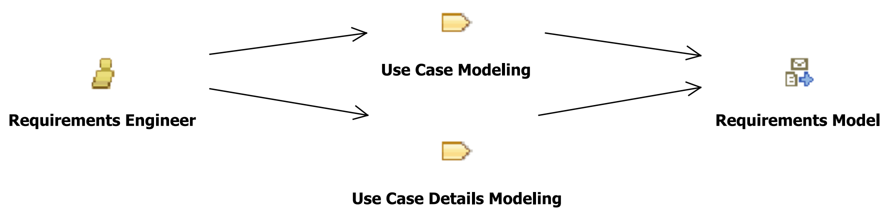
2 Method Content

The MeDUSA *Method Content* is basically defined in terms of six disciplines, namely *Requirements Modeling*, *Analysis Modeling*, *Architectural Design Modeling*, *Detailed Design Modeling*, *Implementation*, as well as *Real-Time Analysis*, which group strongly related *Task Definitions*. They are briefly introduced in the following, giving an initial impression on the covered *Task Definitions* together with related *Role Definitions*, serving as task performers. For each discipline, all subsumed *Task Definitions* are then subsequently described in detail, covering their related (i.e. produced) *Work Product Definitions* and specific supporting *Guidelines*. While SPEM supports a multiplicity of different guidance kinds, for the sake of simplicity, only *Guidelines* will be denoted here. All other *Guidances* may be inferred from the electronic method definition provided on the MeDUSA project site [MeD].

It has to be pointed out that, as MeDUSA is characterized to be based on the UML notation, most *Task Definitions* are defined to produce UML-related artifacts, which are first and foremost UML diagrams. However, it has to be clear that a UML diagram can only be consistently defined if being related to a consistent underlying UML model. And while the diagrams produced by the different *Task Definition* of a discipline are often not directly related to each other, the underlying UML models indeed are. In fact, all *Task Definitions* grouped into a respective discipline are understood to contribute a fraction to an overall integrated model, which is being shared between the respective tasks, to guarantee that all contributed modeling artifacts are consistent to each other. The work products of a task are therefore not only defined in terms of the UML diagrams being developed, but in terms of the model fraction being implicitly contributed to the overall integrated model. To refer to the entirety of all work products being produced by the tasks of the five constructive disciplines (i.e. *Requirements Modeling*, *Analysis Modeling*, *Architectural Design Modeling*, *Detailed Design Modeling*, and *Implementation*) the terms *Requirements Model*, *Analysis Model*, *Design Model*, and *Implementation Model* will be used respectively [2], including UML diagrams, underlying UML model, and other models and documents respectively.

Before we give detailed information on the defined disciplines in the following chapters, we will briefly sketch their purpose and structure in the following.

Requirements Modeling discipline

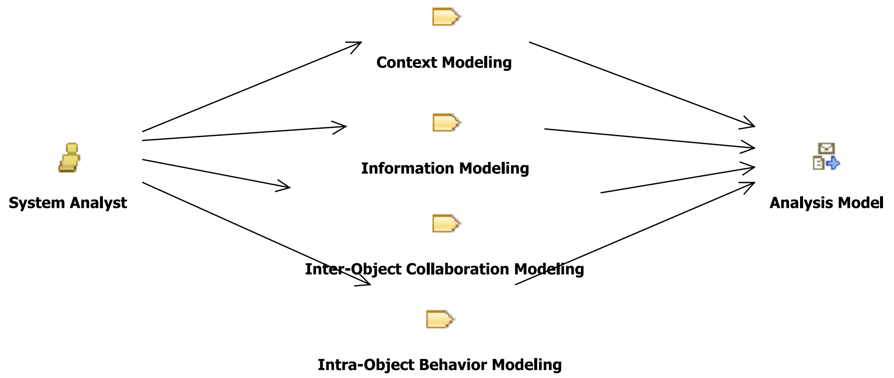


The *Requirements Modeling* discipline is concerned with understanding and capturing the functional requirements of the system under development, as well as the non-functional timing and concurrency concerns that constrain them. As MeDUSA is use

[2] indeed the *Design Model is shared between the Architectural Design Modeling and Detailed Design Modeling disciplines*

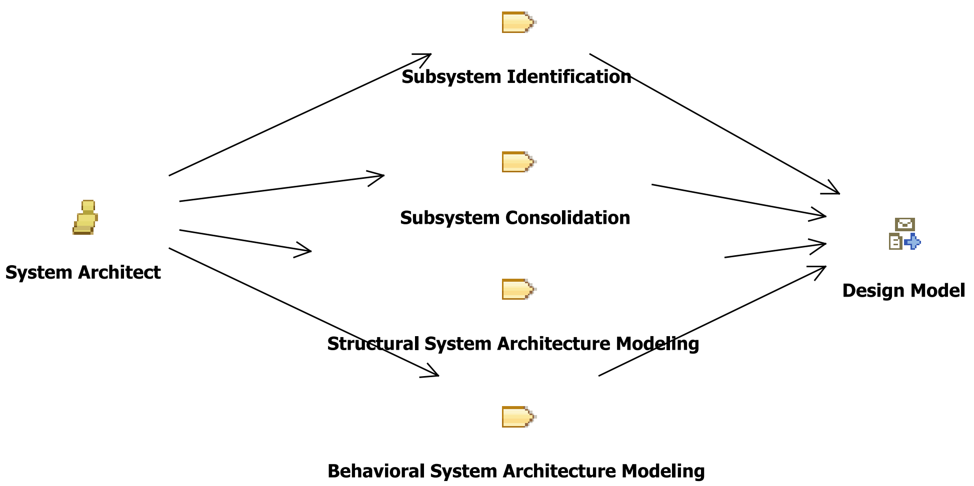
case-driven, the *Requirements Model* is established in form of a UML use case model as well as a narrative model, specifying detailed narrative descriptions for the identified use cases. Both tasks of the *Requirements Modeling* discipline are performed by the *Requirements Engineer*, which is also responsible for all work products produced.

Analysis Modeling discipline



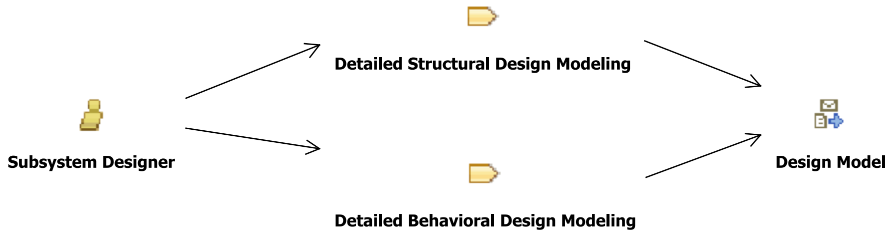
The *Analysis Modeling* discipline deals with understanding the problem domain. That is, it is modeled in terms of analysis objects, who collaboratively perform the scenarios, being subsumed by the use cases captured in the *Requirements Model*. All tasks comprised by the *Analysis Modeling* discipline are performed by the *System Analyst*, who is also responsible for the produced work products.

Architectural Design Modeling discipline



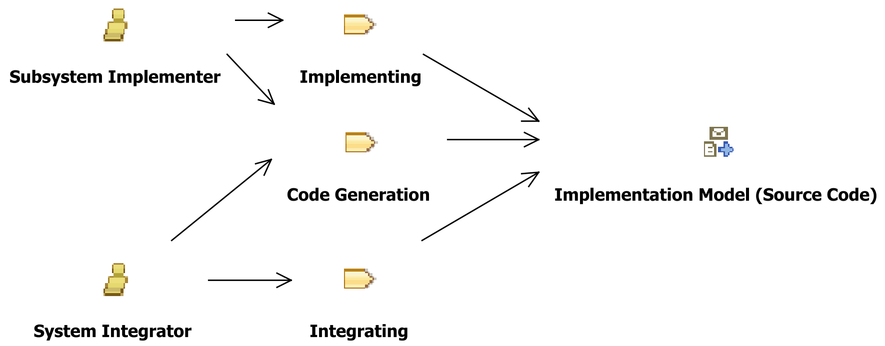
The *Architectural Design Modeling* discipline is concerned with the specification of the software architecture. That is, based on the analysis objects captured in the *Analysis Model*, a system architecture is defined in terms of subsystems, which are gained by grouping together (analysis) objects and by subsequently consolidating the resultant initial set of subsystems under design considerations. The system architecture is not only defined from a structural viewpoint, specifying how the subsystems are structurally interconnected via their visible interfaces, but also from a behavioral viewpoint, specifying the inter-subsystem communication. All tasks in the *Architectural Design Modeling* discipline are performed by the *System Architect*, who is also responsible for all work products being produced.

Detailed Design Modeling discipline



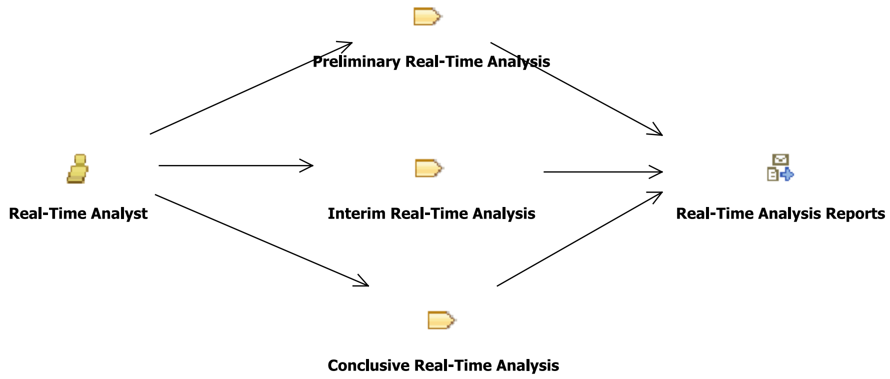
The *Detailed Design Modeling* discipline is concerned with developing the detailed design for the internal decomposition of each identified subsystem. While the externally visible interfaces of all subsystems (together with the involved data types) are defined already as part of the software architecture, a detailed structural and behavioral design has to be developed for those objects, being internally contained by the subsystems. *Detailed Design Modeling* is thus performed individually for each subsystem by a responsible *Subsystem Designer*.

Implementation discipline



The *Implementation* discipline groups those tasks that perform the transformation of the detailed design into source code and of supplementing it with all needed code details to gain a complete implementation model (i.e. source code). The tasks comprised by the *Implementation* discipline are performed by the *Subsystem Implementer* and the *System Integrator*, dependent on whether the code is related to a single subsystem or whether the integration of subsystems is concerned.

Real-Time Analysis discipline



As real-time requirements (timing and concurrency) are of outstanding importance, a real-time analysis is continuously performed. All tasks related to such an analysis, which is performed based on the information captured in the *Requirements Model*, *Analysis Model*, or *Design Model* respectively, are subsumed by this discipline. While each analysis gains increased precision and yields more resilient results with respect to the details level of each respective input model, all tasks are basically performed applying the same techniques of real-time scheduling theory. They are thus all performed by a respective *Real-Time Analyst*, that has to be educated accordingly.

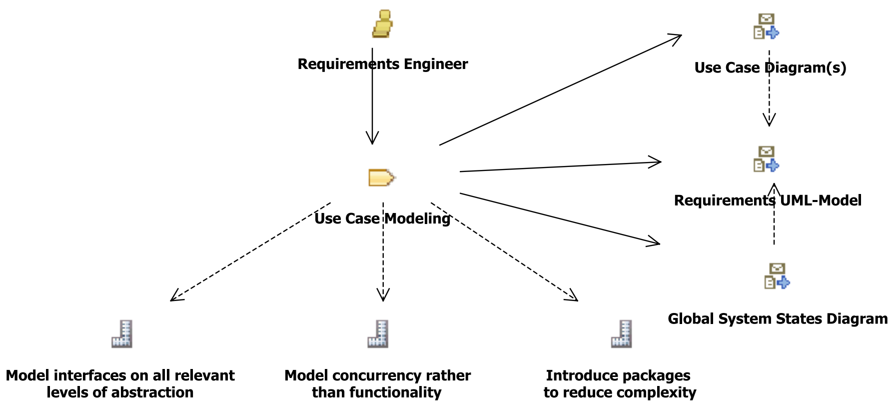
2.1 Requirements Modeling Discipline

The *Requirements Modeling* discipline groups those *Task Definitions* 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 in the context of real-time systems *Requirements Modeling* has to deal with capturing those non-functional constraints as well.

The *Requirements Model* is developed in terms of use cases and detailed (UML or textual) use case descriptions. This is why the *Requirements Modeling* discipline is broken down into the following tasks:

1. *Use Case Modeling*: 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 fulfill those.
2. *Use Case Details Modeling*: Document the details of each use case by using additional behavior diagrams or by describing them in narrative use case descriptions, to capture the detailed flow of events of each use case and to document pre- and post-conditions as well as other valuable information.

2.1.1 Use Case Modeling



Use Case Modeling deals with the development of a use case model in terms of actors, use cases, and their relationships. Use cases describe sequences of interaction between the software system under development and the specified 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 modeling 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 modeling.

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 modeling.

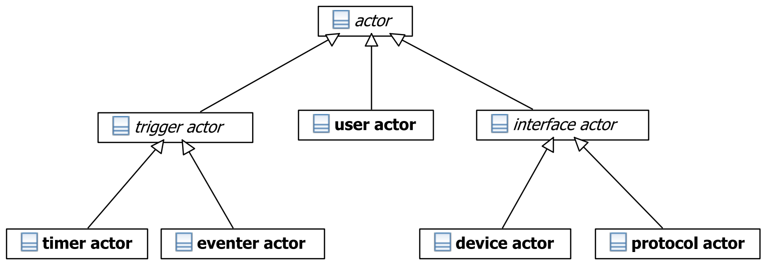


Fig. 6: MeDUSA Actor Taxonomy

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. These 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 interface actors to be secondary ones, thus uncoupling any timing and concurrency concerns from them.

The complete *MeDUSA Actor Taxonomy* is defined as shown in Figure 6. According to this besides the classification of trigger actors into timer and eventer actors, which was motivated before, interface actors are also further divided 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 aperiodic 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.

A user actor may be used to depict that a human user is the communication partner of the device. However, as a human user never directly interacts with an embedded software system, but only indirectly via another software system or hardware device, such a user actor will not be directly attached to a use case but will specify a dependency to the respective trigger or interface actor, which serves as direct communication partner of the system (compare [NL07b] for a detailed discussion on this).

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, as detailedly outlined in [NL07b].

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. After having identified the actors, the next step is to regard what behavior 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 with the primary actors. Analyzing those use cases in terms of similar interaction sequences might lead to the identification of new use cases encapsulating that behavior. 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 modeling task are captured in a use case diagram. An example is shown in Figure 7.

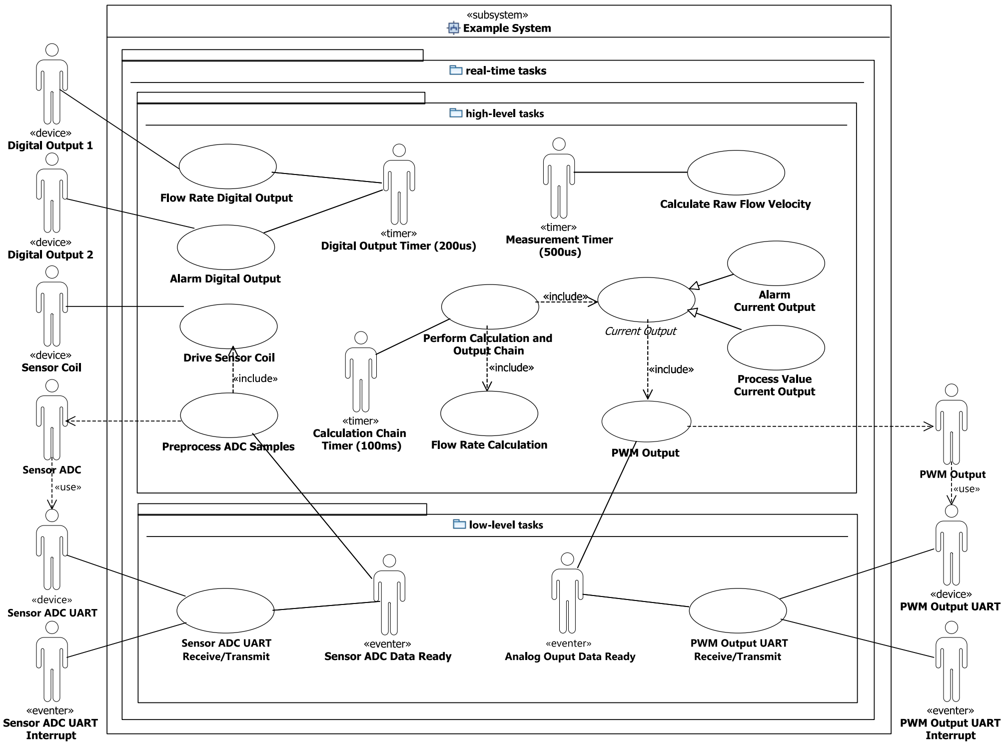


Fig. 7: Example: Use Case Diagram (excerpt)

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 eventer actors
- external (hardware) device, (software) protocol, or (human) user actors,
- relationships between use cases (generalization, include, extend),
- relationships between actors (generalization), and
- relationships between use cases and actors (associations)

– *Global System States Diagram* In case the system shows global system states (e.g. different operation modes), it is helpful to capture them explicitly by means of a *Global System States Diagram* rather than implicitly within the use cases and their detailed descriptions. A *Global System States Diagram* is realized as a UML state machine diagram. An example is depicted by Figure 8.

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

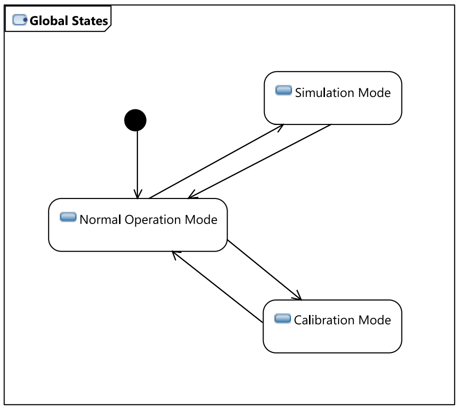


Fig. 8: Example: Global System State Diagram

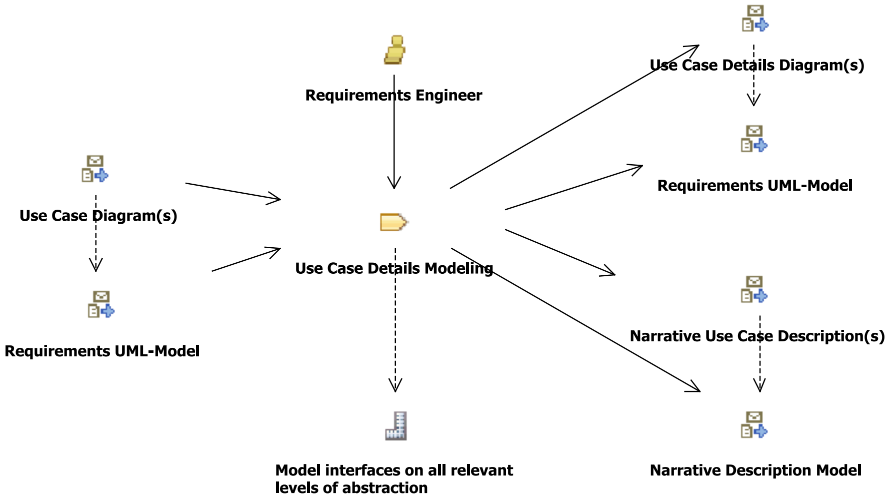
use cases. This might also be helpful in use case models having a smaller amount of use cases, to group use cases according to certain aspects, e.g. if they can be accounted among the real-time related tasks of the system.

– *Model concurrency rather than functionality*: As timing and concurrency concerns are of outstanding importance, we propose to model them with the help of trigger actors as documented in detail in [NL07b]. As timing and concurrency constraints are thus modeled 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 modeled 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 are of outstanding importance for real-time systems and are thus heavier weighted than the functional dependencies that might be identified.
– *Model interfaces on all relevant level of abstraction* From our lessons learned, a major modeling problem one often has to deal with is that an actor has interfaces to the system on different layers of abstraction (compare [NL07b]). 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 operating 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 the respective operating system communication protocol), or both should be reflected in the use case model. 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 with, 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.

The UML does not provide sufficient support to model such a scenario with interfaces on different levels of abstraction. Our advice to deal with this modeling problem is to use the respective modeling pattern we introduced in [NL07b]. The pattern proposes to model all interfaces as actors, and to group use cases of equal abstraction level into packages. Synchronization events between use cases on different levels of abstraction can be denoted via internal eventer actors. In the example of our analog PWM output protocol, being exchanged over a UART hardware device, the situation could thus be modeled as denoted in the example presented in Figure 7 in terms of the `Analog Output` and `PWM Output UART Receive/Transmit` use cases and the related `Analog Output Data Ready` and respective `PWM Output`, `PWM Output UART`, and `PWM Output UART Interrupt` actors.

2.1.2 Use Case Details Modeling



Based on the use case model, a detailed description of each identified use case 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.

While the UML offers several behavior diagrams (activity, sequence) that can be used to describe the details of a use case, narrative textual descriptions seem to be widely used and accepted. Where a UML behavior diagram is not better suited, we therefore propose to capture the details of a use case in a textual narrative form, as proposed in [Wal07] and [WNHL08].

The respective notation proposed 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. It has been inspired by the notation presented in [BS02], making use of a concept denoted as "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 exceptional behavior or error handling. Inclusion and extension of other use cases is also expressed in terms of dependencies between their respective flows of events. Even generalization between use cases may be transferred into the domain of flows, where generalization is understood in terms of generalization 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 can be easily validated. To gain further understanding and a detailed introduction into the developed notation, we propose to refer to [WNHL08].

WORK PRODUCTS

– *Use Case Details Diagram*: Where appropriate, a UML behavior diagram (sequence, activity) may be employed to document the details of a use case, as exemplarily depicted by Figure 9.

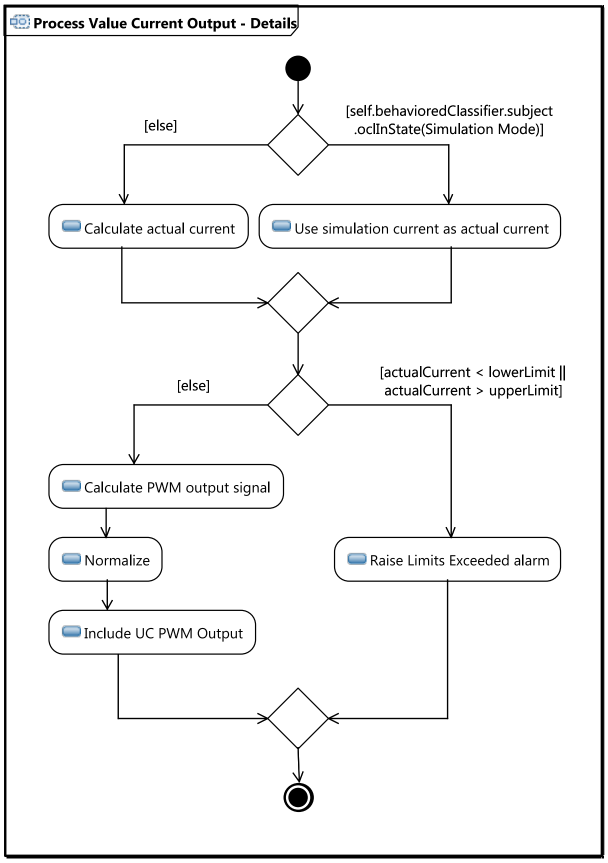


Fig. 9: Example: Use Case Details Diagram

– *Narrative Use Case Description:* Where a UML behavior diagram is not regarded to be better suited, a detailed narrative description should be developed for each use case, as exemplarily shown in Figure 10 for some of the use cases depicted in Figure 7, based on the notation presented in [WNHL08].

GUIDELINES

– *Determine the right granularity:* Finding the right granularity is often difficult when identifying and modeling use cases. If use cases are modeled too fine-grained, a lot of trivial use cases are the result. In such a situation, a lot of associations, include, exclude, and generalization relationships are also modeled in consequence, so that the overall use case model gets rather complex. If use cases are modeled too course-grained, they tend to be internally complex (lots of instructions and alternative branches) what makes their narrative descriptions difficult to handle.

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 `PWM Output`.
9	Alternative Extension Point: End
End	
Alternative Flow Simulate Current	
Start	At Choose Between simulation and calculation, if simulation mode has been set
1	Use simulation current as actual 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 `Process Value 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	Validate that no alarm has been raised
3	Alternative Extension Point : Alarm state validated
4	Include Use Case `Flow Rate Current Output`
5	Alternative Extension Point: End
Alternative Flow Output Alarm.	
Start	At Alarm state validated, if any kind of alarm has been raised
1	Include Use Case `Alarm Current Output`.
End	Continue at End

Fig. 10: Example: Narrative Use Case Descriptions

We propose to consult the narrative use case 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 modeled. 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 modeled too 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 coarse-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 Modeling Discipline

The *Analysis Modeling* discipline is concerned with the development of an *Analysis Model* that helps to understand the problem domain in terms of objects, whose collaborative behavior performs the use cases identified beforehand. Construction of the *Analysis Model* can conceptually be broken down into three main objectives:

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

Identifying objects is of course a quite complicated task that has to be broken down into manageable units to get manageable. MeDUSA addresses the identification of objects successively during three activities of the *Analysis Modeling* 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 11.

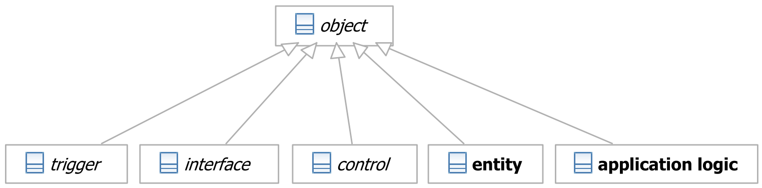


Fig. 11: MeDUSA Object Taxonomy

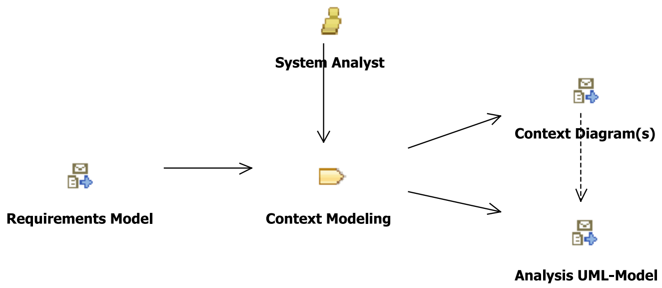
According to it, analysis objects can be classified into *trigger*, *interface*, *entity*, *control*, and *application-logic* objects.

– *trigger objects* represent periodic or aperiodic sources of events, by which any concurrent system behavior is stimulated.
– *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 behavior.
– *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 Modeling* discipline aim at identifying objects of different categories each.

- *Context Modeling* supports the identification of interface and trigger objects by questioning which interfaces from the software system under development towards its external environment have to exist, and to which external event sources the system under development has to respond.
- *Information Modeling* helps to identify entity objects, which are needed to store long-living data that has to be handled by the system.
- *Inter-Object Collaboration Modeling* takes into consideration the use cases identified during *Requirements Modeling*. It supports the identification of objects that make up application-logic or control-flow 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 behavior that results from performing each use case.
- *Intra-Object Behavior Modeling* deals with specifying the intra-object behavior by synthesizing the partial behavior each identified object shows in the collaborations it participates in.

2.2.1 Context Modeling



During *Context Modeling* 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 *Context Diagram* is developed to capture the gathered information in terms of a UML object diagram, which captures the respective trigger and interface objects:

– *Trigger objects* are modeled 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 12 depending 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 directly inferred from the trigger actors captured in the requirements model.

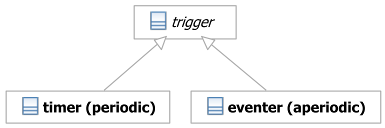


Fig. 12: MeDUSA Trigger Object Taxonomy

– *Interface objects* serve as interaction points for incoming or outgoing communication of the system under development towards its external environment. As shown in Figure 13, interface objects are further categorized into hardware and software interfaces.

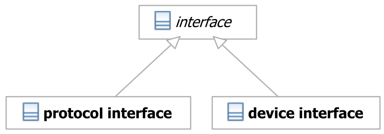


Fig. 13: MeDUSA Interface Object Taxonomy

Identifying interface and trigger objects is done by inferring them from respective actors of the *Requirements Model*. While trigger actors are directly mapped to trigger objects, interface actors will normally 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 modeling are captured in a *Context Diagram*, which is developed in form of a UML object diagram as depicted in Figure 14.

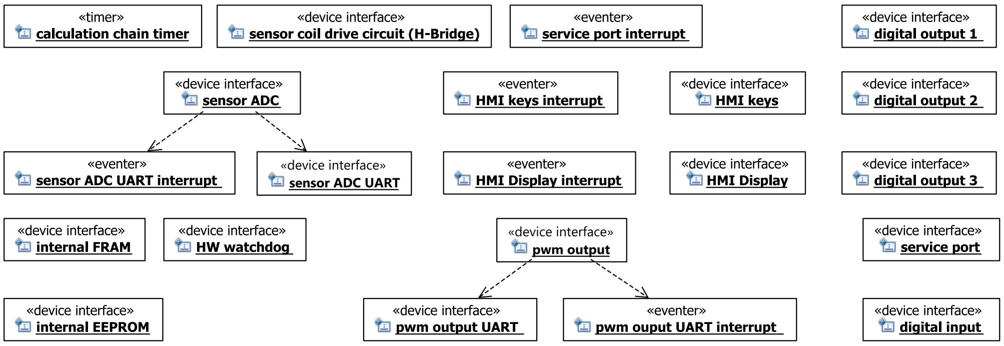
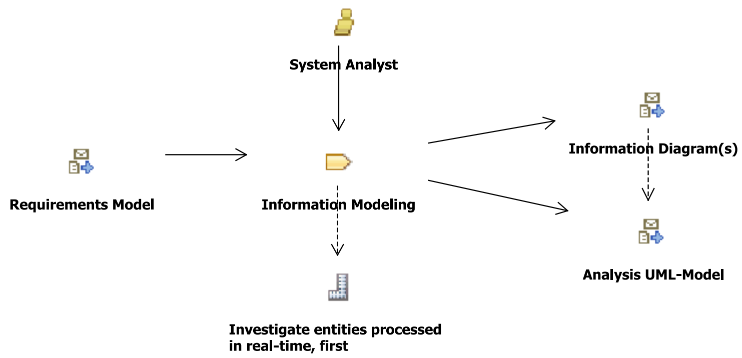


Fig. 14: Example: Context Diagram

It shows the composed 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 13. Trigger objects are stereotyped accordingly as specified by the MeDUSA Trigger Object Taxonomy shown in Figure 12.

2.2.2 Information Modeling



Information Modeling 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. They may represent measured physical quantities, real world objects, abstract concepts or any other data as constraints, configuration, or calibration information.

WORK PRODUCTS

– *Information Diagram:* The results of the *Information Modeling* task should be captured in an *Information Diagram*, which is developed in form of a UML object diagram as shown exemplarily by Figure 15.

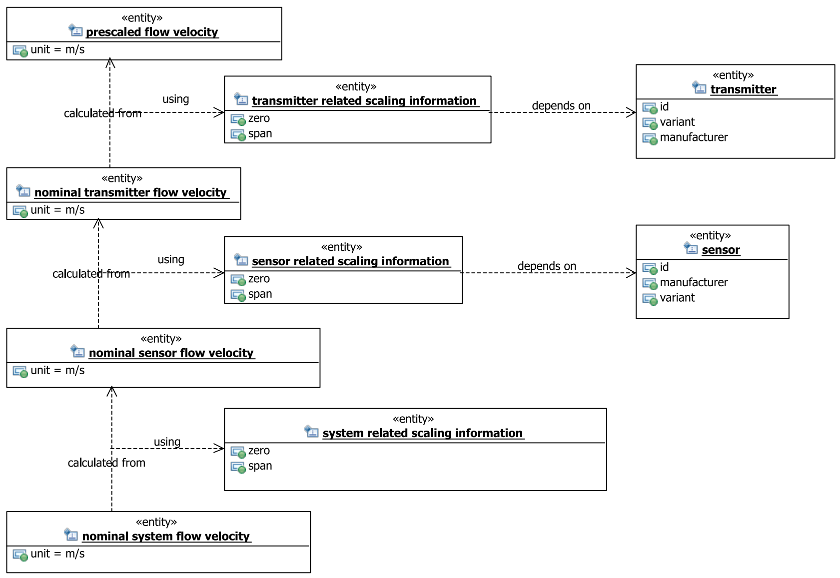


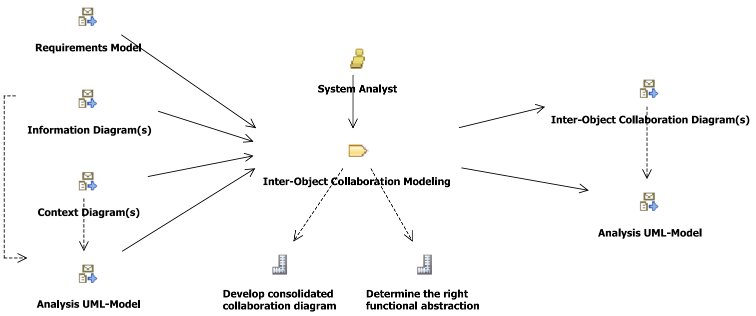
Fig. 15: Example: Information Diagram (excerpt)

It shows the entity objects linked to each other using links or just dependencies (less formal). If supported by the tool, modeling n-ary links or dependencies relating 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 covered by MeDUSA is much focused on value processing, identifying the relevant entity objects is most easily done by first identifying the relevant physical quantities involved in 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, floating through the device, is needed to calculate the mass flow, leading to an entity object called `medium` having a property/slot of name `density`.

2.2.3 Inter-Object Collaboration Modeling



Collaboration of objects now have to be identified, which collaboratively perform the scenarios, subsumed by the identified use cases. As a starting point to identify the respective object collaborations, the objects initiating the execution of the respective scenarios have to be first 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 Modeling* task, as these are the objects that are directly inferred from the actors identified during *Requirements Modeling* (they, together with the interface objects, are indeed the only objects that manifest interactions with the external environment of the software system). In case the use case is included by another use case or it 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 *Context Modeling* and *Information Modeling* 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 logic. As prescribed by the *MeDUSA Control Object Taxonomy* shown in Figure 16, control objects can be further classified into *coordinator* and *state-dependent-control* objects. While state-dependent-

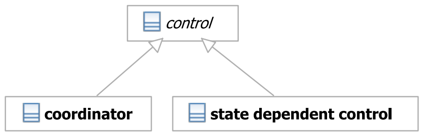


Fig. 16: MeDUSA Control Object Taxonomy

control objects encapsulate state-dependent behavior, coordinator objects encapsulate the non-state-dependent coordination of objects.

Having identified the necessary control and application-logic objects, the main flow of events of the use case can be described in terms of messages between the identified objects. This helps to gain an understanding on how the collaborative interplay of the identified objects performs the main flow. 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 Modeling* has been performed, all necessary objects should be identified and it should be understood how these objects collaboratively work together to perform the use cases identified during *Requirements Modeling*.

WORK PRODUCTS

– *Inter-Object Collaboration Diagram*: The results of *Inter-Object Collaboration Modeling* should be captured in one or more *Inter-Object Collaboration Diagram(s)*. Those *Inter-Object Collaboration Diagrams*, are developed in form of UML communication or sequence diagrams as exemplarily depicted by Figure 17 and 18. The decision whether to use a communication diagram or a sequence diagram to depict the collaborative behavior 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).

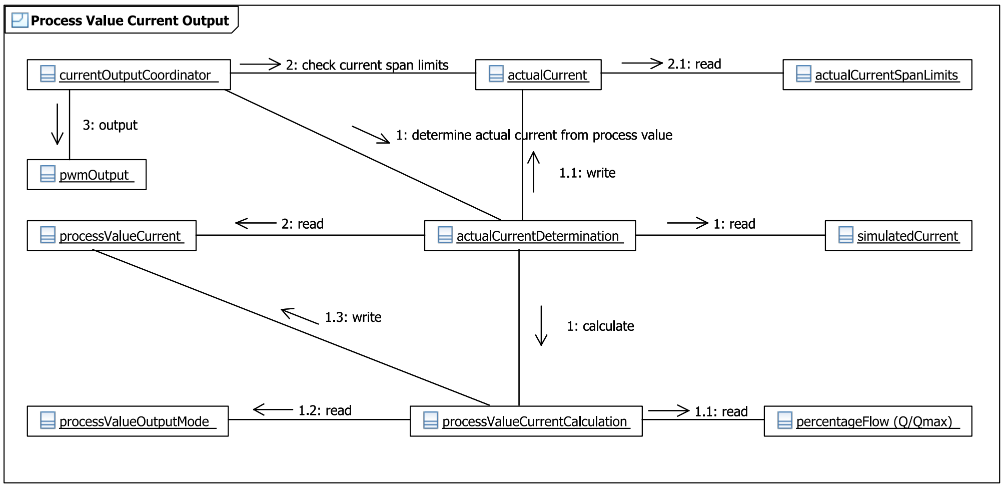


Fig. 17: Example: Inter-Object Collaboration Diagram (Communication)

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 modeled by an additional sequence diagram, as it better supports the modeling of optional or alternative messages by the use of interaction fragments. It may however be reasonable to just have a single communication diagram (if there are no alternative

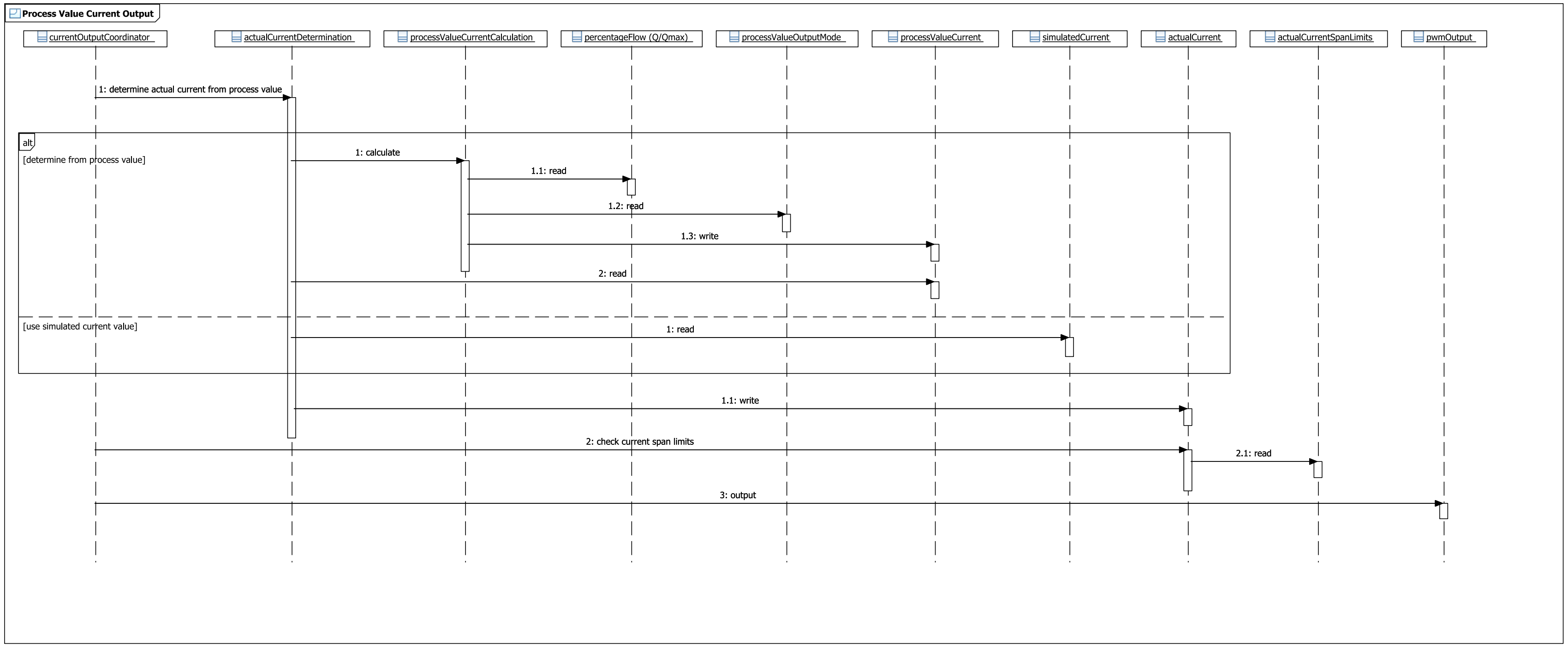


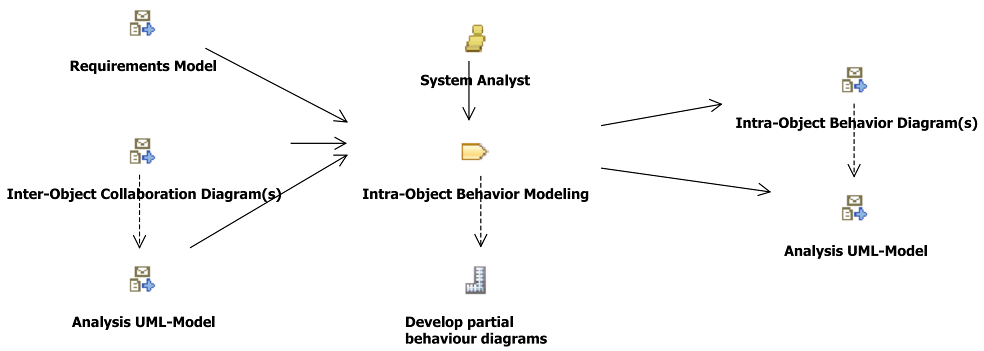
Fig. 18: Example: Inter-Object Collaboration Diagram (Sequence)

flows or if they are trivial) or just a single sequence diagram, if the number of objects is quite low.
It may not always be necessary or reasonable to have *Inter-Object Collaboration Diagrams* for each individual use case identified during *Requirements Modeling*. 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 another use case or if the number of objects or messages grows too large when 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* of the extended one.

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 Modeling* (as an object often participates in more than one collaboration and also a single collaboration is often modeled 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 modeled by an application-logic object, and when it is just a function of an entity object (i.e. it is modeled 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 involved, encapsulating the function or algorithm by an application-logic object is the better choice. If just one entity object is involved it might usually be better to use a simple function in the respective entity object.

2.2.4 Intra-Object Behavior Modeling



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

For all state-dependent control objects, which were identified during modeling of the system collaborations, the state-dependent behavior 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 behavior of that object has to be synthesized from the partial use case based behavior of the object in all collaborations it participates in.

Similar to specifying the behavior of the identified control objects, it may be reasonable to also describe the behavior 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 Behavior Diagram*: As shown exemplarily in Figure 19, the internal behavior of each non-trivial object should be captured by an *Intra-Object Behavior Diagram*.

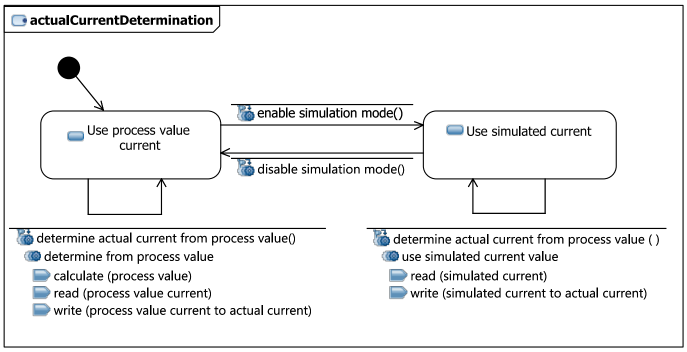


Fig. 19: Example: Intra-Object Behavior Diagram - ActualCurrentDetermination

In case of a state-dependent application object this is done in the form of a state-machine diagram. The internal behavior of application-logic objects will most likely be best captured by using an activity diagram. However, other diagrams may be employed if applicable.

GUIDELINES

- *Develop partial behavior diagrams*: If the synthesizing of the partial state dependent behaviors 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 behaviors of the object in all collaborations first, which can then be used as input for the synthesizing.

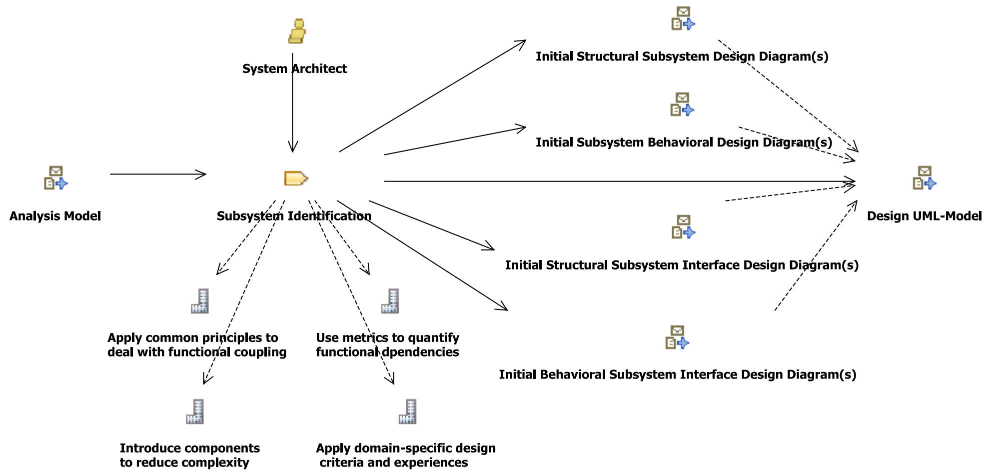
2.3 Architectural Design Modeling Discipline

While the *Analysis Modeling* discipline deals with breaking down the problem domain, *Architectural Design Modeling* can be seen as the first step of composing a solution. The central goal of the tasks comprised by the *Architectural Design Modeling* discipline is to develop the software architecture, which specifies subsystems, as well as the structural and behavioral relationships between them.

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

- *Subsystem 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.
- *Subsystem Consolidation* is done by consolidating the initial subsystem design, which has been gathered by a division of the analysis objects, under design considerations. In particular active objects have to be investigated under performance aspects and have to be clustered together (e.g. two timers can be merged, if having a related period). Passive objects have to be investigated from a design aspect as well (i.e. they may be merged together or split apart, if this would be feasible under design considerations).
- *Structural System Architecture Modeling* is performed to constitute the structural software architecture by specifying the structural relationships between the identified subsystems, established via their required and provided interfaces.
- *Behavioral System Architecture Modeling* is done by regarding, how the system behavior is performed by the collaborative behavior of the identified subsystems. Communication between subsystems can of course only be established via the structural relationships defined previously.

2.3.1 Subsystem Identification



Subsystem Identification is the first *Architectural Design Modeling* task. It is concerned with grouping the identified trigger, interface, entity, control and application-logic (analysis) 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 being targeted, 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 trigger objects 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 subsystems being affected by more than one task.
- Reusability: Another criterion that should be taken into consideration is the reusability of already existing subsystems. Analysis objects might be grouped together so that the comprised 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 communication management or storage management.

After a group of objects has been decided to form a subsystem, the interaction points of the subsystem towards its surrounding environment have to be defined. They can be determined by looking at the behavioral 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 is referred to as a port. Further, by differentiating on whether a message enters or leaves the subsystem, and by grouping respective messages together, required and provided interfaces can be derived, which allow to specify the interaction, established via each port, in detail. Note that all interfaces, provided as well as required, are always described from the viewpoint of each respective subsystem to sustain its self-encapsulation. In particular, required interfaces are always specified from the viewpoint of the *requiring* subsystem, explicitly not referencing any provided interface of another subsystem.

WORK PRODUCTS

– *Initial Structural Subsystem Design Diagram*: The internal structure of each subsystem, which is obtained by grouping together related objects from the analysis model, is captured in a corresponding *Initial Structural Subsystem Design Diagram*. An example is shown in Figure 20.

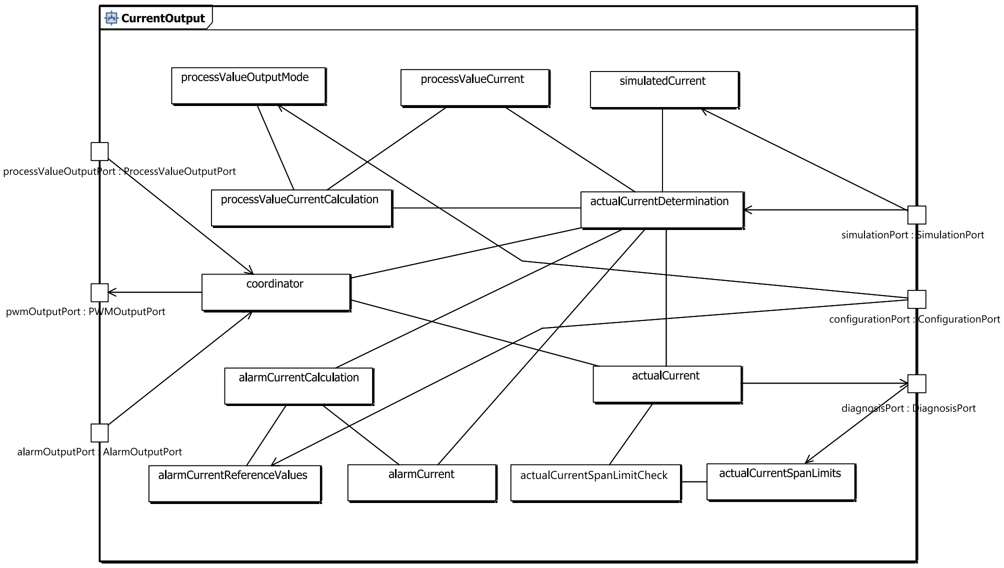


Fig. 20: Example: Initial Structural Subsystem Design Diagram

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 environment. The provided and required interfaces aggregated by each port may be (optionally) 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 decomposition of the subsystem is modeled in terms of parts, representing the trigger, interface, entity, control and application-logic objects partitioned into it. Relationships between the objects are modeled using assembly connectors. Where objects do have relationships to external objects (which are partitioned into other subsystems) delegation connectors can be modeled 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 by inferring the structural relationships from the analysis model, and consolidation of it is not regarded by this task. Indeed such a consolidation of the internal subsystem decomposition is addressed by the *Subsystem Consolidation* task. It therefore may also be decided to now draw a respective *Initial Structural Subsystem Design Diagram* but to directly construct a *Consolidated Structural Subsystem Diagram* by merging the *Subsystem Identification* and *Subsystem Consolidation* tasks together.

– *Initial Structural Subsystem Interface Design Diagram* As the *Initial Structural Subsystem Design Diagram*, which shows the external interaction points only in terms of ports, an initial version of the externally visible provided and required interfaces' signatures has to be defined in an *Initial Structural Subsystem Interface Design Diagram*. Such a diagram 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, and the port types, establishing usage or interface realization relationships to the required and provided interfaces respectively. Note that the port's classes implement all methods of the required as well as the provided interfaces. This is, because they may forward all method calls, either to the internal decomposition of the subsystem, in case of method subsumed by a provided interface, or to the external environment, in case of a required interface.
– *Initial Behavioral Subsystem Design Diagram*: The internal communications inside each subsystem has to be documented in one or more *Initial Behavioral Subsystem Design Diagrams*, as it is exemplarily denoted by Figure 22.
  It is developed in terms of a UML2 sequence diagram showing message-based communication between the ports and parts composed by the subsystem (as modeled in the *Initial Subsystem Design Diagram*). It may be possible to denote several scenarios by a single *Initial Behavioral Subsystem Design Diagram*, using interaction fragments, most often however, several behavioral diagrams will be used for purposes of clarity and readability.
  As in case of the *Initial Structural Subsystem Design Diagram*, the *Initial Behavioral Subsystem Design Diagram* will be consolidated during *Subsystem Consolidation*. Therefore it is possible - as in the latter case - to drop this diagram and directly build a *Consolidated Behavioral Subsystem Design Diagram* if the *Subsystem Identification* and *Subsystem Consolidation* tasks are merged together.
– *Initial Behavioral Subsystem Interface Design Diagram*: To describe the interaction possibilities offered by the subsystem via its ports from a behavioral perspective, thus describing possible dependencies between the offered methods as well

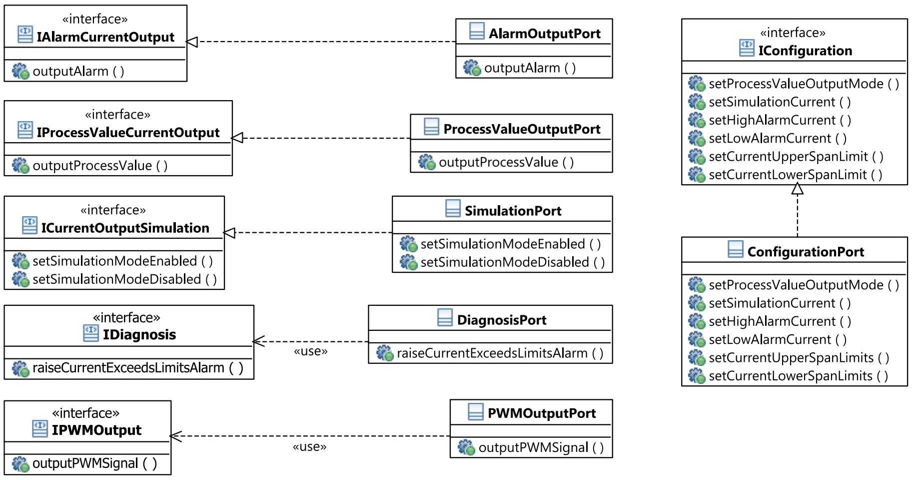


Fig. 21: Example: Initial Structural Subsystem Interface Design Diagram

as pre- and post-conditions, one or more *Initial Behavioral Subsystem Interface Design Diagram(s)* should be developed in the form of a UML state machine diagram (as protocol state machine). Why it is regarded to be mandatory to develop at least one *Initial Behavioral Subsystem Interface Diagram* to show global dependencies between methods offered by the different ports of the subsystem, as exemplarily denoted by Figure 23, as respective diagram may also be developed for an individual port, or even interface.
Here, in case of all former working products developed by this task, the *Initial Behavioral Subsystem Interface Design Diagram* will be consolidated during *Subsystem Consolidation*. Therefore it may as well be an option in this case to drop this diagram and directly build a *Consolidated Behavioral Subsystem Interface Design Diagram* instead.

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 objects (and cohesion of object clusters) during *Subsystem Consolidation*. The number and frequency of

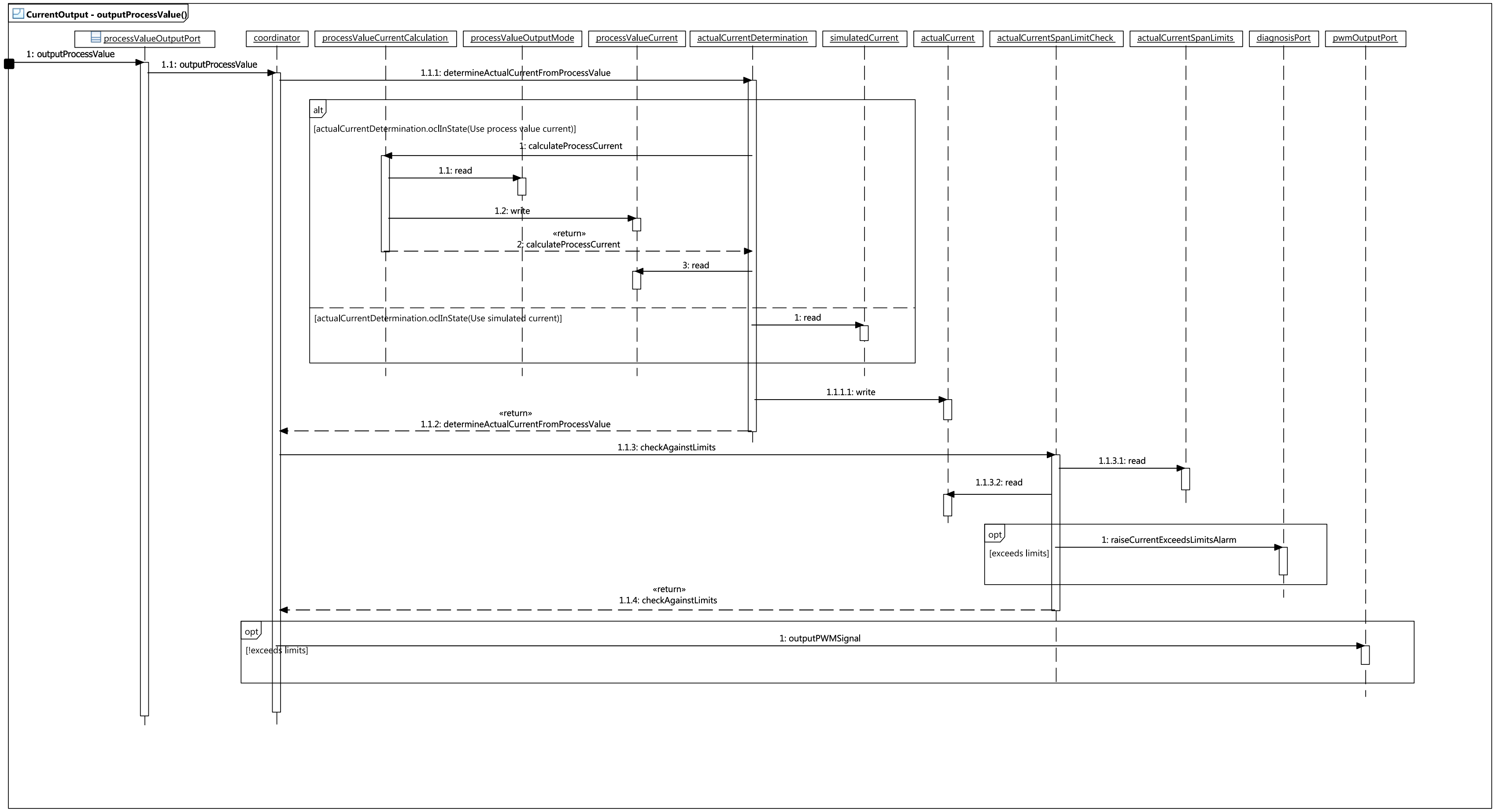


Fig. 22: Example: Initial Behavioral Subsystem Design Diagram

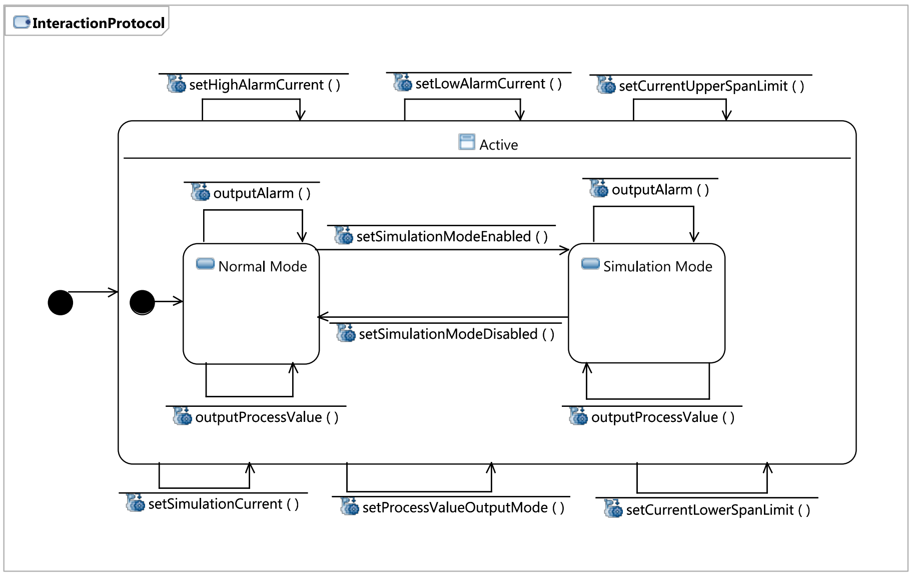
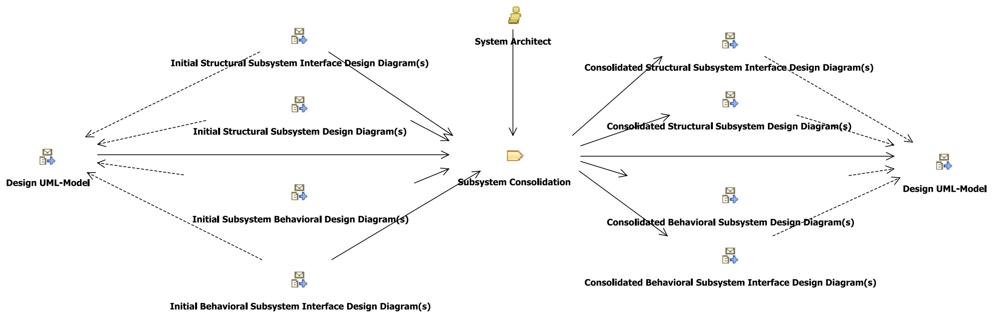


Fig. 23: Example: Initial Behavioral Subsystem Interface Design Diagram

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 predefined 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 not to split it into several smaller 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.

2.3.2 Subsystem Consolidation



After the active and passive analysis objects have been partitioned into subsystems according to certain criteria, the resulting subsystems have to be consolidated under design considerations.

In detail, all active objects have to be evaluated on whether they 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). Eventer objects may be clustered together if the events are indeed not truly concurrent events but occur in a sequential or mutually exclusive manner. We propose to refer to the task clustering criteria described in [Gom00] to get an inspiration on optimization potentials.

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 7 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`, which 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.

The passive objects also have to be consolidated, as they were identified from an analysis viewpoint (they were just partitioned) and not from a design viewpoint. 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.

Last, the interaction points of each subsystem, the ports and their respective provided and required interfaces have to be consolidated (they may be grouped under certain aspects) and the detailed class design of all classes being exchanged as parameters over those interfaces has to be developed.

WORK PRODUCTS

– *Consolidated Structural Subsystem Design Diagram*: A *Consolidated 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.

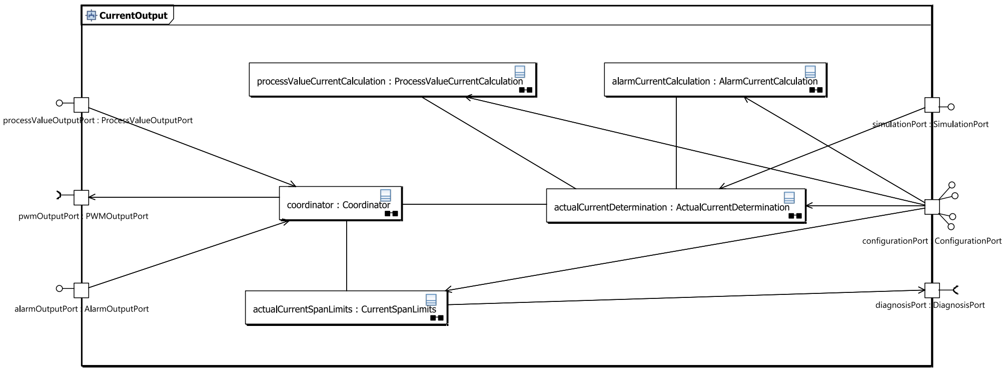


Fig. 24: Example: Consolidated Structural Subsystem Design Diagram

- *Consolidated Structural 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 25. It is an advancement to the *Initial Subsystem Interface Design Diagram*, developed during *Subsystem Identification*, where method parameters and their data types are additionally defined.

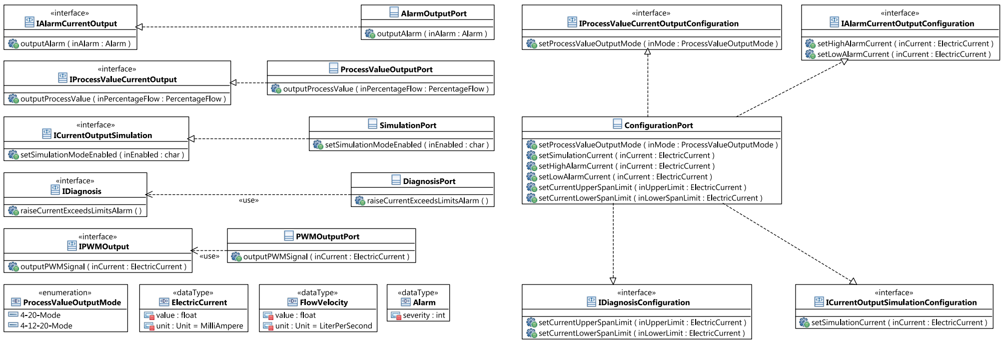


Fig. 25: Example: Consolidated Structural Subsystem Interface Design Diagram

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 occurring in the signatures of the interfaces, 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.

- *Consolidated Behavioral Subsystem Design Diagram*: Changes to the structural subsystem design will imply changes to its behavioral design as well. Therefore, one or more *Consolidated Behavioral Subsystem Design Diagrams* should be developed as advancements to the *Initial Behavioral Subsystem Design Diagrams* created during *Subsystem Identification*. Furthermore, the messages should now be equipped with arguments to denote the parameter objects being passed (while parameter types are not yet regarded). An example for a *Consolidated Behavioral Subsystem Design Diagram* is shown in Figure 26.

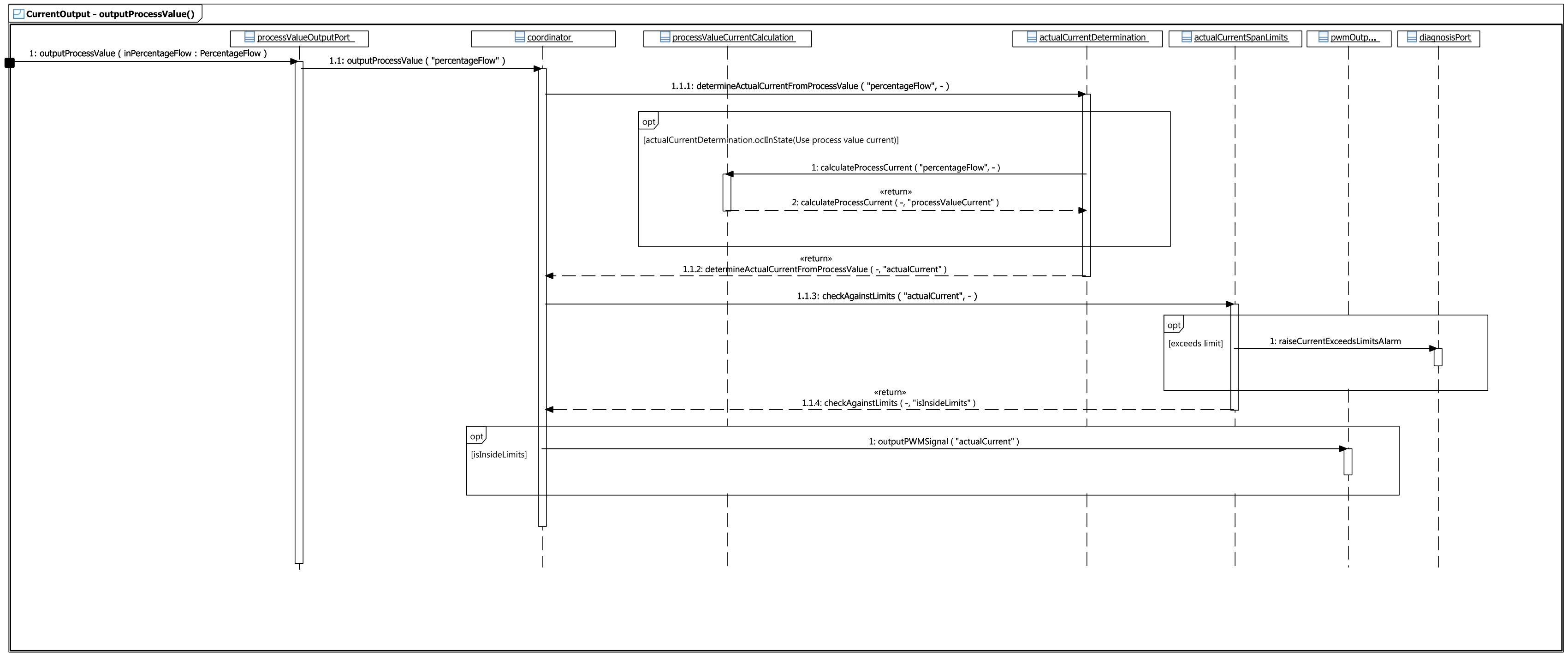


Fig. 26: Example: Consolidated Behavioral Subsystem Design Diagram

– *Consolidated Behavioral Subsystem Interface Design Diagram*: Changes to the structural interface design likely affect the behavioral interface design as well. Therefore, the *Initial Behavioral Subsystem Interface Design Diagram* has to be consolidated to reflect the changes. It also has to be detailed, as now, method signature and guards can be specified on a more fine-grained level than in the *Initial Behavioral Subsystem Interface Design Diagram*. An example for a *Consolidated Behavioral Subsystem Interface Design Diagram* is denoted by Figure 27. It does not show significant changes towards the initial diagram version as the changes did not have great impact. Yet, the detail level was slightly enhanced to reflect the introduced method signatures.

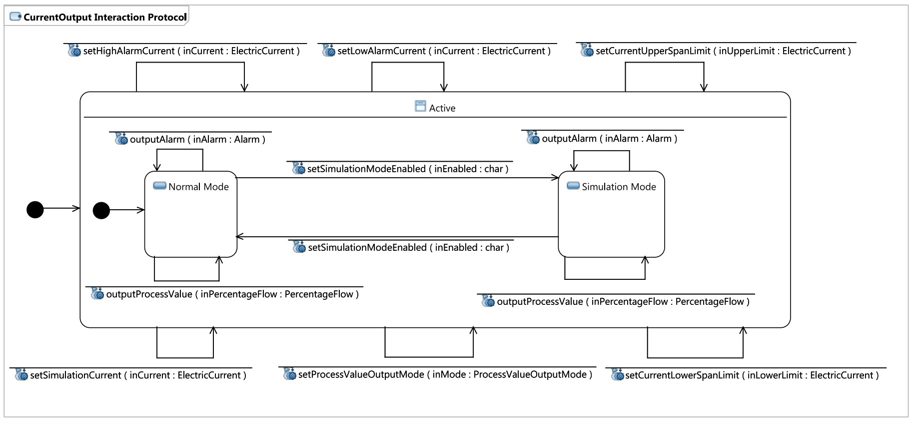
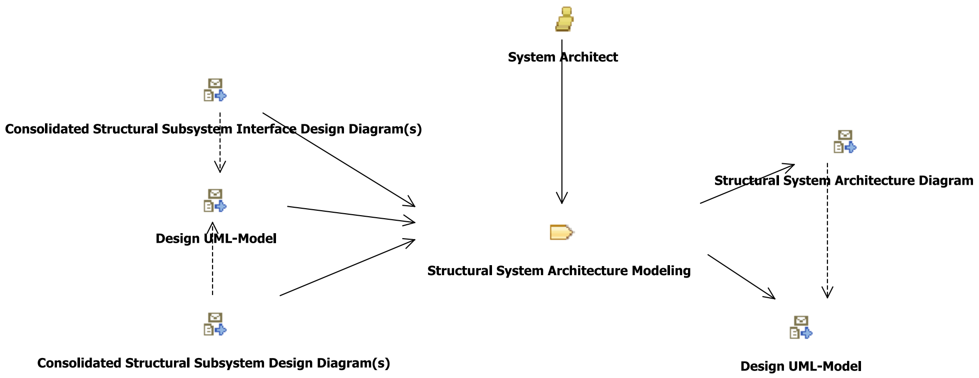


Fig. 27: Example: Consolidated Behavioral Subsystem Interface Design Diagram

2.3.3 Structural System Architecture Modeling



After having identified subsystems by grouping together and consolidating the analysis objects, and after having defined interaction points (ports) with required and provided interfaces, the next step is to describe how the subsystems are structurally related via those interfaces, to denote that the subsystems structurally fit together.

WORK PRODUCTS

– *Structural System Architecture Diagram*: The results of this task should be captured in a *Structural System Architecture Diagram*, which is developed in terms of UML composite structure diagrams, showing the subsystem (instances), their provided and required interfaces and the structural relationships established via those interfaces. An example is shown in Figure 28.

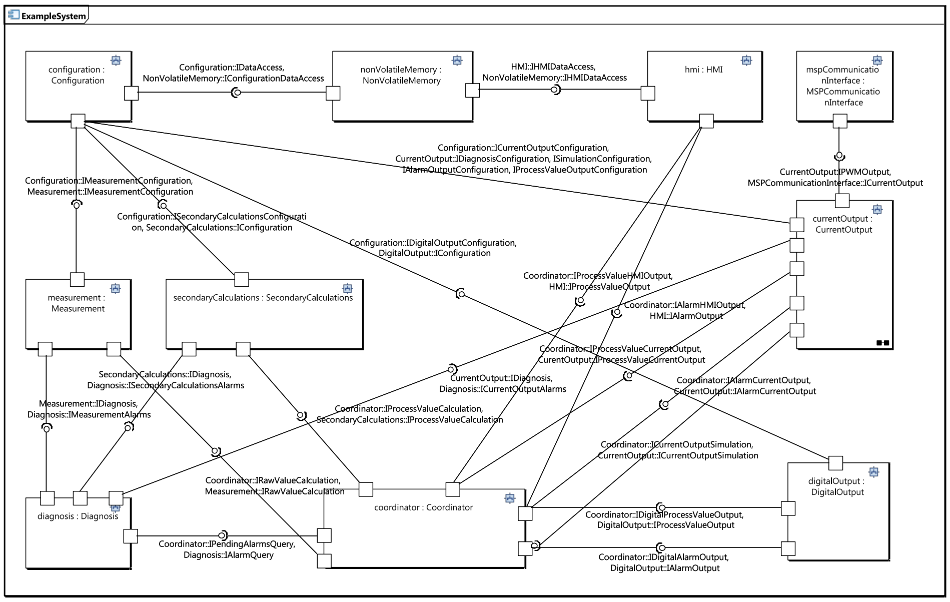
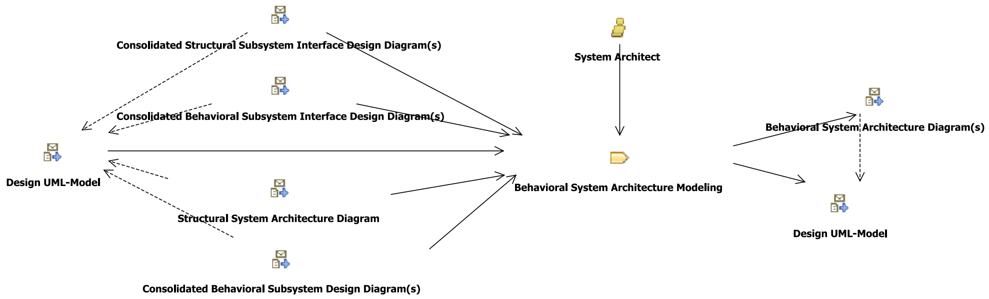


Fig. 28: Example: Structural System Architecture Diagram

2.3.4 Behavioral System Architecture Modeling



After having integrated the subsystems structurally, it is necessary to describe, how system-wide behavior (that is affecting more than one subsystem) does affect the subsystems via their provided/required interfaces. This is done by investigating, how the system behavior is collaboratively performed by the subsystems (i.e. taking all use cases into account, that span more than one subsystem).

WORK PRODUCTS

– *Behavioral System Architecture Diagram*: The results of the *Behavioral System Architecture Modeling* task are captured in a couple of *Behavioral System Architecture Diagrams*, which are realized as UML sequence diagrams. As an example, Figure 29 shows a combined *Behavioral System Architecture Diagram* for the use cases `FlowRateCurrentOutput` and `AlarmCurrentOutput`.

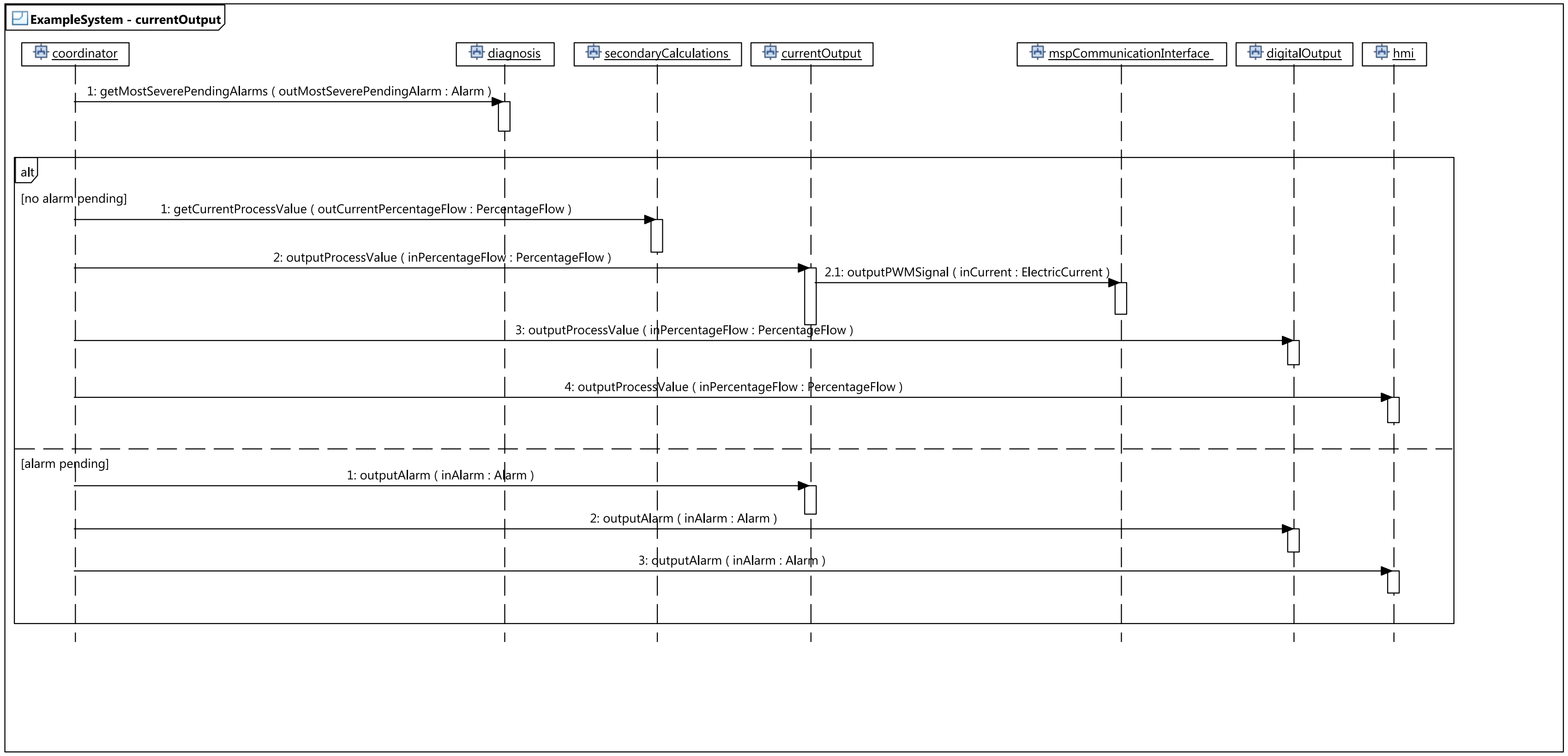


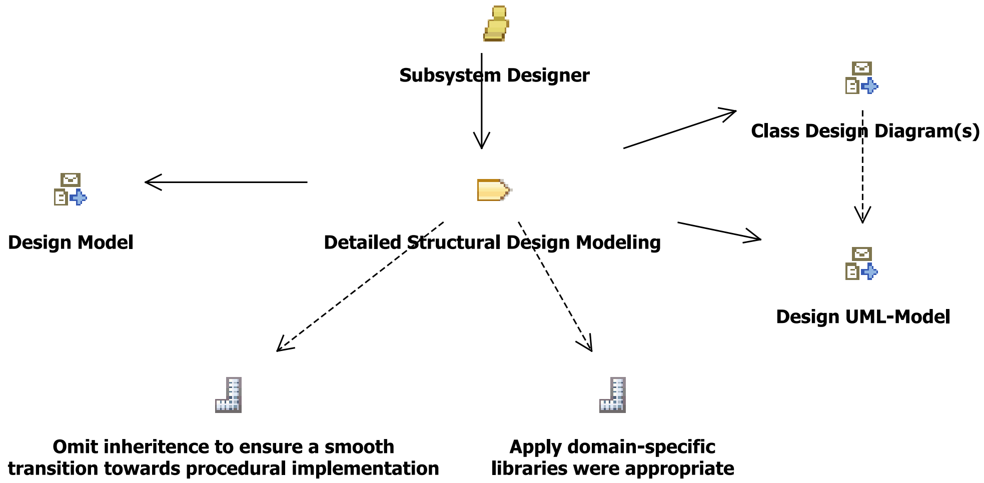
Fig. 29: Example: Behavioral System Architecture Diagram

2.4 Detailed Design Modeling Discipline

While the externally visible interfaces of each subsystem have already been completely defined in terms of ports offering provided and required interfaces (including a detailed class design for all involved parameter types), a detailed design for the internal decomposition of each subsystem has to be developed, so detailed that it can be taken as direct input for the successive 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 prior disciplines are indeed performed within the scope of the overall system, *Detailed Design Modeling* is performed distinctly for each subsystem, thus allowing an independent and potentially concurrent processing of each.

2.4.1 Detailed Structural Design Modeling



After the internal decomposition of the subsystem has been consolidated, the detailed class design for all design objects (modeled as parts) composed by the subsystem has to be developed. That is, classes have to be designed to type each composed part, having attributes corresponding to the slots of the objects and operations corresponding to the messages the object may receive (that is, it has to implement those methods that the external ports forward to it, captured in its provided interfaces, or which it may receive from other composed parts). Where parts are connected to each other by an assembly connector, or to a respective port via a delegation connector, associations have to be designed on the class level, which are used to type the respective connector.

WORK PRODUCTS

- *Structural Detailed Design Diagram*: The results of the *Detailed Structural Design Modeling* should be documented in UML class diagrams. It has to capture all relevant classes and associations, which are used as types for all parts, ports and connectors captured in the *Consolidated Structural Subsystem Design Diagram* respectively. All classes should be modeled with their respective attributes and operations, so that the class design can be taken as a detailed building plan for the succeeding 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 30.

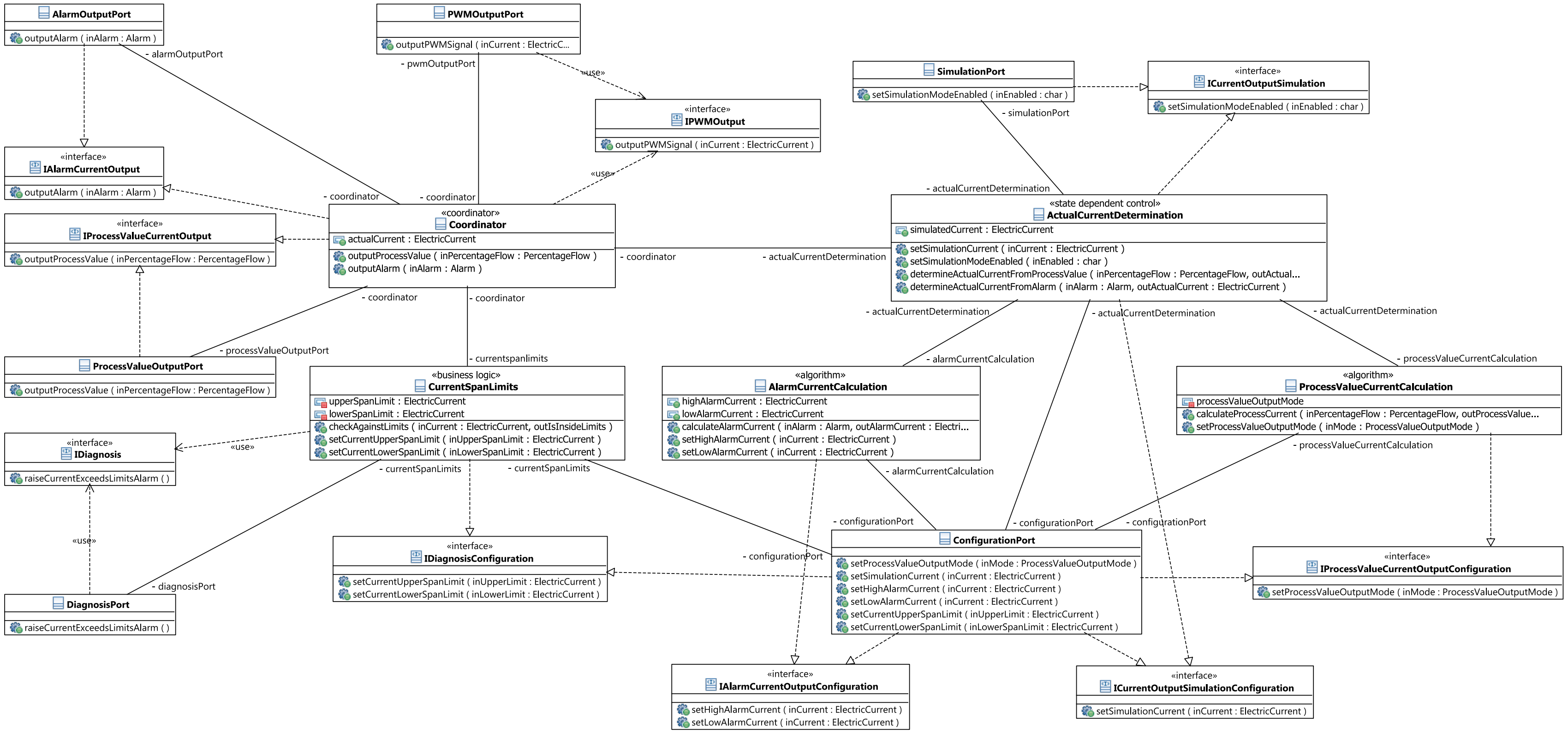
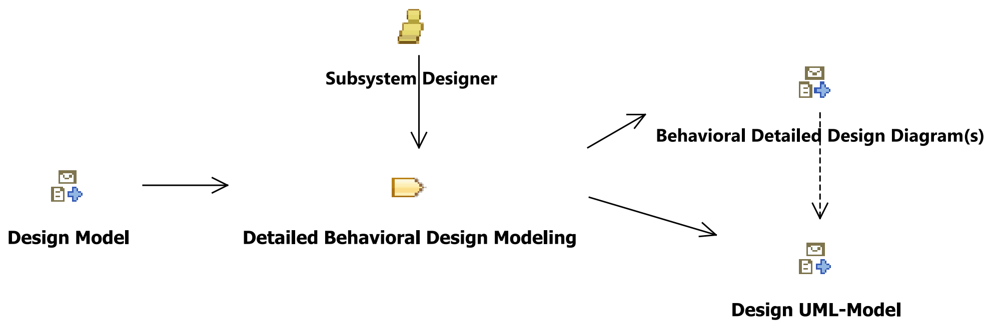


Fig. 30: Example: Structural Detailed Design Diagram

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 the method was designed to be class-based rather than object-oriented; the instance-driven nature of the method causes that inheritance and related polymorphism mechanisms are not regarded, as instances (objects) rather than classes were modeled. 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 modeling, 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, which 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 affects 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. Therefore, we propose to build up a class library to achieve best reuse in such a situation.

2.4.2 Detailed Behavioral Design Modeling



While a structural specification for those design objects, being composed by a subsystem, are developed in terms of classes during *Detailed Structural Design Modeling*, a behavioral specification has to be developed as well. Based on the structural features defined by the developed classes, the internal behavior of each design object is captured during *Detailed Behavioral Design Modeling*. That is, the internal behavior, as it has been specified during *Intra-Object Behavior Modeling* has to be updated to reflect the changes made during *Subsystem Consolidation* (merging or splitting of objects). It further has to be enriched with additional detail, so that it can serve as direct input to the subsequent implementation.

It has to be pointed out that - as the effort related to the developed of the detailed behavioral design is not to be ignored - this should only be performed for those design objects, which do not have trivial behavior, and where the internal behavior cannot be directly inferred from other behavioral specifications developed during *Architectural Design Modeling*. Most often, *Detailed Behavioral Design Modeling* will thus only be applied to specify the detailed behavior of *state-dependent control* and *application-logic* objects, as the behavior of *trigger*, *interface*, and *entity* objects is mostly trivial, and as the behavior of *coordinator* objects can normally be directly inferred from the *Behavioral Subsystem Design Diagrams*, being developed during *Subsystem Consolidation*.

WORK PRODUCTS

- *Behavioral Detailed Design Diagram*: A *Behavioral Detailed Design Diagram* is developed for each design object, which does not have trivial or obvious behavior (to be precise, it is developed for the class of the design object, which is developed during *Detailed Structural Design Modeling*). Dependent on which behavioral formalism is best suited, a UML state machine diagram, as exemplarily outlined by Figure 31, or a UML activity diagram may be employed for this purpose. If not being affected by major changes during *Subsystem Consolidation*, the developed diagram will most likely be detailed versions of the *Intra-Object Behavior Diagrams* developed during *Analysis Modeling*.

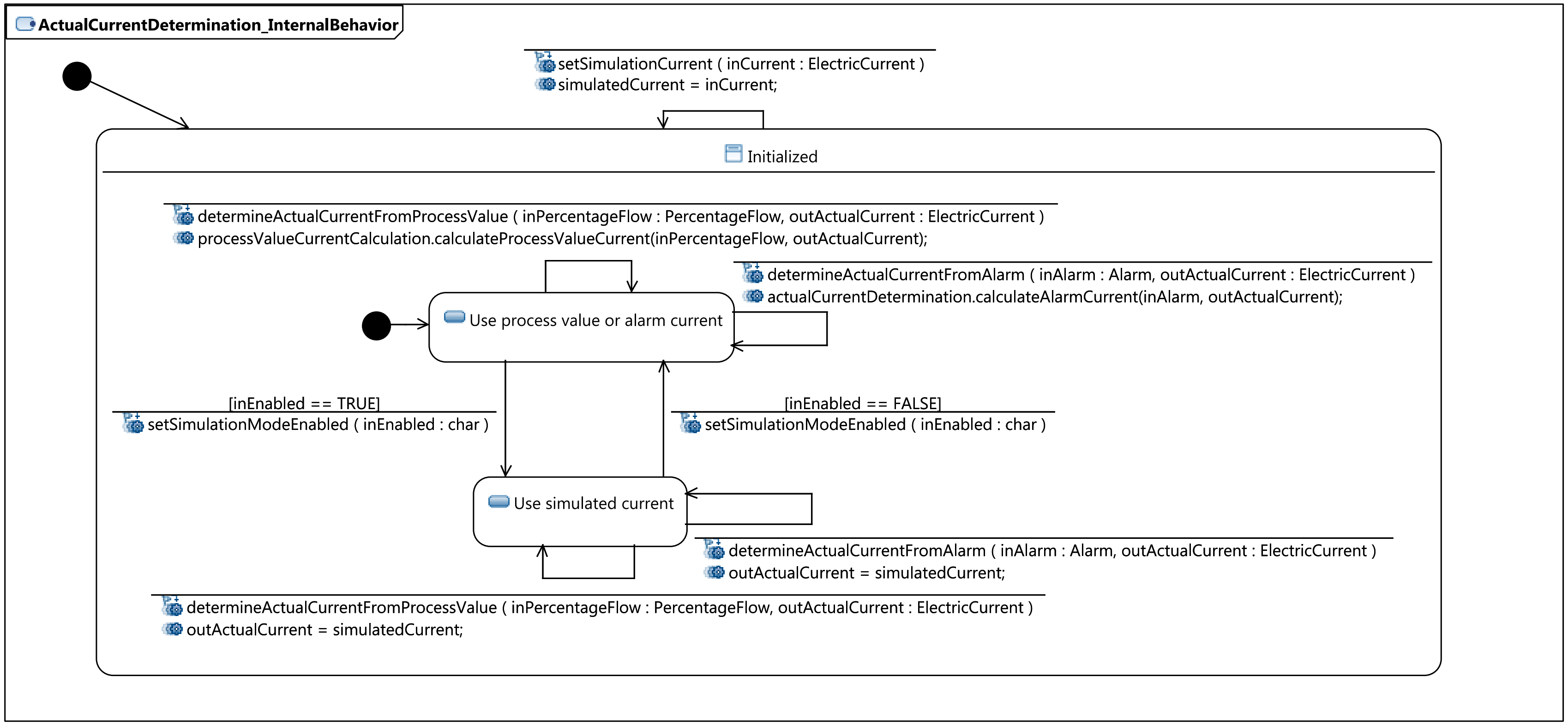


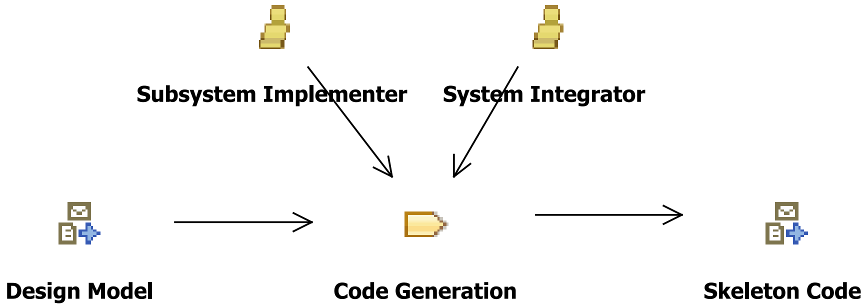
Fig. 31: Example: Behavioral Detailed Design Diagram

2.5 Implementation Discipline

The *Implementation* discipline is concerned with transferring the detailed design into running source code, so it comprises the following tasks:

- *Code Generation* deals with transferring all elements captured in the detailed design into respective source code equivalents (skeleton code). While most of the code will of course reflect the structural design aspects, behavioral code may as well be transferred. Most likely, this task will predominantly be performed automatically by a respective code generation tool.
- *Implementation* is concerned with adding coding details to the skeleton code generated during *Code Generation* to obtain fully running source code for each subsystem.
- *Integration* is concerned with adding glue code to integrate the code fragments for the individual subsystems into a valid source code base for the overall system.

2.5.1 Code Generation



Code Generation is concerned with transferring the information captured in the *Design Model* into their respective source code equivalents. In most cases, all structural aspects captured in MeDUSA *Design Model* can be automatically transferred into their ANSI-C equivalents, as has been demonstrated in [Fun06] by an approach transferring each modeled classifier (i.e. class, interface, association, data type) into a respective C translation unit. Even while this approach does not guarantee best performant and lowest resource-consuming source code and leaves much potential for optimization (e.g. related to the internal wiring of parts and ports inside each subsystem), it shows that a seamless transition of the respective design concepts is achievable (something that was especially enforced by the instance-driven and object-based nature of the method).

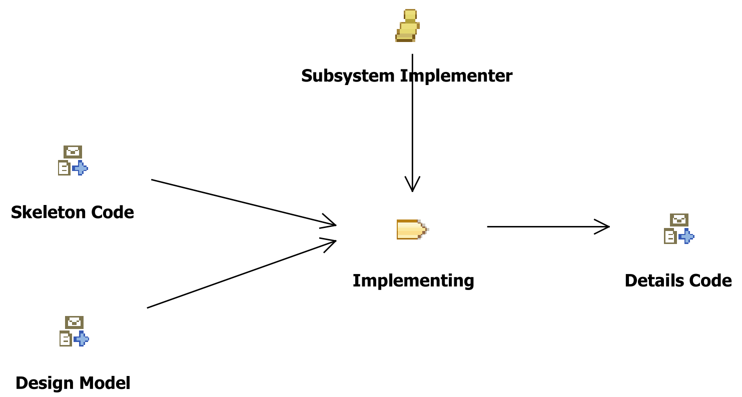
The behavioral aspects captured in the design models are a bit more challenging and usually require manual interaction. This is, because the behavioral system aspects do affect the method bodies and can not be simply mapped to structural source code constructs. As the complete generation of method bodies is normally not possible due to missing details, what is often the case when transferring inter-object behavior, as here only the messages between objects are specified, but not the behavior occuring between those message sending events. As demonstrated in [Kev07], code for intra-object behavior can however be most often automatically generated by a respective tool. Where this is not possible, it is therefore often regarded to be clearer and more straight-forward to not generate any method body detail at all and leave the respective mapping to the subsystem implementer.

Even if all information captured in the *Design Model* is transferred into corresponding source code, it has to be clear that the resultant source code is by nature not complete, and that implementation details are missing. The source code produced by this task is therefore denoted as *Code Skeletons* to emphasize that implementation details have to be added by a successive task.

WORKING RESULTS

– *Skeleton Code*: The result of the *Code Generation* task is source code that is produced by transferring the structural and behavioral information captured in the *Design Model* into respective code equivalents. Examples can be found in Appendix C, where a general transformation for MeDUSA UML models is provided.

2.5.2 Implementing

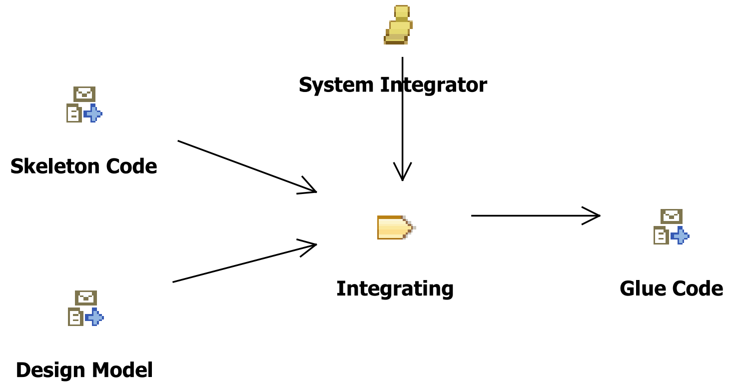


While *Code Generation* is concerned with the direct (potentially automated) transferring of design information into source code, *Implementing* is about adding all implementation details, which are necessary to obtain fully running source code for a subsystem. This comprises implementation of all method bodies, which could not be automatically inferred from already specified behavior in the design models, as well as platform dependent (i.e. hardware dependent) code that was not captured in the *Design Model*. It is performed by the *Subsystem Implementer* for each respective subsystem, and has to be performed in close cooperation with the *Integration* task, being performed by the *System Integrator*.

WORK PRODUCTS

– *Details Code*: As the *Skeleton Code* produced by the *Code Generation* task is naturally missing details, it has to be enriched to form a fully running source code basis. The code fragments that are missing to reach this goal are as a whole referred to as the *Details Code*.

2.5.3 Integrating



While adding source code details to the code bases of the individual subsystems, as performed during *Implementing*, the implementation of the glue code needed to integrate the different code fragments, belonging to the subsystems, to an overall running source code base, is performed once for the overall system. The *System Integrator* is not only responsible of implementing the respective integration code, but indeed also of system wide implementation aspects, as for example related to the startup and initialization.

WORK PRODUCTS

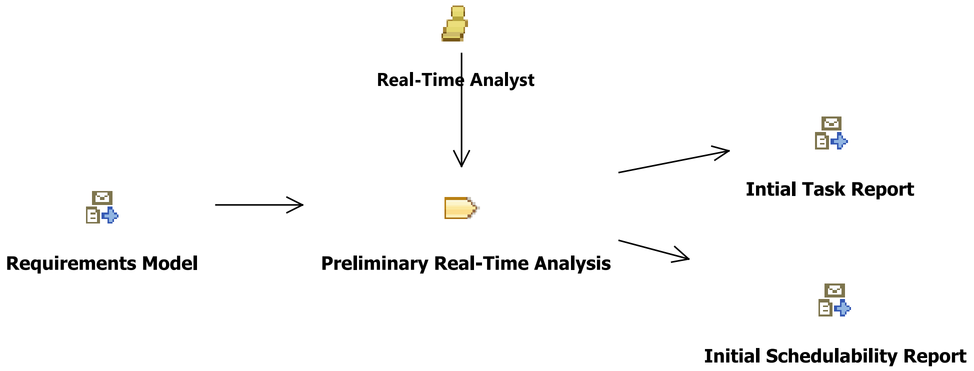
- *Glue Code*: All code details, which may not be attributed to the source code base of an individual subsystem, are globally referred to as *Glue Code*. This comprises the actual integration code, needed to combine the subsystem specific code fragments, as well as source code needed for system-wide aspects as the startup and initialization.

2.6 Real-Time Analysis Discipline

As non-functional real-time requirements play an outstandingly important role in the context of embedded & real-time software, they have to be regarded intensely. The *Real-Time Analysis* discipline thus comprises real-time analysis tasks that can be performed based on the different models that are produced by the constructive disciplines. In detail, the following tasks make up the *Real-Time Analysis Discipline*:

- *Preliminary Real-Time Analysis*: Analyzes the use case model (i.e. use cases and their detailed descriptions) regarding timing and concurrency constraints to gain a first impression on feasibility and to identify potential problems.
- *Interim Real-Time Analysis*: Is a real-time analysis similar to the *Preliminary Real-Time Analysis*, just that it is performed based on the *Analysis Model*. As the *Analysis Model* is more detailed than the use case model, real-time analysis can now also be performed in a more detailed fashion. Estimations can e.g. be broken down to message communication between objects rather than whole use cases.
- *Conclusive Real-Time Analysis*: Based on the *Design Model*, the *Conclusive Real-Time Analysis* can take into consideration all changes implied by design consolidation. It is thus even more detailed and precise and is used as the concluding analysis of the system's schedulability and performance.

2.6.1 Preliminary Real-Time Analysis



As timing and concurrency concerns are of outstanding importance for real-time systems, they have to be investigated intensely and as early as possible. As briefly sketched in [NL07b], the application of the *MeDUSA Actor Taxonomy* offers the possibility of an early real-time analysis based on the use case model. While such an analysis might not give the resilient guarantee to proof the schedulability of the software system under construction, it might give valuable insight in terms of potential performance problems and might as well help to detect a non-feasible hardware architecture at an early development stage (MeDUSA assumes that the hardware development has some sort of precedence and is performed with a slight temporal advance).

Real-time analysis is started by annotating timer periods or respective worst-case interarrival times to timer and eventer actors. Those constraints can usually be directly inferred from the non-functional requirements. Each concurrently executed scenario - referred to as a task candidate [3] in the following, has to be estimated regarding its execution time. That is, the worst case scenario being implied (in terms of required computation time) has to be identified and its respective CPU utilization has to be estimated.

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 that are applicable.

WORK PRODUCTS

– *Initial Task Report*: The results of the initial real-time analysis task should be captured in an *Initial Task Report*. It should list for each task candidate the following information:
  - the periodic and aperiodic event source (actor), from which the task candidate originates
  - the task candidate's frequency (timer period or worst case interarrival time, in case of eventer)
  - a description of the purpose of the respective task candidate (derived from the purposes of the involved use cases)

[3] we refer to them as task candidates and not tasks, as the final task design cannot be defined before the system architecture has been defined

- an estimation of the task candidate's CPU consumption time (i.e. the execution time of the worst-case scenario)
- 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 32.

Task Report

#	Trigger	Scenario	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)
$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$

Fig. 32: Example: 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 an *Initial Schedulability Report*. The form of the report depends on the concrete type of selected schedulability analysis, which is not prescribed by MeDUSA. However, independent of the applied technique, the *Initial Schedulability Report* should provide an estimation for the individual task candidates and the overall system schedulability. We will elaborate this on our running 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 32. 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 it has the longer period. Therefore, each task candidate has to be analyzed individually.

Initial 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 $5\mu s$, leading to a utilization of $\frac{5\mu s}{25\mu s} = 0.2$.
  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 $\frac{4\mu s}{T_1} = \frac{4\mu s}{25\mu s} = 0.16$.

  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 $\frac{3\mu s}{T_2} = \frac{3\mu s}{500\mu s} = 0.006$.

  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 $\frac{C_2}{T_4} = \frac{70\mu s}{200\mu s} = 0.35$.
  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 exclusive 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.

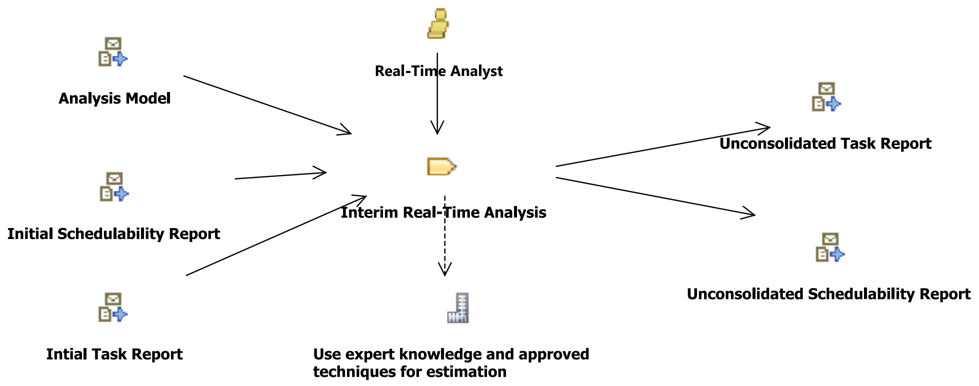
  The worst case utilization of task t_4 can therefore be computed to $0.2 + 0.015 + 0.35 + 0.025 = 0.59$, which is below the utilization bound of 0.69, so also task candidate t_4 will meet its deadline.

Fig. 33: 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.

2.6.2 Interim Real-Time Analysis



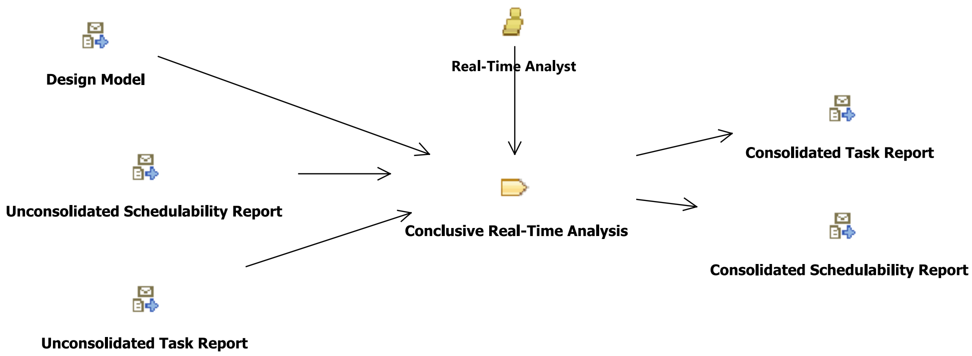
Having identified the sources of concurrent behavior during early *Use Case Modeling* by using trigger actors and having identified during *Inter-Object Collaboration Modeling* how each such concurrent behavior is established in terms of messages between objects, participating in the collaborations identified for each use case, a more detailed real-time analysis on whether each respective task candidate is able to hold its deadline and on whether the overall system is at all schedulable can be performed. That is, because estimating the CPU consumption of a respective task candidate can now be broken down to estimating the time required for the processing of all messages exchanged by the respective object-collaboration performing the task candidate, adding an additional overhead for the message communication itself.

It has to be emphasized again that even if the overall task design is not established yet, valuable information can already 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.

WORK PRODUCTS

– *Unconsolidated Task Report*: The results of the *Interim Real-Time Analysis* is captured in a respective *Unconsolidated Task Report*. It is comparable to the *Initial Task Report* produced during *Preliminary Real-Time Analysis*, just that now trigger objects rather than actors are regarded to be the initiators of a task candidate and that the estimations can be performed much more fine-grained based on the level of inter-object message communication and involved intra-object behavior.
– *Unconsolidated Schedulability Report*: The detailed outcome of the performed schedulability analysis has to be captured in a respective schedulability report. It is a refinement of the schedulability report produced during *Preliminary Real-Time Analysis*, now being based on the data captured in the *Unconsolidated Task Report*.

2.6.3 Conclusive Real-Time Analysis



While the introduction of subsystems may have affected the performance and schedulability of the system in a negative way (inter-subsystem communication is usually more *expensive* than intra-subsystem communication), the consolidation of active objects had a corrective effect. To assess the consequences of both on the overall performance and schedulability, a conclusive real-time analysis has to be performed. This is done using the same techniques as in the *Interim Real-Time Analysis* task of the *Analysis Modeling* discipline, being now however based on an actual task design rather than on task candidates.

WORK PRODUCTS

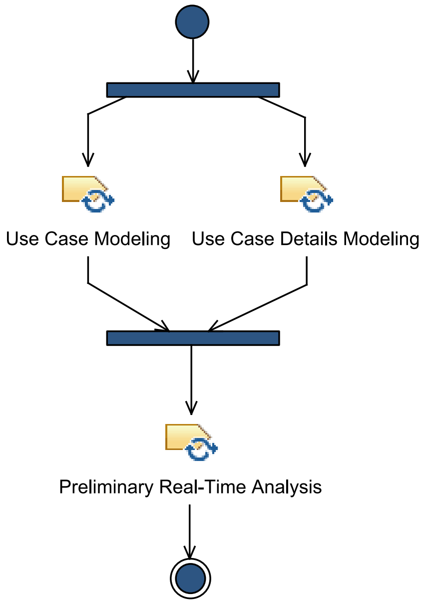
- *Consolidated Task Report*: The results of the conclusive real-time analysis should be captured in a *Consolidated Task Report*, which is identical to the *Unconsolidated Task Report* produced during *Interim Real-Time Analysis* with the exception that it now indeed lists the tasks (gathered by consolidating active objects) and no longer task candidates.
- *Consolidated Schedulability Report*: Besides the *Consolidated Task Report*, which captures 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 *Unconsolidated Schedulability Report* produced during *Interim Real-Time Analysis*.

3 Process

As outlined before, the *Process* part of a method definition specifies how the *Method Content* elements are employed over time. This is done in terms of *Task Uses*, *Iterations*, *Phases*, *Process Patterns*, and *Delivery Processes*. The MeDUSA definition makes use of those SPEM concepts by defining five *Process Patterns* (referred to as workflow patterns in the following), which group all *Task Uses* related to a respective lifecycle phase. The workflow patterns are thus referred to as *Requirements*, *Analysis*, *Architectural Design*, *Detailed Design*, and *Implementation* respectively. They are outlined in detail in the following.

The MeDUSA workflow, defined as a *Delivery Process*, is then introduced. It consists of five *Phases*, namely *Requirements Phase*, *Analysis Phase*, *Architectural Design Phase*, *Detailed Design Phase*, and *Implementation Phase*, where each phase comprises a set of *Iterations* of the homonymous workflow pattern, as well as all workflow patterns related to earlier lifecycle phases, in case reiterations are needed.

3.1 Requirements Workflow Pattern



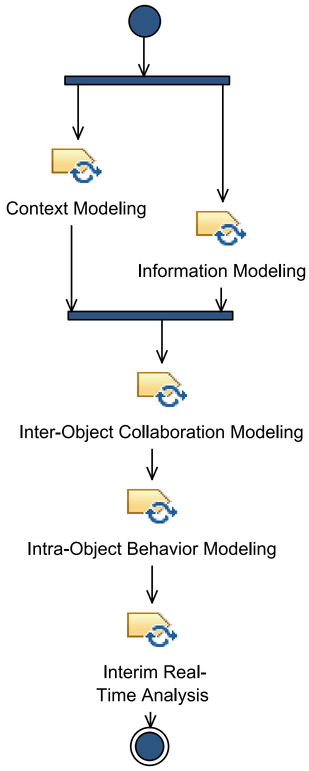
The *Requirements* workflow pattern is concerned with the construction (and analysis) of the *Requirements Model*. As *Requirements UML-Model* and *Narrative Model*, the two major fractions of the *Requirements Model*, have to be consistent to each other and as either of them may serve as valuable input for the construction of the other - e.g. when trying to determine the right granularity - the two tasks 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, use case details for each initially identified use case may be developed, which may lead in turn to modifications inside the *Use Case Diagram(s)* (respectively the *Requirements UML-Model*), as additional use cases or relationships between use cases may be identified. This way, both model fractions are developed in parallel, until they are consistent to each other and their quality is satisfying.

As non-functional timing and concurrency concerns are of outstanding importance, an early preliminary real-time analysis is subsequently performed on the *Requirements Model* to identify potential performance problems as early as possible.

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

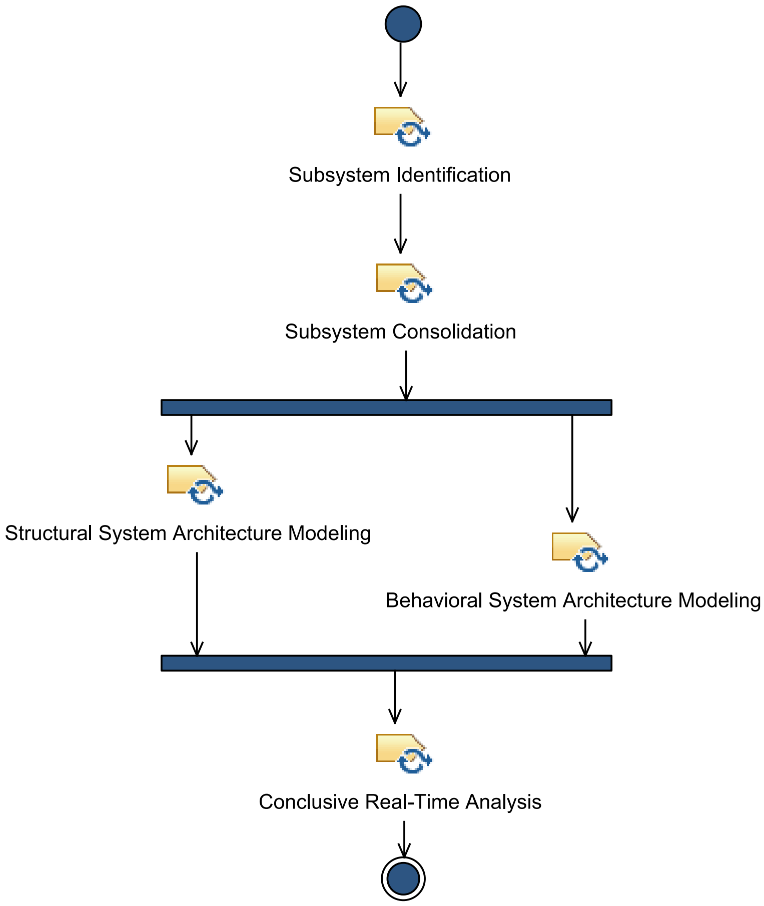
Therefore, the tasks subsumed by the *Requirements* workflow pattern are very essential and have to be performed very thoroughly. Defects and misunderstandings not resolved here can cause costly fixes in later activities. Especially misunderstandings and mistakes related to the concurrency concerns, 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 to be pointed out therefore that even if the regarded domain is already well understood or similar products have already been developed, it is essential to perform this step in the described detail.

3.2 Analysis Workflow Pattern



Based on the functional and non-functional (timing and concurrency) requirements, captured in the *Requirements Model*, the *Analysis* workflow pattern is concerned with gathering a thorough understanding of the problem domain. Therefore, an *Analysis Model* is constructed, which is done in detail by modeling an object collaboration for each identified use case. 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 during *Context Modeling* and *Information Modeling*. As both tasks are concerned with different types of objects, they can be performed very much in parallel. After trigger, interface and entity objects have been defined, the remaining tasks are executed to identify additional control and application-logic objects needed to execute the use cases, and to specify their internal behavior (in case it is not trivial). Last, a more detailed impression on the capability of the software system to meet its performance constraints has to be gained by performing an *Interim Real-Time Analysis*.

3.3 Architectural Design Workflow Pattern

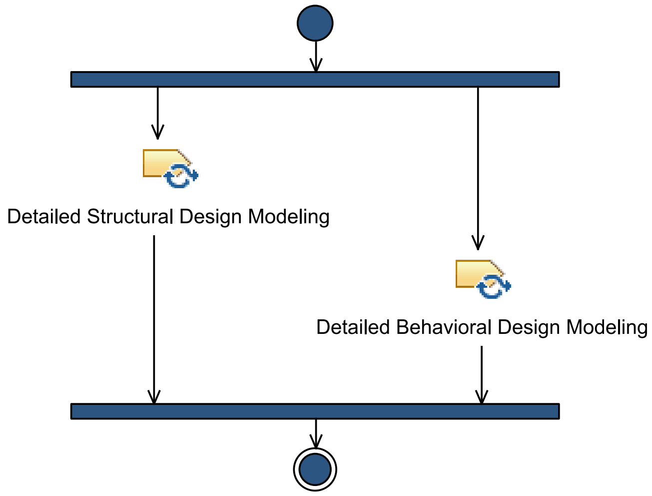


The objective of the *Architectural Design* workflow pattern is the specification of the software architecture. That is, subsystems have to be identified during *Subsystem Identification* by grouping together objects of the *Analysis Model*. An initial version of each subsystems' required and provided interfaces also has to be inferred from the messages being exchanged between objects, partitioned into different subsystems. After the initial partitioning of objects, a consolidation of the subsystem design has to be performed. That is, the active and passive analysis objects, as well as the provided and required interfaces of each subsystem have to be investigated under design considerations. In detail, active objects may be clustered together in order to improve the overall system performance and the provided and required interfaces have to be detailed regarding the data types, which are used as parameter types in their signatures. Both tasks, *Subsystem Identification* and *Subsystem Consolidation* are of course strongly related and may also be merged together (i.e. partitioning and consolidation are done in one step).

After consolidation, the structural and behavioral relationships between the identified subsystems can be designed. This is done during *Structural System Architecture Modeling* and *Behavioral System Architecture Modeling*. The definition of structural and behavioral relationships is strongly intertwined. Of course, one might start with defining an initial version of the structural relationships first, as the behavioral relationships have to reside on them. However, while modeling the behavioral relationships, changes to the structural relationships are likely to occur, so that from the initial definition 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 behavioral relationships has been defined, the overall system performance has to be reflected in terms of a *Conclusive Real-Time Analysis*. In detail, the changes implied by the introduction of subsystems and by their consolidation have to be investigated in terms of the overall system schedulability.

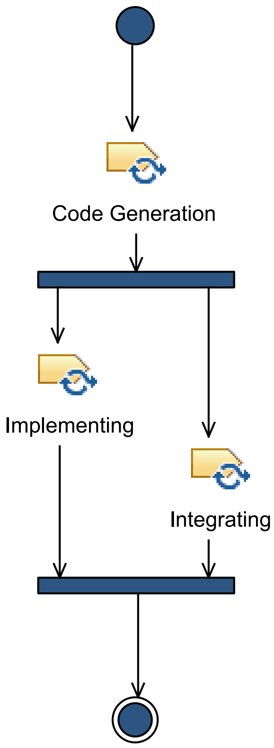
3.4 Detailed Design Workflow Pattern



After having performed the *Architectural Design*, all externally visible required and provided interfaces of the identified subsystems are clearly specified and the system behavior that occurs over these interfaces is well understood. Also, the internal subsystem decomposition in terms of active and passive (design) objects has been defined.

Now, *Detailed Structural Design Modeling* can be applied to create a class design for each subsystem, which can then in turn be seamlessly transferred into source code skeletons by the succeeding implementation activities. Further, *Detailed Behavioral Design Modeling* may be performed to reflect the internal behavior of those design objects, where is it neither trivially clear, nor can be inferred from those behavior diagrams, being developed during *Architectural Design Modeling*.

3.5 Implementation Workflow Pattern



The objective of the *Implementation* workflow pattern is the development of the system's source code based on the detailed system design. First, the detailed system design being captured in the *Design Model* has to be transferred into resultant source code, as subsumed by the *Code Generation* task of the respective *Implementation* discipline. The detailed system design does of course not reach down to the level of abstraction offered by the final running source code. Therefore, the source code being (automatically) generated from the detailed design models is regarded to be skeleton code, which has to be completed by all respective details that are needed to transform it into a complete source code base. This is done during *Implementing* for the code affecting a respective subsystem, as well as during *Integrating*, as far as subsystem integration related source code is concerned.

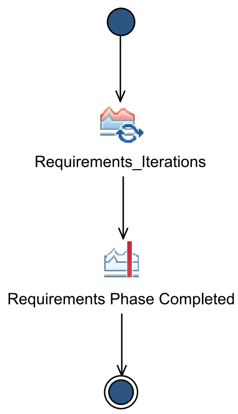
3.6 The MeDUSA Workflow

The definition of the *MeDUSA Workflow* is based on the five already introduced workflow patterns that define the sequencing of the tasks belonging to the six disciplines (five modeling disciplines plus the additional real-time analysis discipline) of the *Method Content*. In detail, the workflow is defined in terms of five *Phases*, namely *Requirements Phase*, *Analysis Phase*, *Architectural Design Phase*, *Detailed Design Phase*, and *Implementation Phase*, as shown in Figure 34.

Fig. 34: MeDUSA Delivery Process

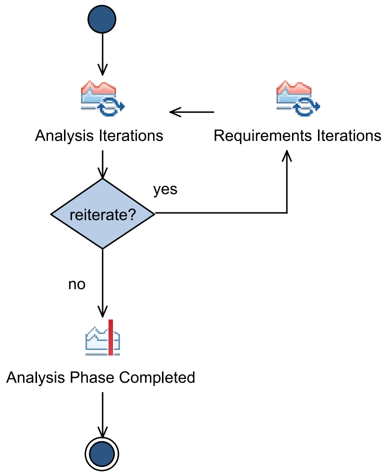
Each phase is self-contained unit, concerned primarily with the iterative execution of the homonymous *Requirements*, *Analysis*, *Architectural Design*, *Detailed Design*, and *Implementation* workflow pattern, as well as with re-iterations of workflow patterns, primarily executed in prior phases. The goal of each phase is to develop a concise model (i.e. a *Requirements Model* at the end of the *Requirements Phase*, an *Analysis Model* at the end of the *Analysis Phase* and so on), which is indicated by a respective *Milestone* at the end of each *Phase*.

Requirements Phase



The *Requirements Phase* is concerned with the iterative execution of the *Requirements* workflow pattern to develop a concise and consistent *Requirements Model*.

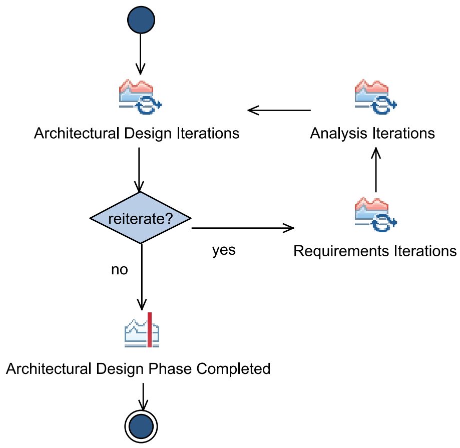
Analysis Phase



The *Analysis Phase* is concerned with producing a concise *Analysis Model*. Here the activities defined by the *Analysis* workflow pattern are executed iteratively. What has to be pointed out is that especially *Analysis* 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 Modeling* but not earlier than during *Inter-Object Collaboration Modeling*, when the object collaborations performing the use cases are developed. It may also be the case that during the activities of the *Analysis* interface or trigger objects are identified that were not already determined during *Requirements Modeling*. Therefore the *Analysis Phase* allows to reiterate the *Requirements* workflow pattern.

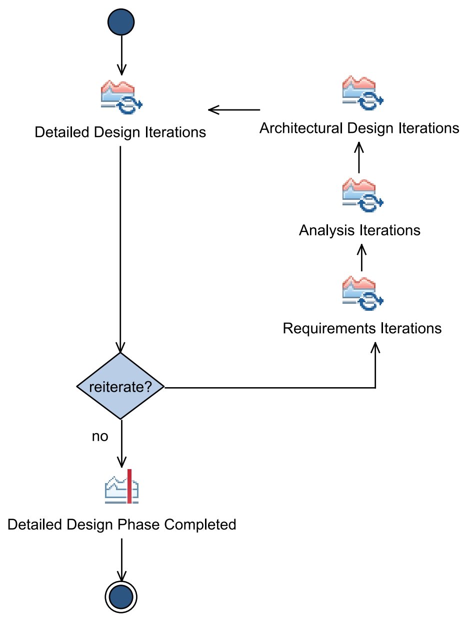
When reaching the concluding milestone, all relevant interface, trigger, entity, control and application-logic objects should be identified and their collaborating behavior for each of the use cases identified during the *Requirements Phase* should have been captured. Also, the internal behavior of each non-trivial object should be captured. That is, with the end of *Analysis Phase*, a thorough understanding of the problem domain has been achieved, so that in the following *Architectural Design Phase* the composition of a solution can be started.

Architectural Design Phase



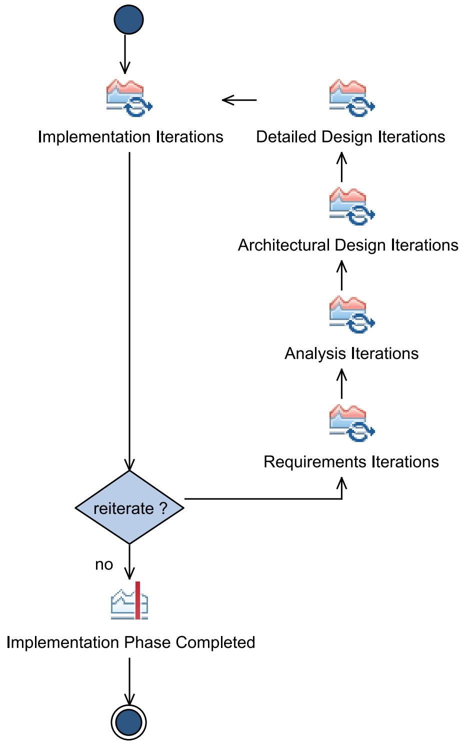
The *Architectural Design Phase* deals with the specification of the software architecture. That is, the *Architectural Design* workflow pattern is iteratively executed to construct a concise *Design Model*, defining the software architecture in terms of structurally and behaviorally interrelated subsystems. It has to be pointed out that *Architectural Design* is rather iterative, first in itself, as the activities may have to be gone through in several iterations until a feasible software architecture has been specified. Second in a sense that most likely changes to the *Analysis Model* will be noticed while developing the *Architectural Design 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 behavior is extracted, it may be easier to place the entity object in a subsystem.” As changes to the *Analysis Model* or even the *Requirements Model* may therefore be likely, reiterations of the *Analysis* and *Requirements* workflow patterns are covered accordingly.

Detailed Design Phase



The *Detailed Design Phase* is concerned with adding additional detail to the *Design Model*, so that it can be taken as input for the implementation.In distinction to the previous phases, the tasks subsumed by the *Detailed Design* workflow pattern are not executed once for the overall system, but individually for each subsystem. Therefore, if changes to the *Architectural Design Model* (or even *Analysis Model* or *Requirements Model*) are determined, the *Architectural Design* (and possibly *Analysis* and *Requirements*) workflow pattern may be reiterated again to incorporate these changes.

Implementation Phase



The *Implementation Phase* forms the last phase of the *MeDUSA Workflow*. Its objective is to develop fully running source code. This is achieved by iterating the *Implementation* workflow pattern. As with the other phases, reiterations of previous patterns are possible.

4 Summary & Conclusion

Having started as a mere advancement to the COMET method, following the goal to overcome the shortcomings, the COMET method showed during its practical application, MeDUSA had successively grown into a self-contained method, initially published as a technical report [NL07a].

As we stated in the introduction, MeDUSA was initially designed having in mind the major goal to provide a continuous, breachless approach, covering all respective construction activities and further, to reflect the special technical, organizational, and economical constraints that can be faced in the domain of small embedded & real-time software.

MeDUSA in particular faces the first objective by its class-based and instance-driven nature. As class design is done quite late, a seamless transition into a procedural implementation language like C can be easily achieved. This is explicitly covered in terms of the *Implementation* phase (and discipline).

The second objective is addressed by investigating non-functional timing and concurrency constraints right from the beginning, to identify performance problems as early as possible and to thus explicitly consider the strong technical constraints. It is further addressed by choosing the UML as the underlying notation, as the selection of a standardized notation has several organizational and economical advantages, as well as by the iterative nature of the method, which offers increased flexibility and customizability.

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 of the method as well as further supporting material.

A MeDUSA UML-Models - Instance Specifications

While the definition of the tasks, as it is provided in Chapter 2, concentrates very much on the detailed specification of the UML diagram work products, a deep understanding of each task requires that the underlying UML-model contribution is also clearly defined. This is done in the following by illustrating the underlying model artefacts, being contributed by each example UML diagram, which is provided in Chapter 2.

We employ the notation of UML object diagrams to do so, using *InstanceSpecifications* to represent the individual contributed model artefacts (which are instances of UML meta classes) and using *Links* to denote relationships between them (thus representing instances of associations, being defined between the UML meta classes).

As it is not practical to list all contributed model artefacts, we will restrict ourselves to a subset of typical model artefact representatives, to enhance readability and accessibility. Therefore each model instance specification is advanced by a modified version of the related UML example diagram, where those diagram artefacts, which are represented in the model instance specification, are highlighted. Where model artefacts were contributed by earlier developed diagrams, and not directly by the related UML example diagram, the respective instance specification diagram will show those artefacts grayed out.

A.1 Requirements UML-Model

The major contributions to the *Requirements UML-Model* are made via the *Use Case Diagram(s)* in terms of a system *Component*, *Use Cases*, owned by the system, as well as internal (i.e. owned) and external *Actors*. Relationships between the use cases and actors are manifested in terms of *Includes*, *Extends*, *Generalizations*, and *Associations*, as well as *Dependencies*, as depicted by Figure 36.

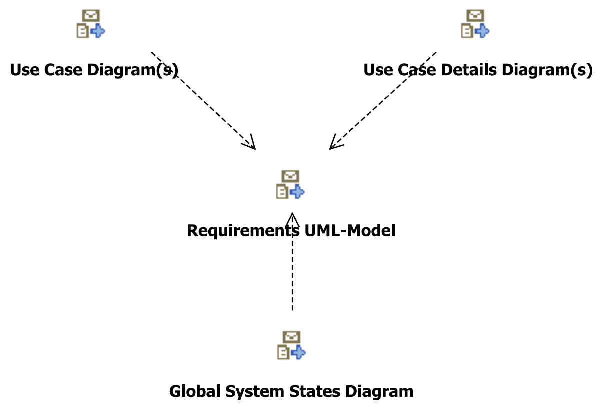
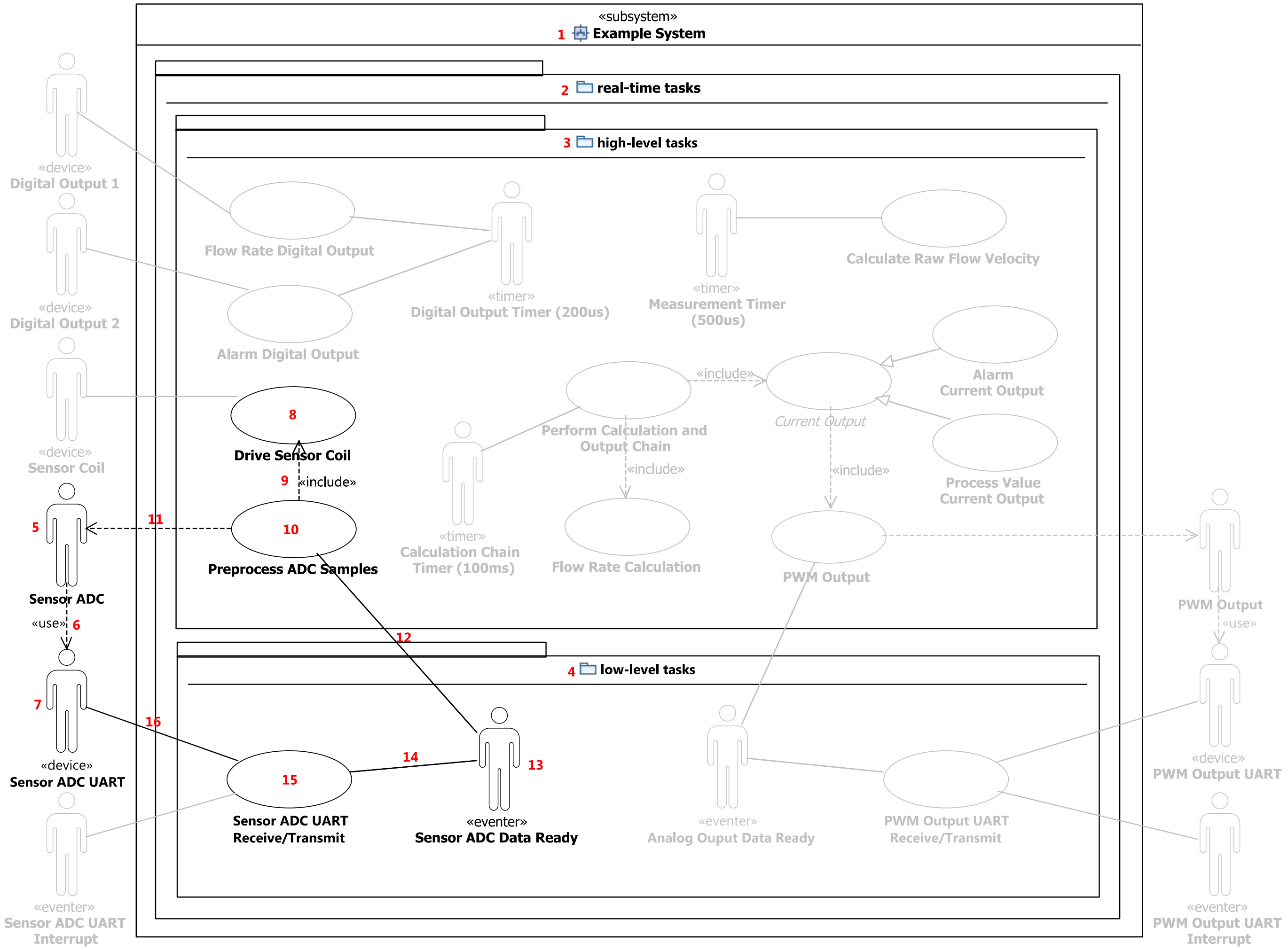


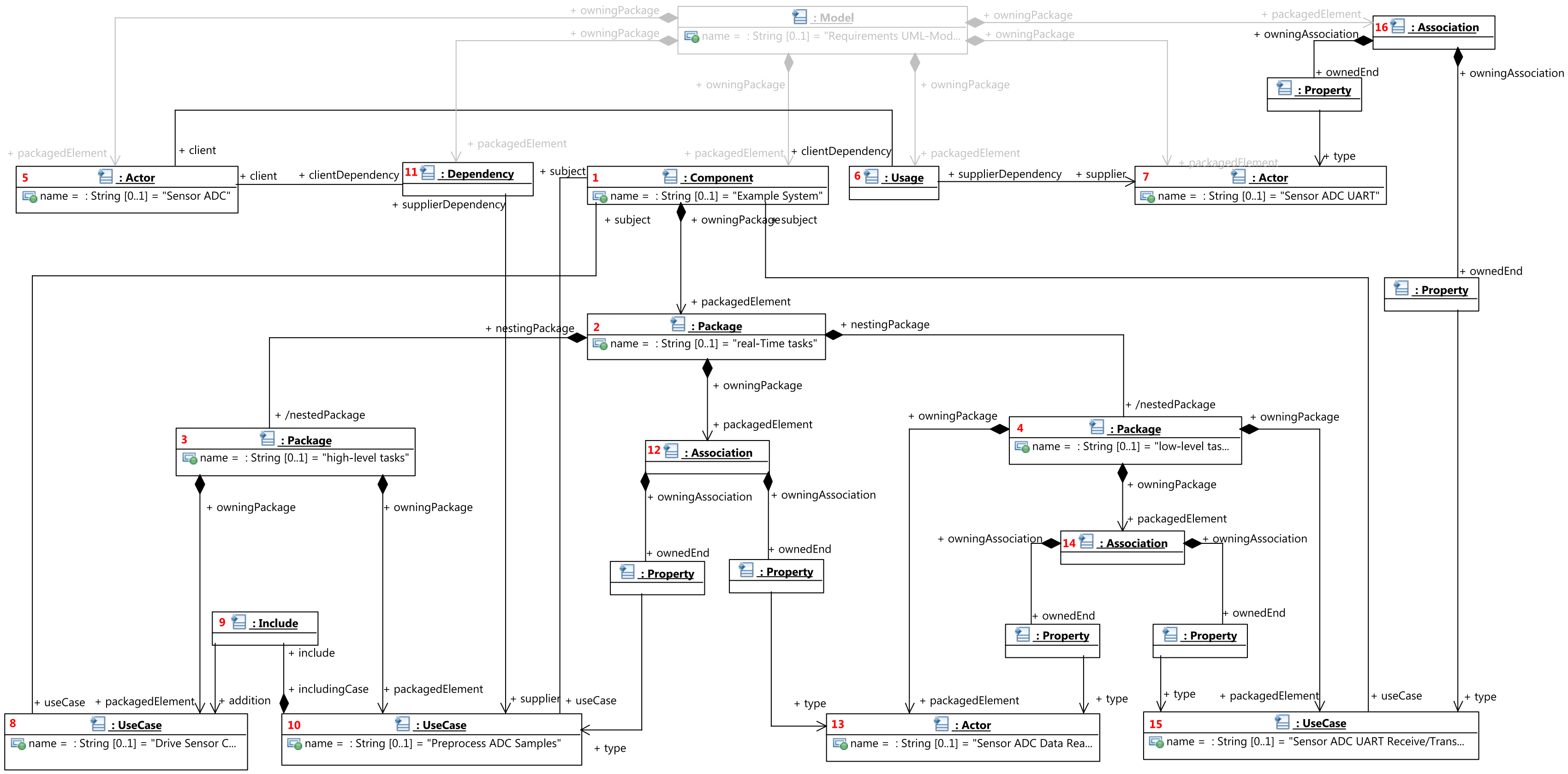
Fig. 35: Requirements UML-Model and related UML diagrams

As outlined by Figure 35, further artefacts are contributed via the *Global System States Diagram* and several *Use Case Details Diagrams*. While the basic contribution added by the *Global System States Diagram* is a *State Machine* owned by the system *Component*, the artefacts contributed via the *Use Case Details Diagrams* are owned *Behaviors* in the form of *Activities*, or *Interactions* and related nested elements, dependent on the respective behavioral formalism. Figure 38 depicts the contribution in case an activity diagram is employed.



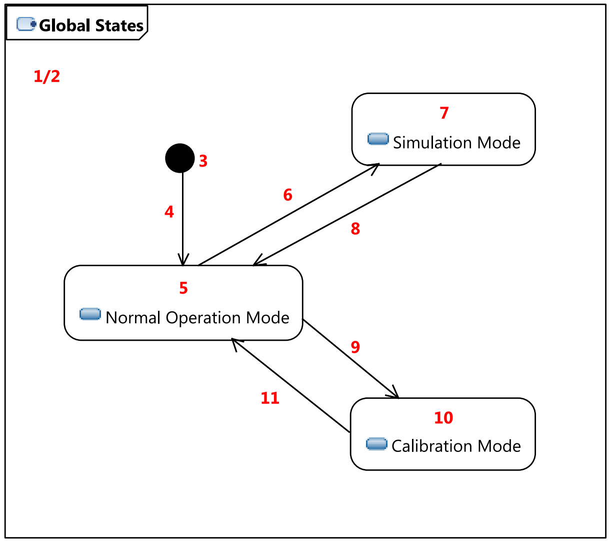
(a) Example Diagram

Fig. 36: Requirements UML-Model - Use Case Diagram



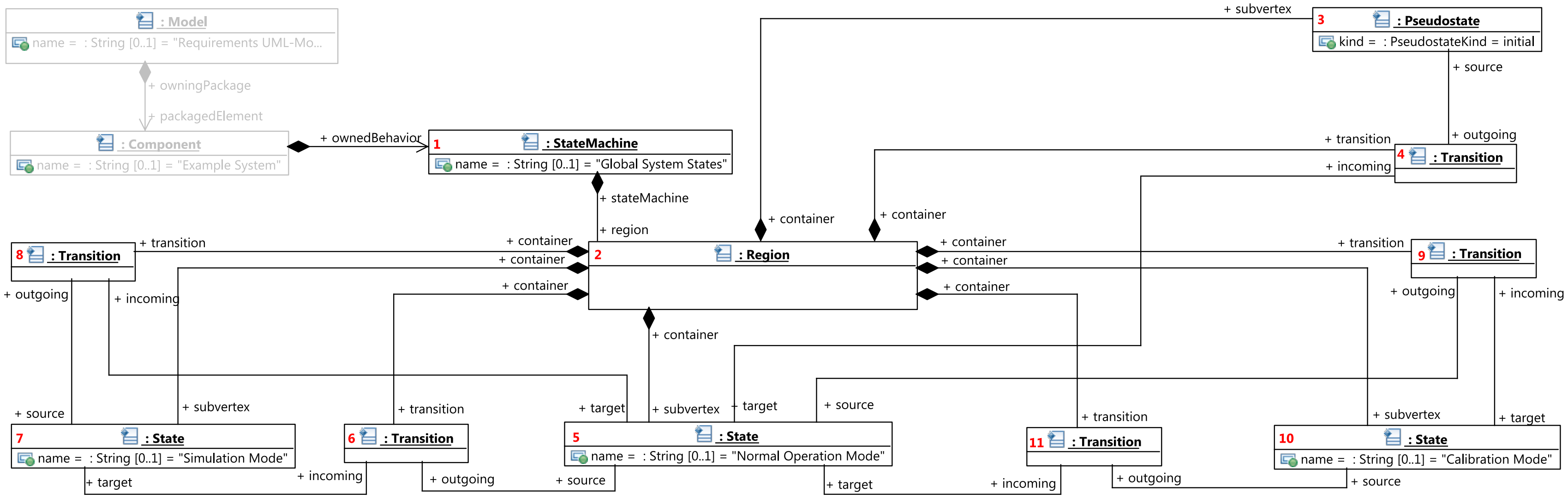
(b) Contributed Model Artefacts

Fig. 36: Requirements UML-Model - Use Case Diagram



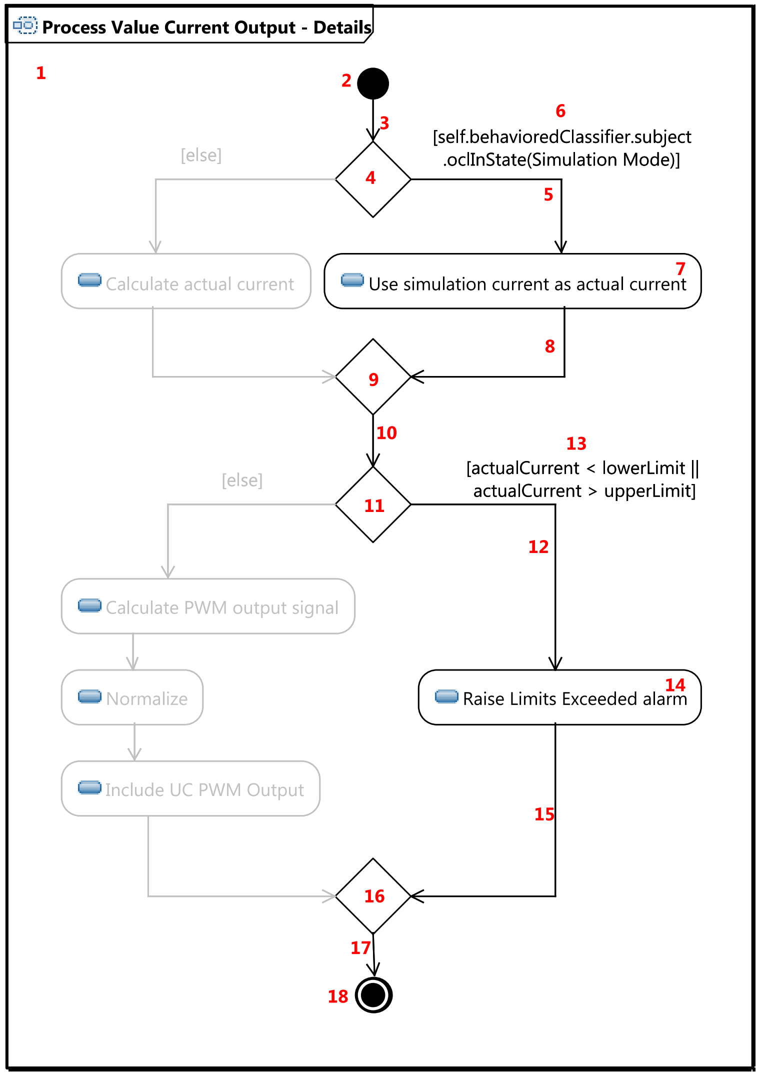
(a) Example Diagram

Fig. 37: Requirements UML-Model - Global System States Diagram



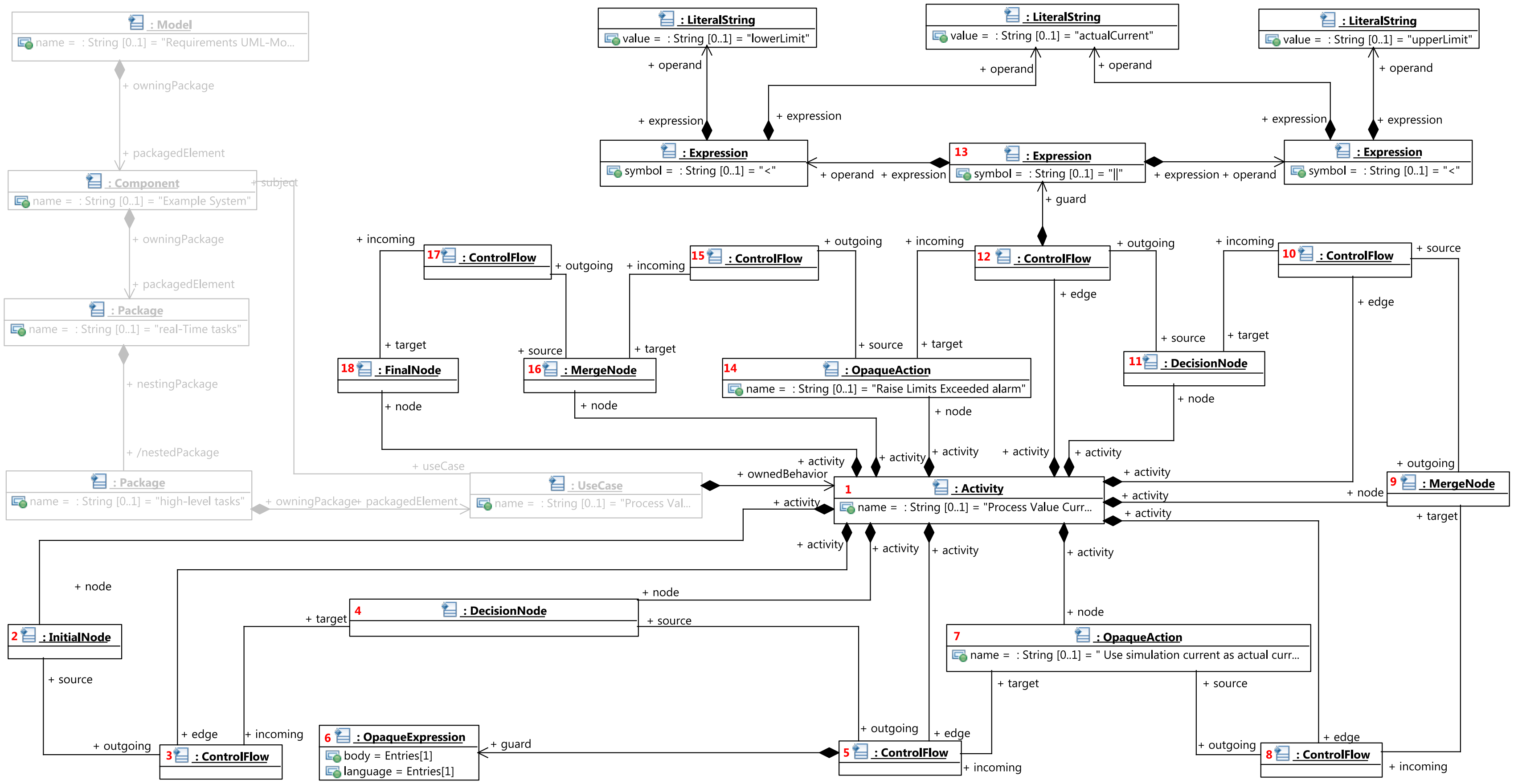
(b) Contributed Model Artefacts

Fig. 37: Requirements UML-Model - Global System States Diagram



(a) Example Diagram

Fig. 38: Requirements UML-Model - Use Case Details Diagram



(b) Contributed Model Artefacts

Fig. 38: Requirements UML-Model - Use Case Details Diagram

A.2 Analysis UML-Model

The contributions added to the *Analysis UML-Model* via the *Context Diagram(s)* and *Information Diagram(s)* are *Instance Specifications* used to represent analysis objects, as well as *Links* and *Dependencies* to denote relationships between them.

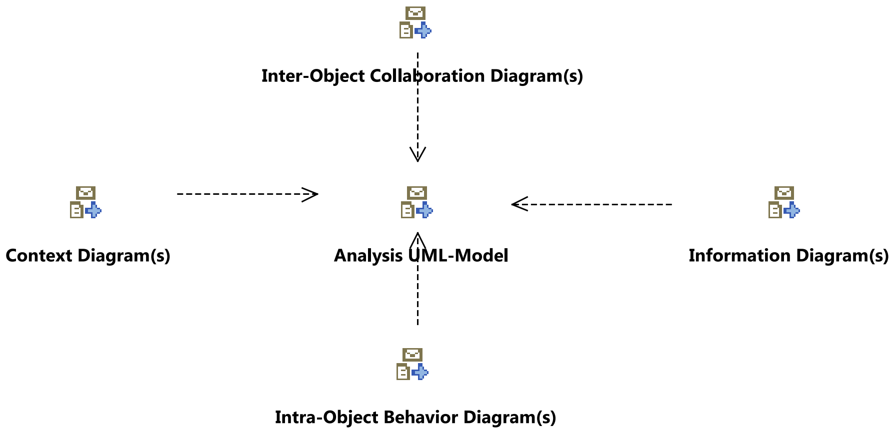
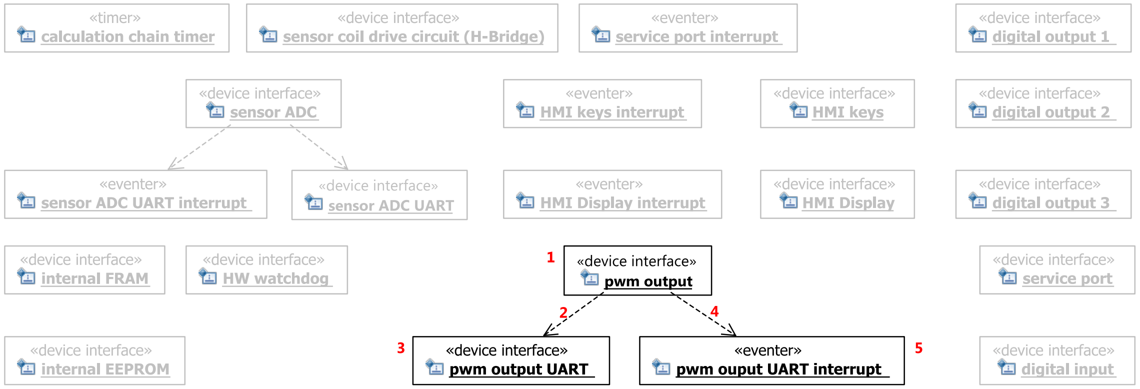


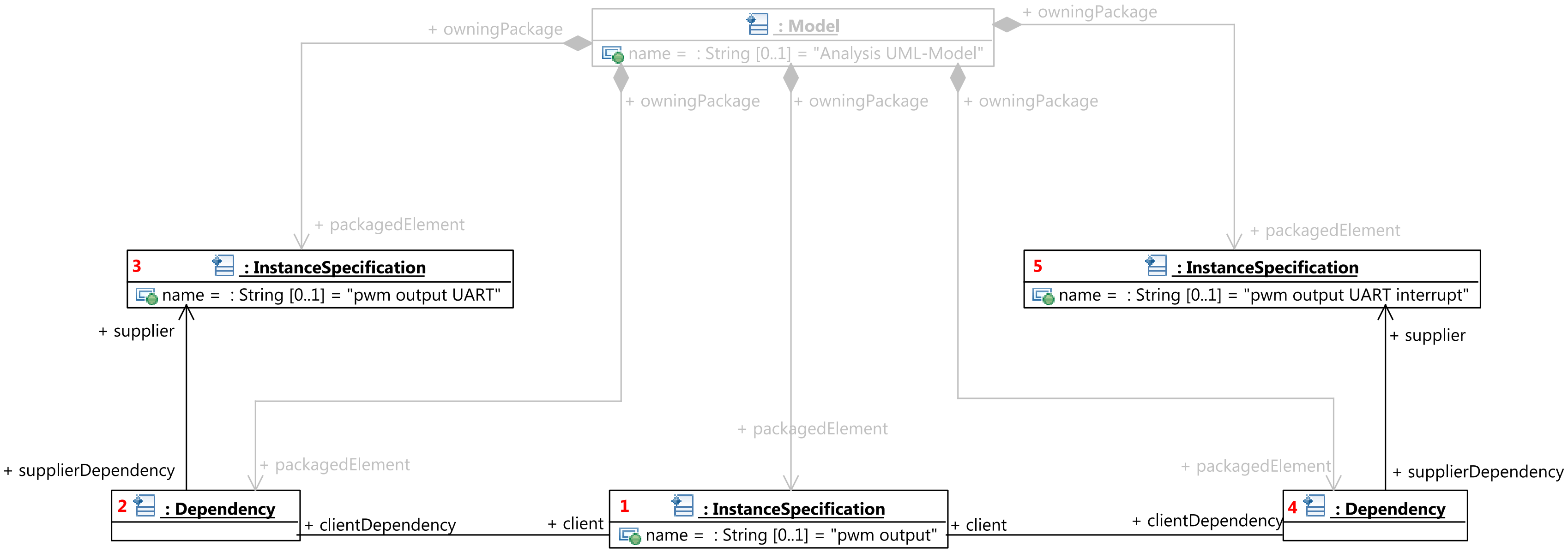
Fig. 39: Analysis UML-Model and related UML diagrams

As outlined by Figure 39 additional contributions are made via the *Inter-Object Collaboration Diagrams*, as well as *Intra-Object Behavior Diagrams*. Independent on whether a sequence or communication diagram is used, the major contribution of a *Inter-Object Collaboration Diagram*, is an *Interaction*, together with its nested *Lifelines* (used to represent the analysis objects) and *Messages*, being exchanged between them. For the *Intra-Object Behavior Diagrams*, different *Behaviors* may be contributed, dependent on the concrete behavioral formalism that is applied (i.e. activity or state machine diagram). In all cases, it has to be pointed out that all *Behaviors*, being contributed by the *Inter-Object Collaboration Diagrams* as well as the *Intra-Object Behavior Diagrams*, are directly contained in the *Model*, as no *Classifiers* are modeled yet and *Instance Specifications* and *Lifelines*, which are used to represent analysis objects in the different employed diagram types, do not support owned *Behaviors*.



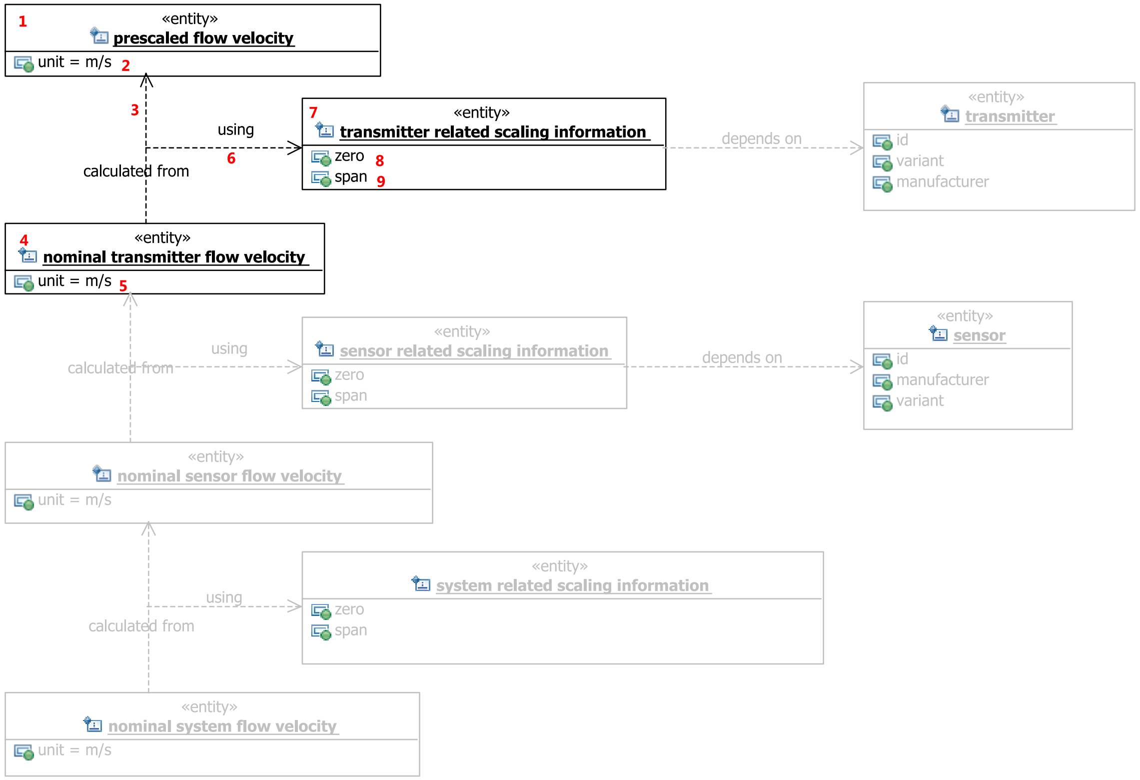
(a) Example Diagram

Fig. 40: Analysis UML-Model - Context Diagram



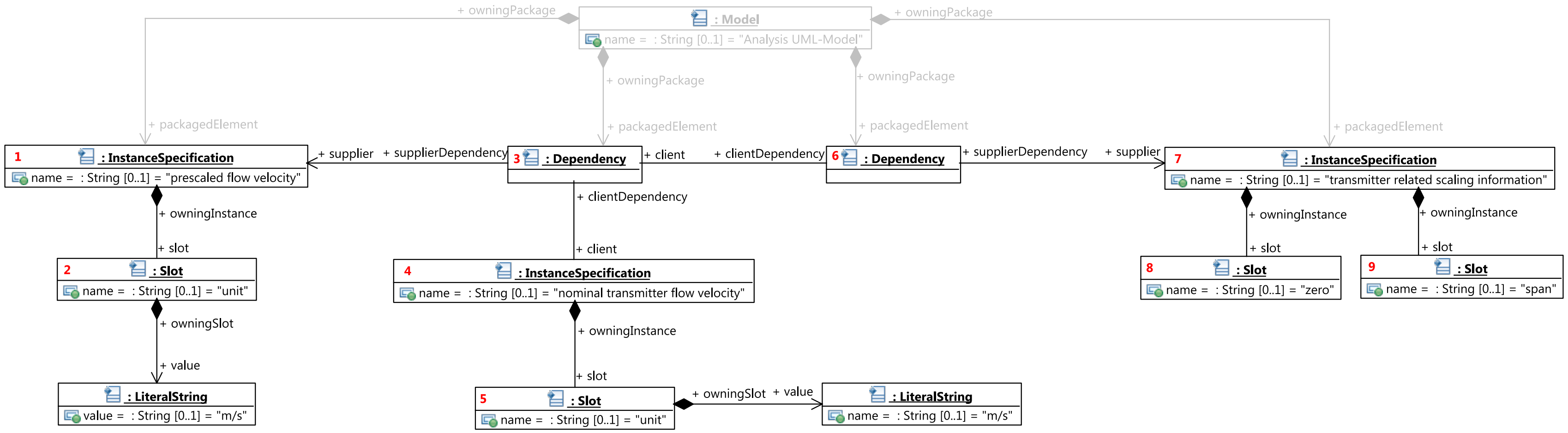
(b) Contributed Model Artefacts

Fig. 40: Analysis UML-Model - Context Diagram



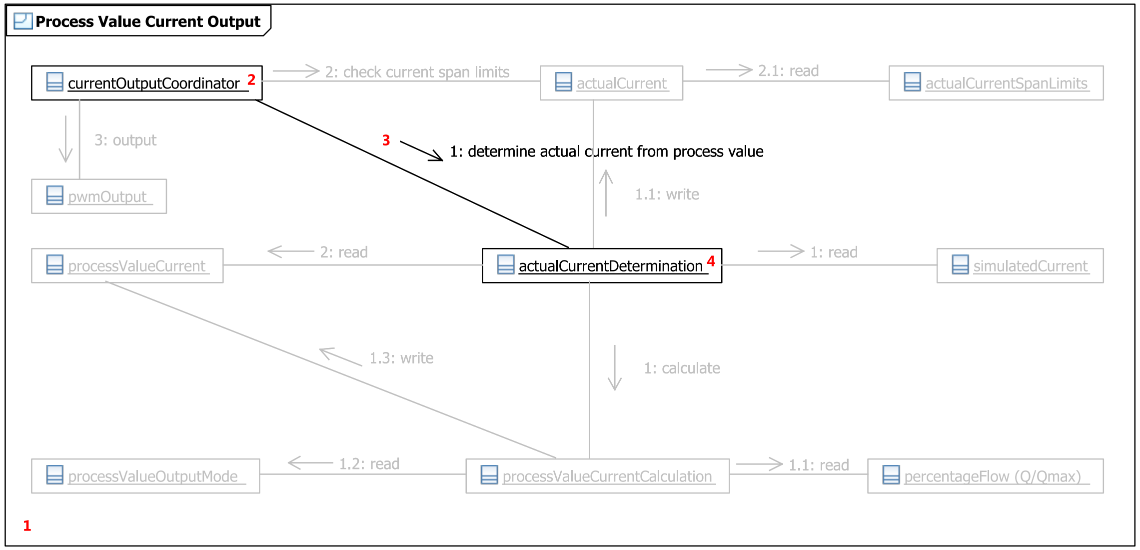
(a) Example Diagram

Fig. 41: Analysis UML-Model - Information Diagram



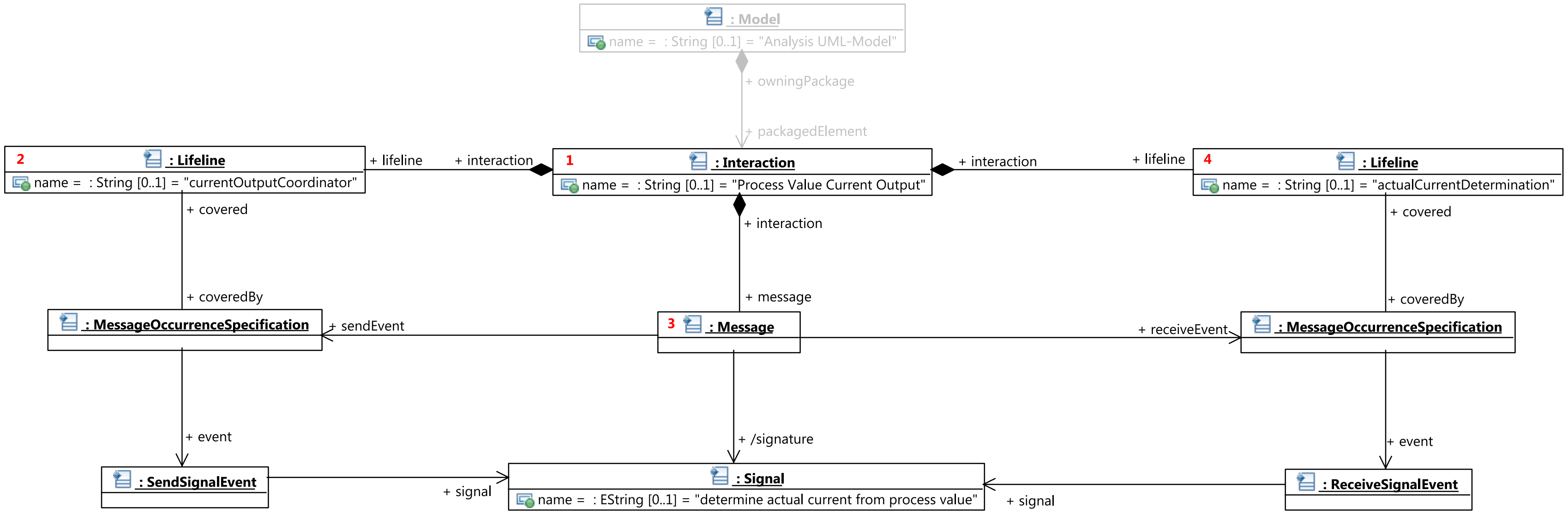
(b) Contributed Model Artefacts

Fig. 41: Analysis UML-Model - Information Diagram



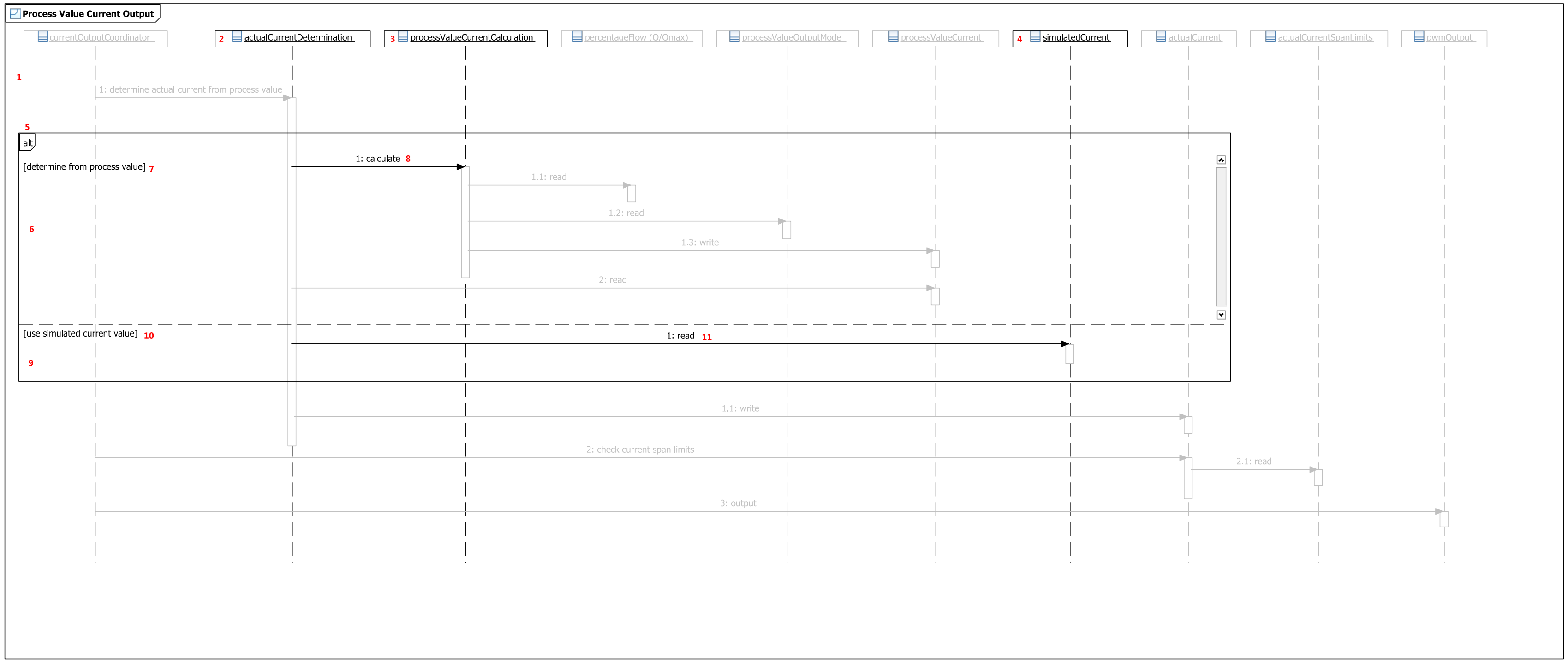
(a) Example Diagram

Fig. 42: Analysis UML-Model - Inter-Object Collaboration Diagram (Communication)



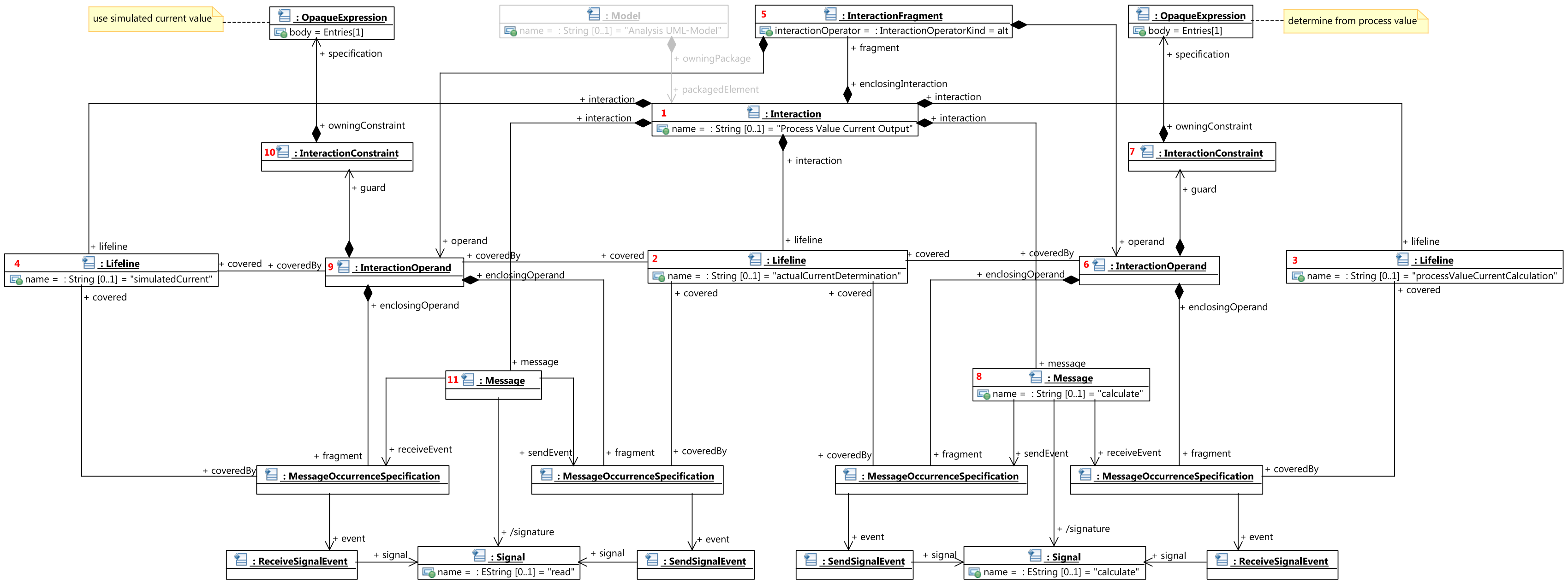
(b) Contributed Model Artefacts

Fig. 42: Analysis UML-Model - Inter-Object Collaboration Diagram (Communication)



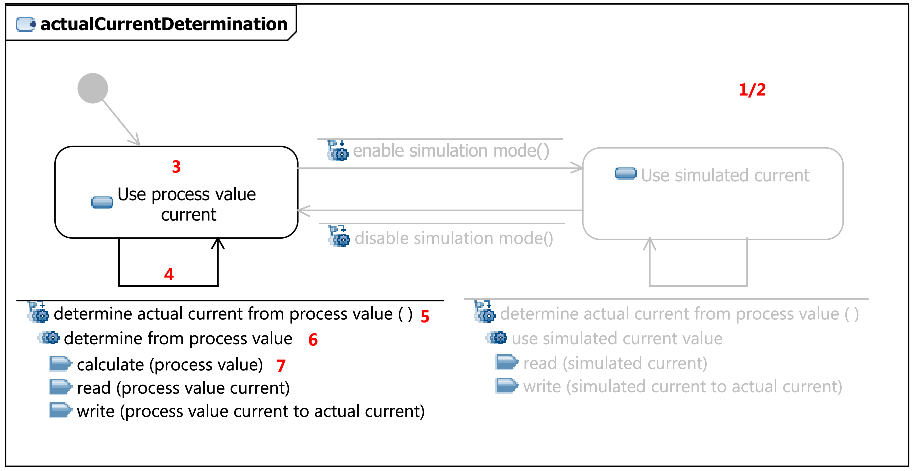
(a) Example Diagram

Fig. 43: Analysis UML-Model - Inter-Object Collaboration Diagram (Sequence)



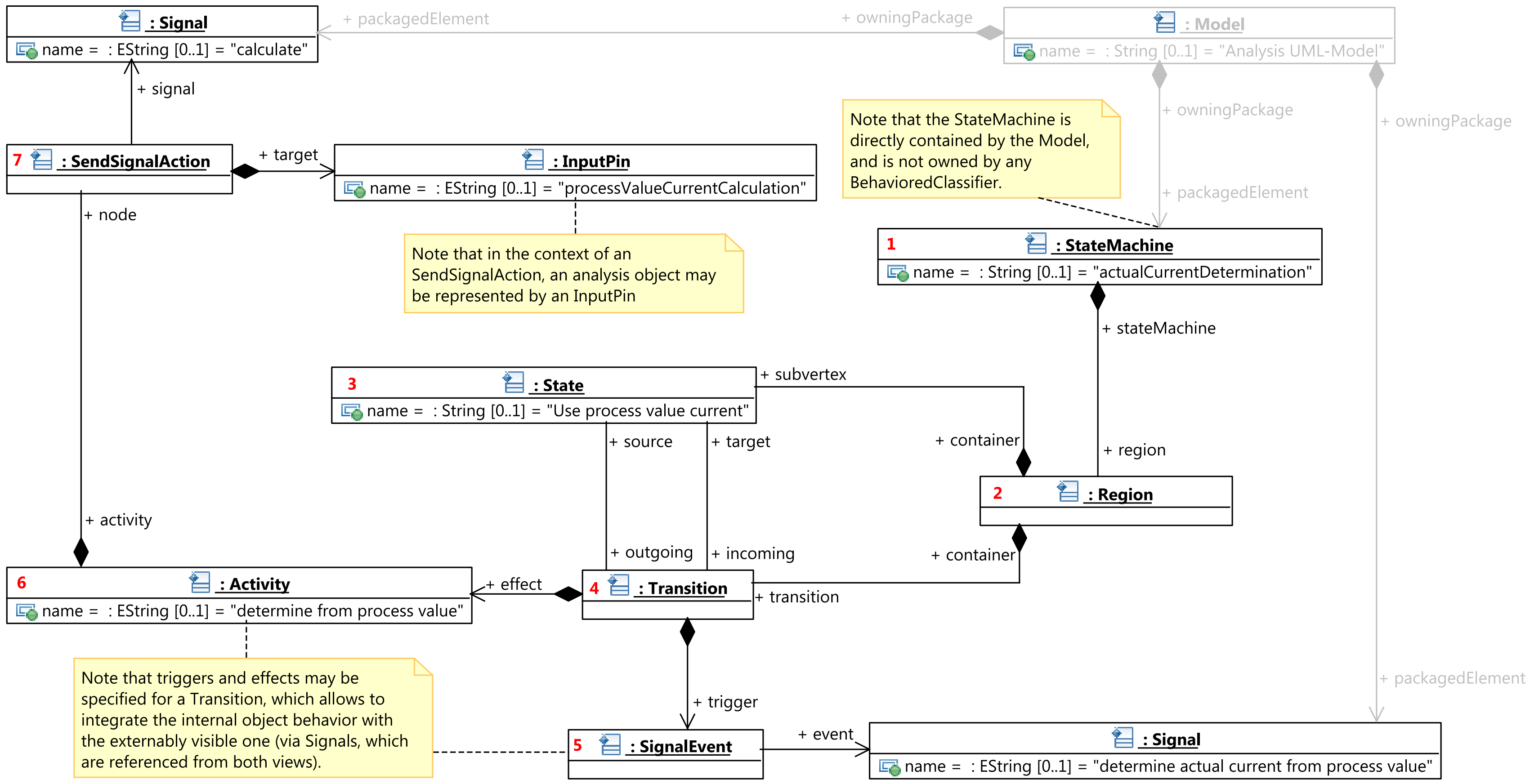
(b) Contributed Model Artefacts

Fig. 43: Analysis UML-Model - Inter-Object Collaboration Diagram (Sequence)



(a) Example Diagram

Fig. 44: Analysis UML-Model - Intra-Object Behavior Diagram



(b) Contributed Model Artefacts

Fig. 44: Analysis UML-Model - Intra-Object Behavior Diagram

A.3 Design UML-Model

The structural system architecture is basically manifested in terms of a system *Component* and subsystem *Components*. The aggregation of subsystems by the system component is realized in form of a composite structure, as contributed by the *Structural System Architecture Diagram* in terms of parts (i.e. *Properties*, being typed by the subsystem *Components*, being owned by the system *Component* and respective assembly connectors established between the parts (respectively their ports). The structural interfaces of each subsystem are modeled in terms of *Ports*, owned by the subsystem *Component*, as well as required and provided *Interfaces*, as defined via the *Initial* and *Consolidated Structural Subsystem Design Interface Diagrams*, the internal decomposition, as contributed via the *Initial* and *Consolidated Structural System Design Diagrams*, is manifested in terms of owned *Properties* (i.e. parts), representing the aggregated design objects, which are related to each other and to the *Ports* of the subsystem via assembly respectively delegation *Connectors*.

The behavioral system architecture in turn is defined in terms of (protocol) *State Machines*, defined for the *Ports* of the subsystems, or for the subsystem *Component* itself, added via the *Initial* and *Consolidated Behavioral Subsystem Interface Design Diagrams*, as well as *Interactions*, depicting the inter-subsystem communication in terms of *Lifelines*, representing subsystem (instances), as well as *Messages*, being exchanged between them.

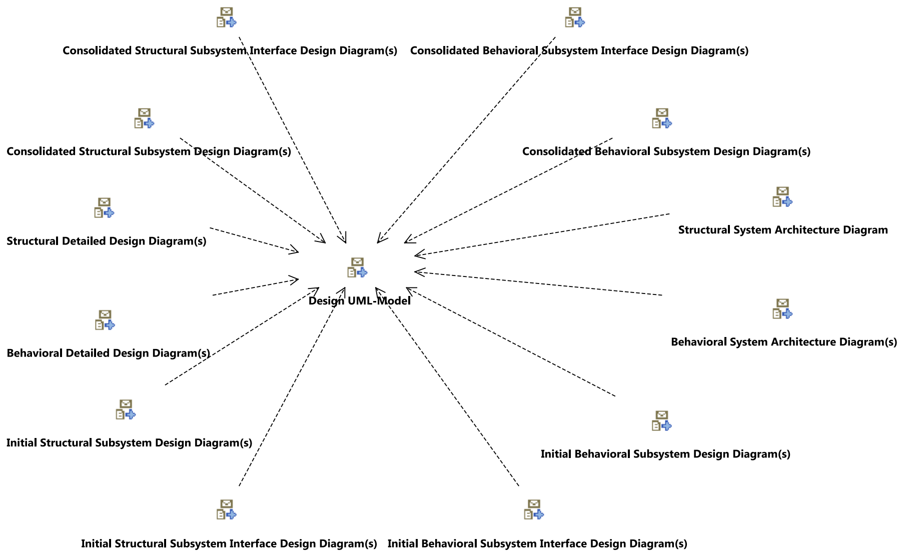
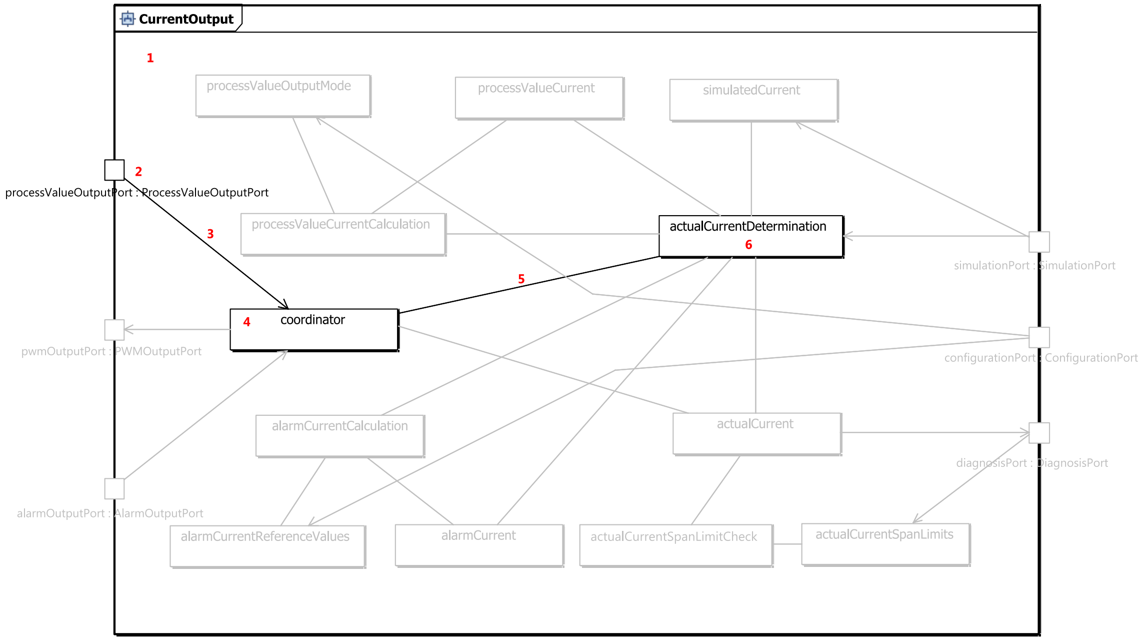


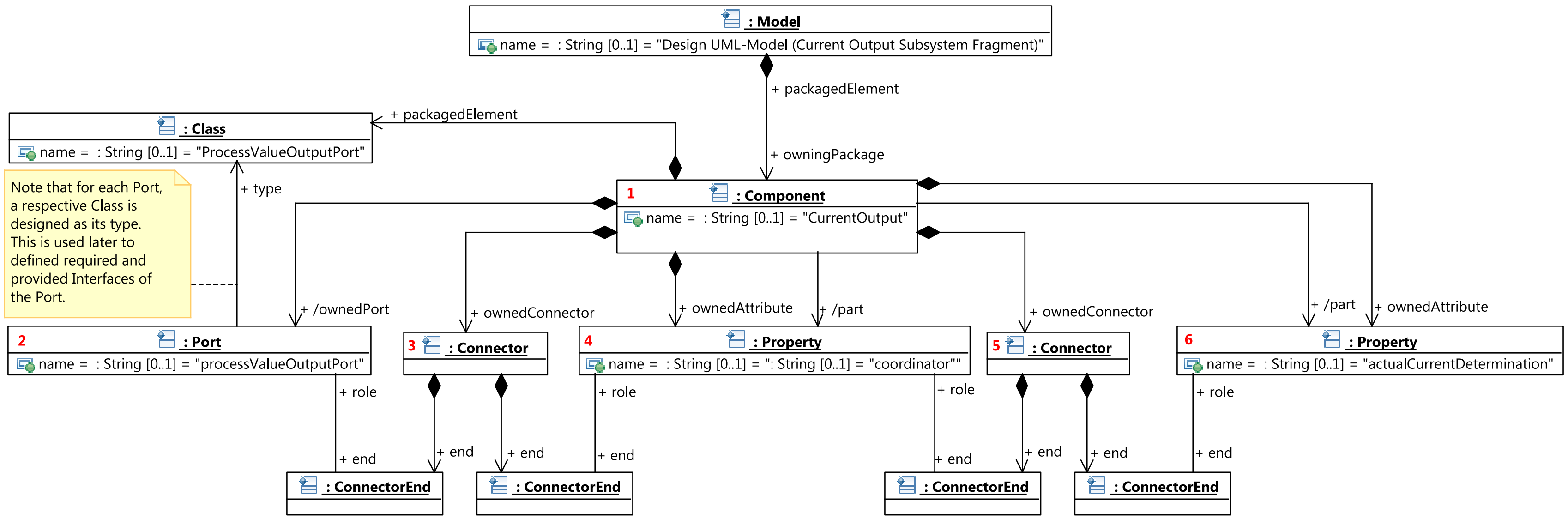
Fig. 45: Design UML-Model and related UML diagrams

As depicted by Figure 45, the final contribution to the *Design UML-Model* is added via the *Class Design Diagrams* in terms of classes , designed as types of the *Properties*, composed by the subsystem *Component*, as well as *Associations*, used to type the *Connectors*, which are as well composed.



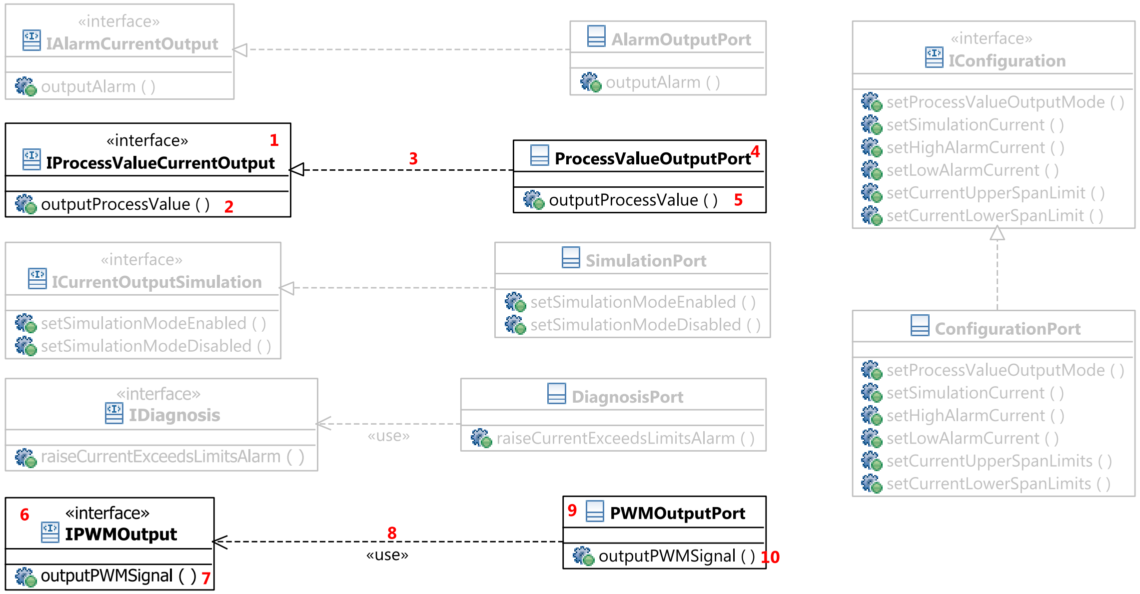
(a) Example Diagram

Fig. 46: Design UML-Model - Initial Structural Subsystem Design Diagram



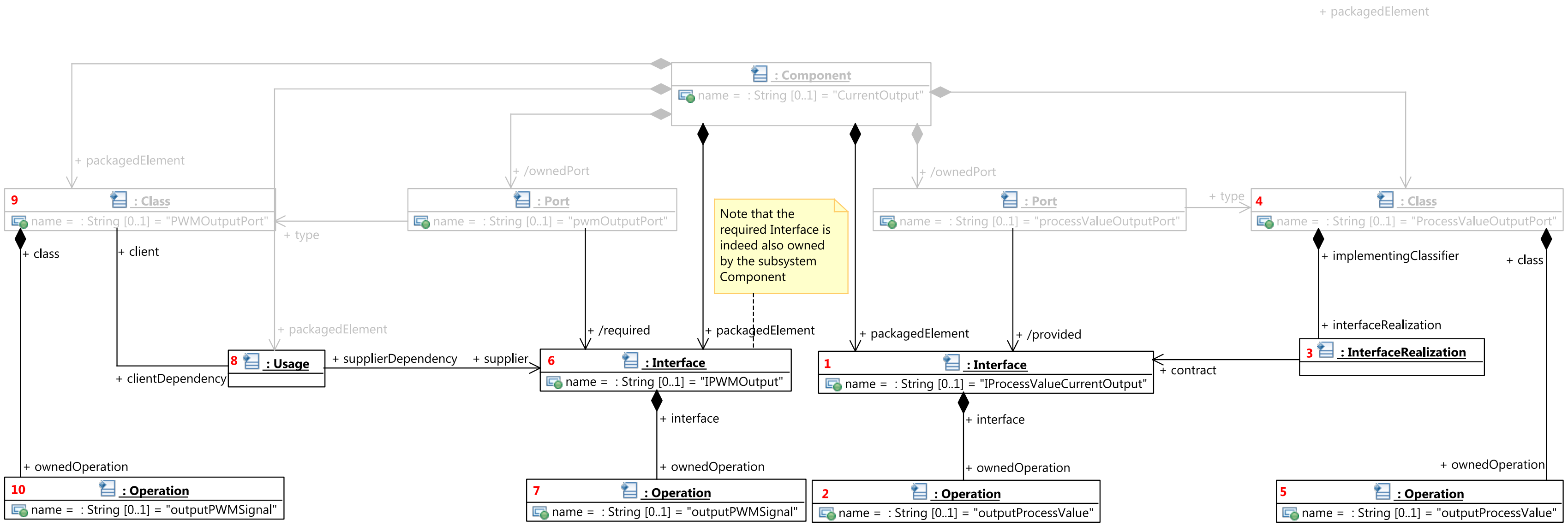
(b) Contributed Model Artefacts

Fig. 46: Design UML-Model - Initial Structural Subsystem Design Diagram



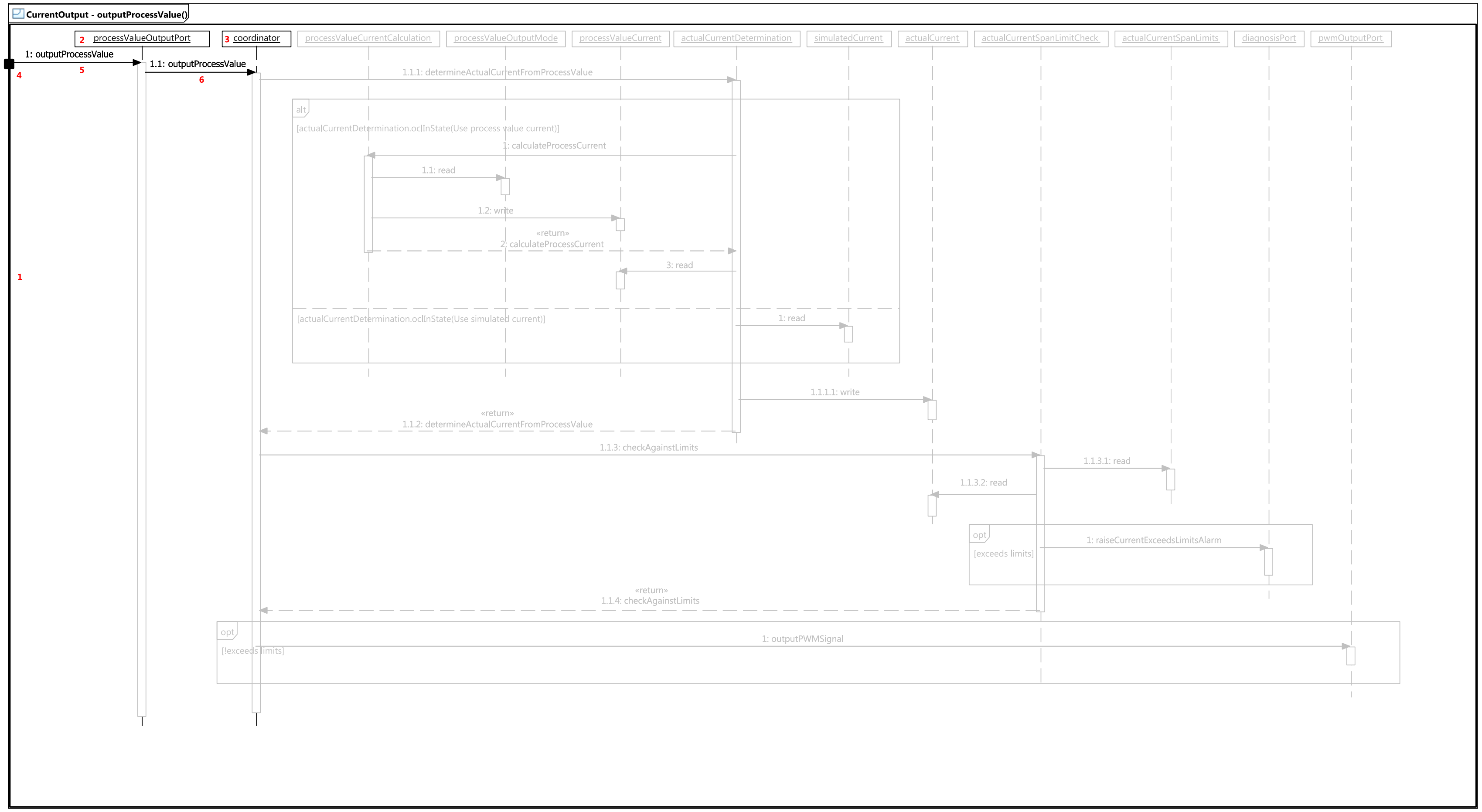
(a) Example Diagram

Fig. 47: Design UML-Model - Initial Structural Subsystem Interface Design Diagram



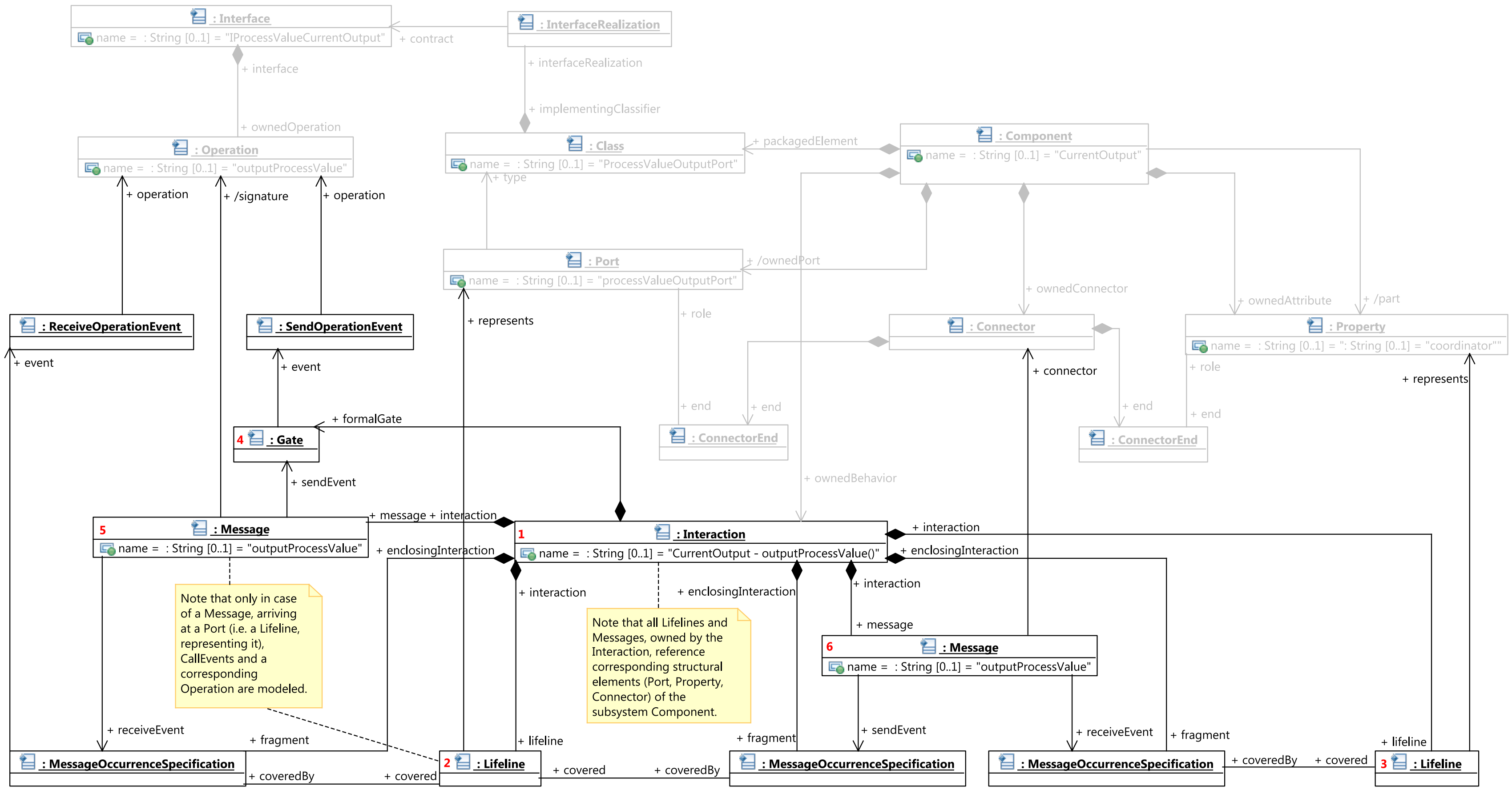
(b) Contributed Model Artefacts

Fig. 47: Design UML-Model - Initial Structural Subsystem Interface Design Diagram



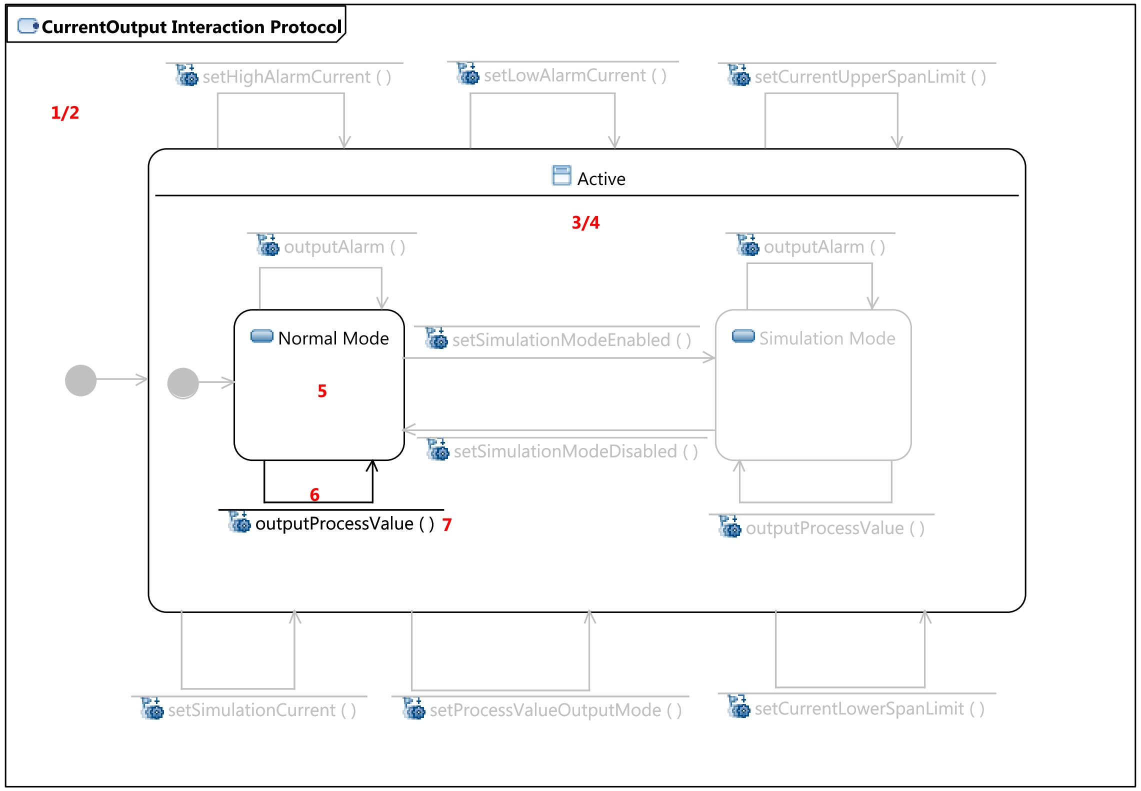
(a) Example Diagram

Fig. 48: Design UML-Model - Initial Behavioral Subsystem Design Diagram



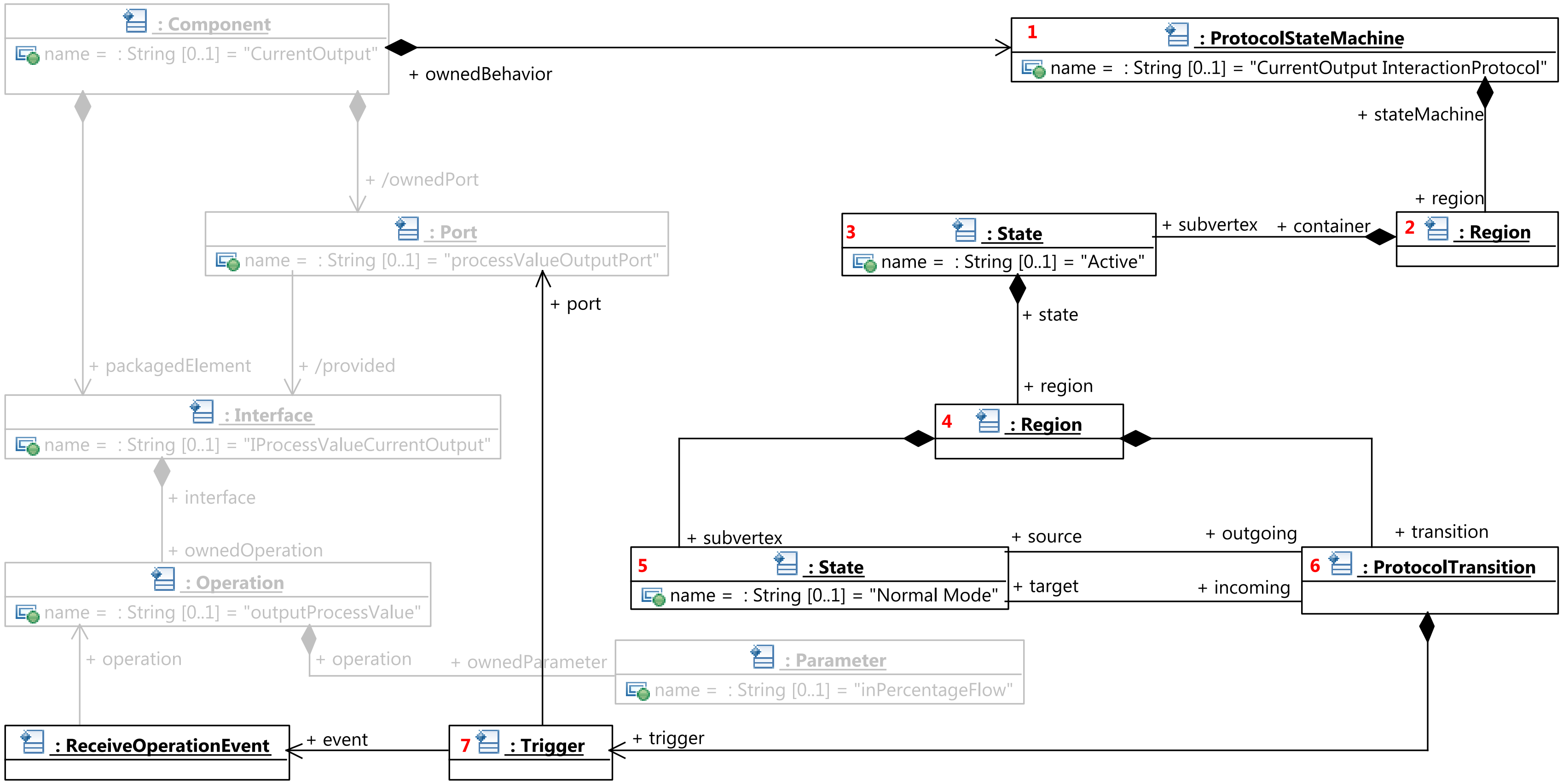
(b) Contributed Model Artefacts

Fig. 48: Design UML-Model - Initial Behavioral Subsystem Design Diagram



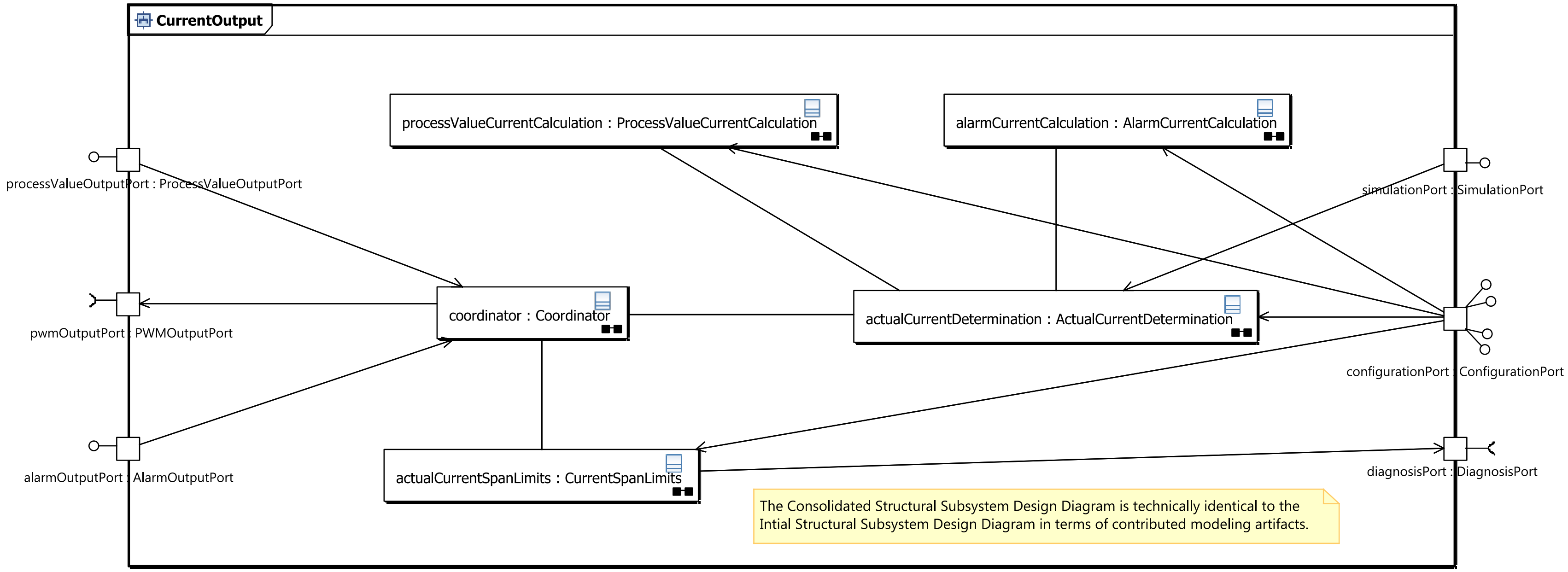
(a) Example Diagram

Fig. 49: Design UML-Model - Initial Behavioral Subsystem Interface Design Diagram



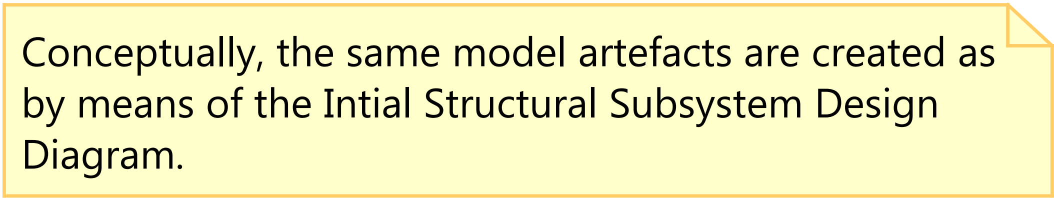
(b) Contributed Model Artefacts

Fig. 49: Design UML-Model - Initial Behavioral Subsystem Interface Design Diagram



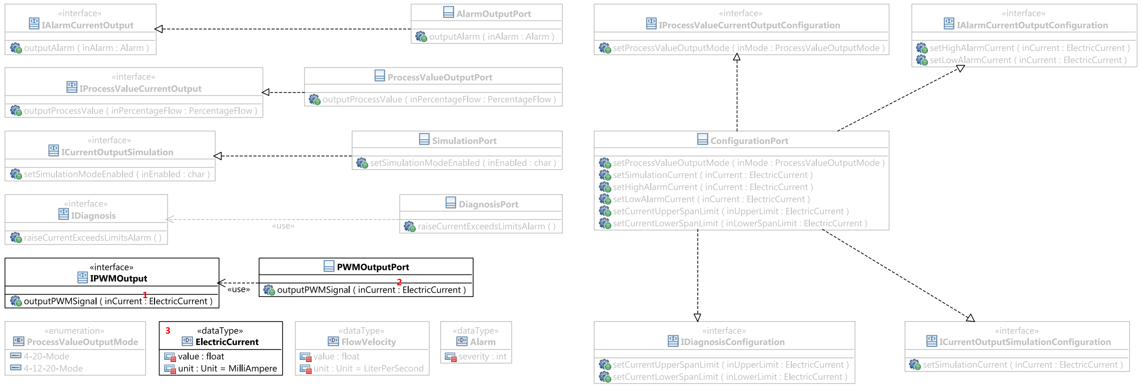
(a) Example Diagram

Fig. 50: Design UML-Model - Consolidated Structural Subsystem Design Diagram



(b) Contributed Model Artefacts

Fig. 50: Design UML-Model - Consolidated Structural Subsystem Design Diagram



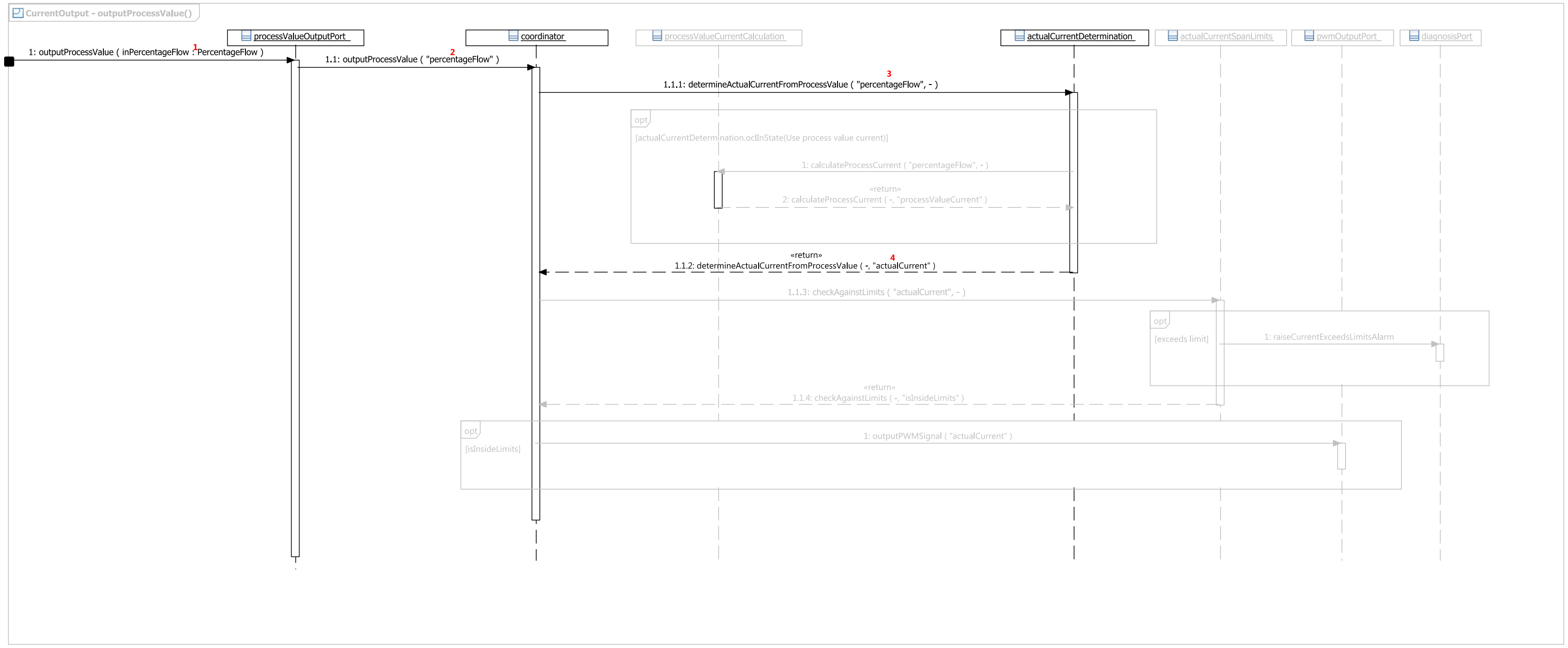
(a) Example Diagram

Fig. 51: Design UML-Model - Consolidated Structural Subsystem Interface Design Diagram

Conceptually the same as Initial Behavioral Subsystem Interface Design Diagram, only that parameters are now shown, which were actually contributed via the Consolidated Structural Interface Design Diagram.

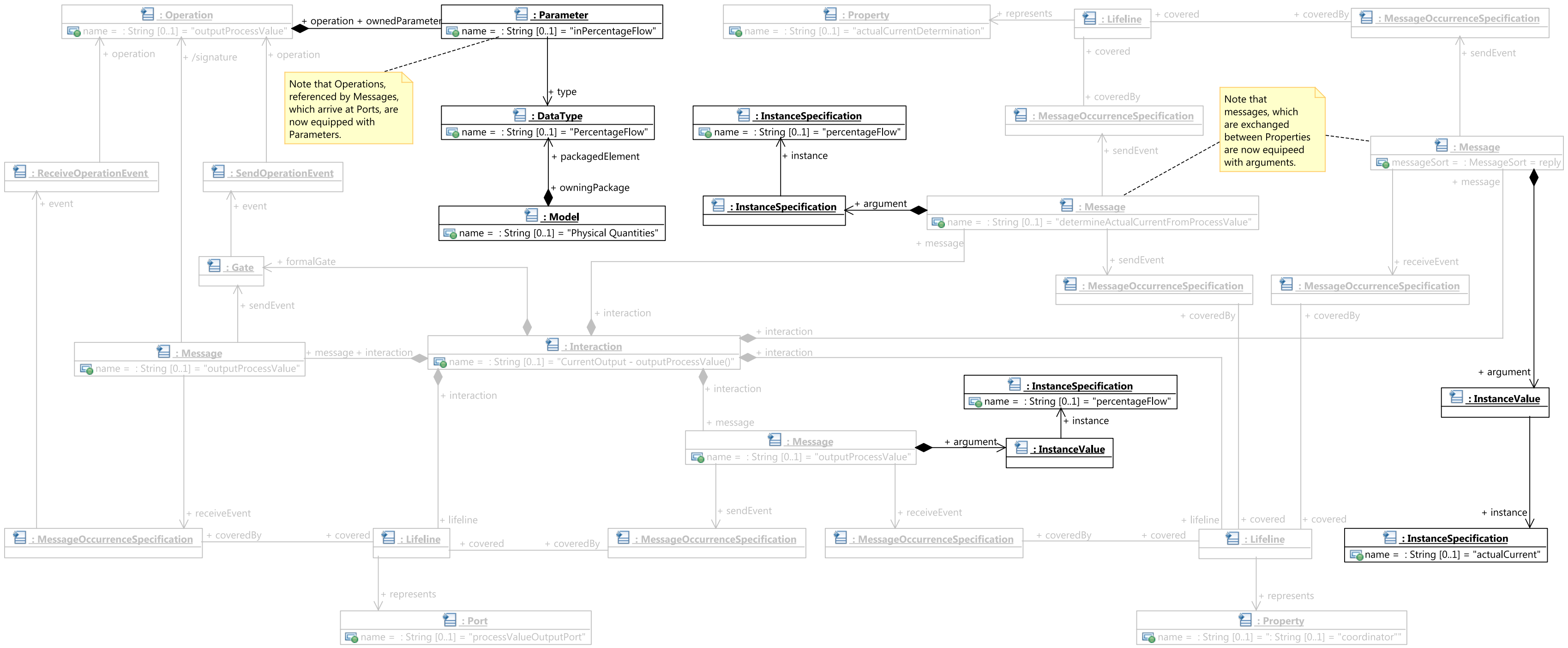
(b) Contributed Model Artefacts

Fig. 51: Design UML-Model - Consolidated Structural Subsystem Interface Design Diagram



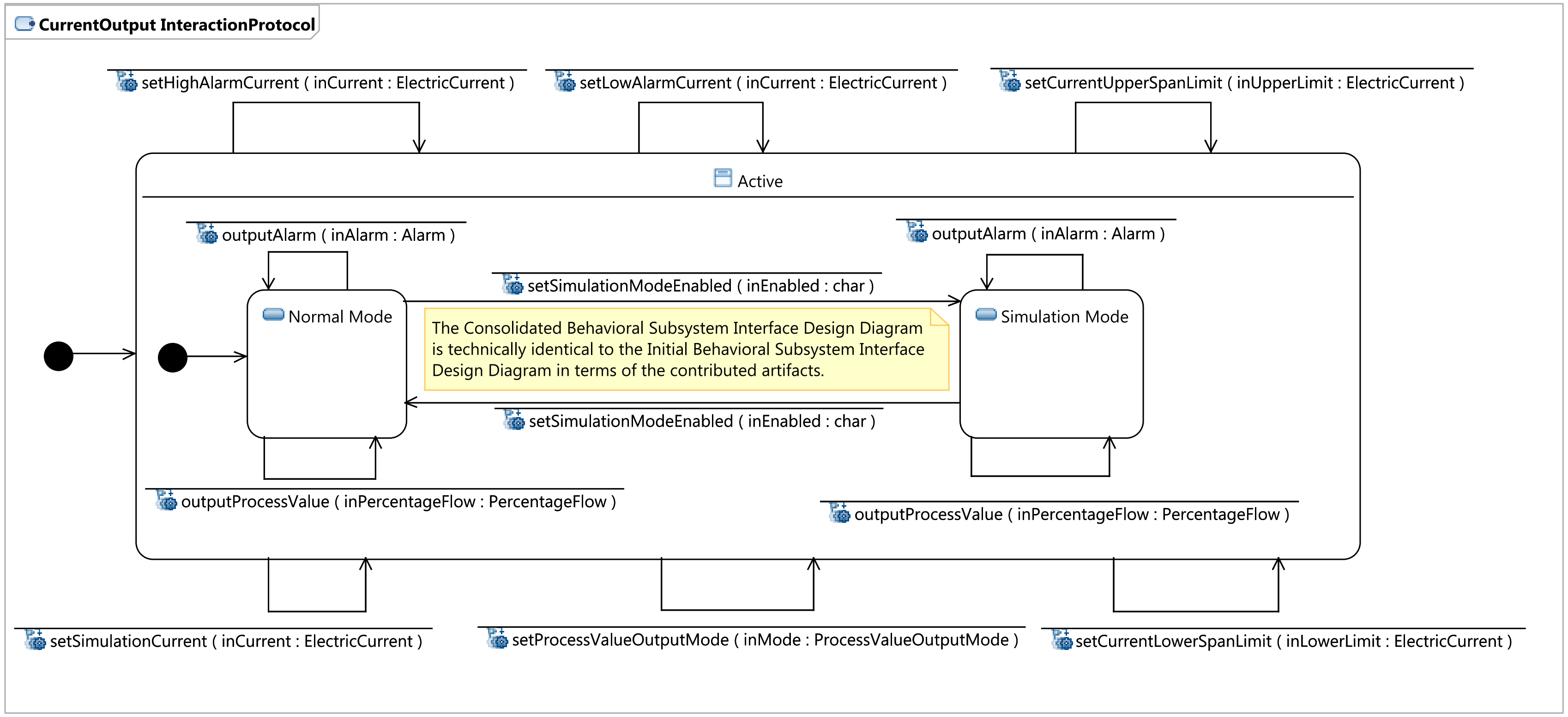
(a) Example Diagram

Fig. 52: Design UML-Model - Consolidated Behavioral Subsystem Design Diagram



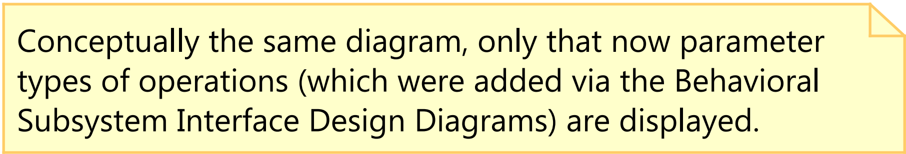
(b) Contributed Model Artefacts

Fig. 52: Design UML-Model - Consolidated Behavioral Subsystem Design Diagram



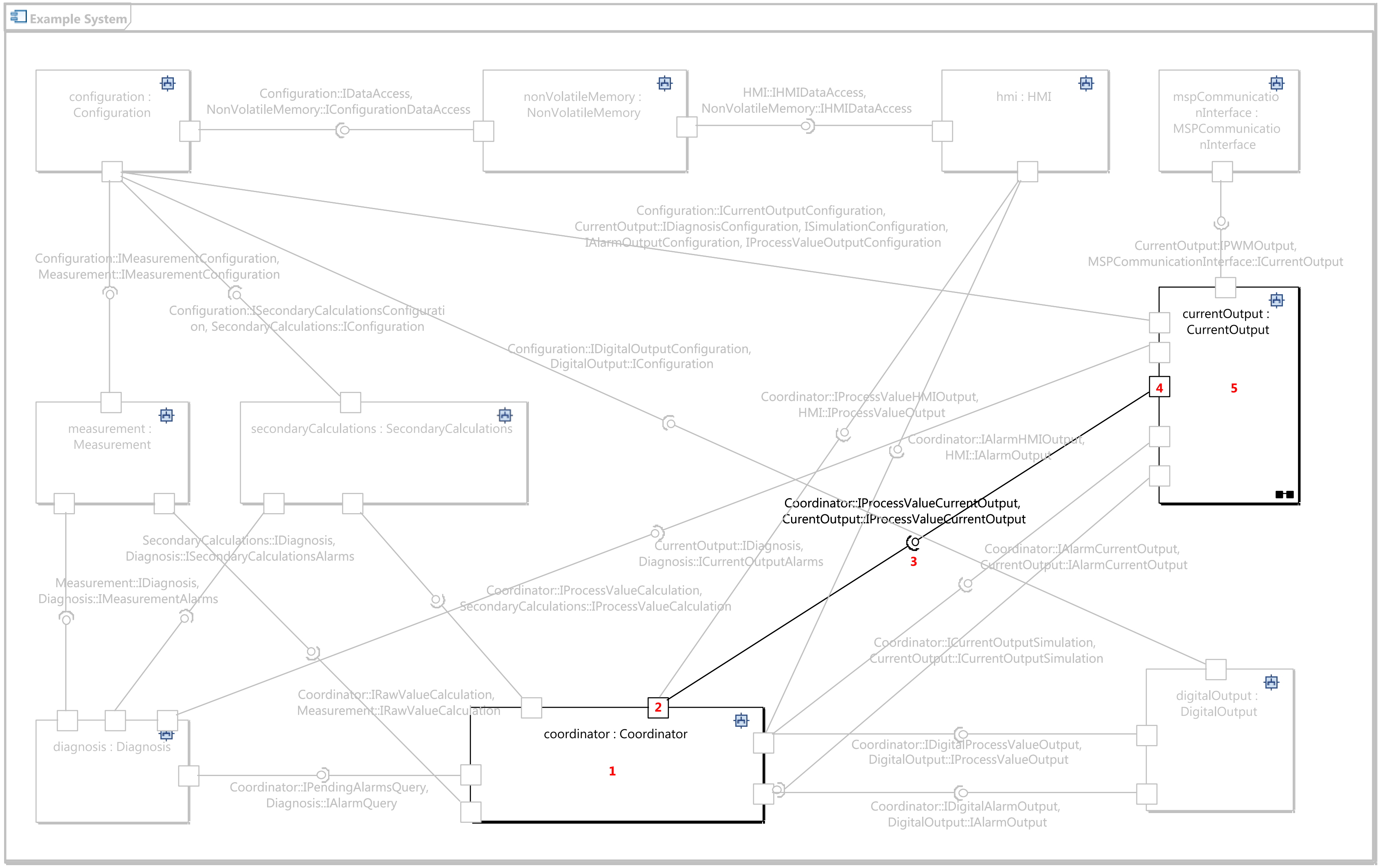
(a) Example Diagram

Fig. 53: Design UML-Model - Consolidated Behavioral Subsystem Interface Design Diagram



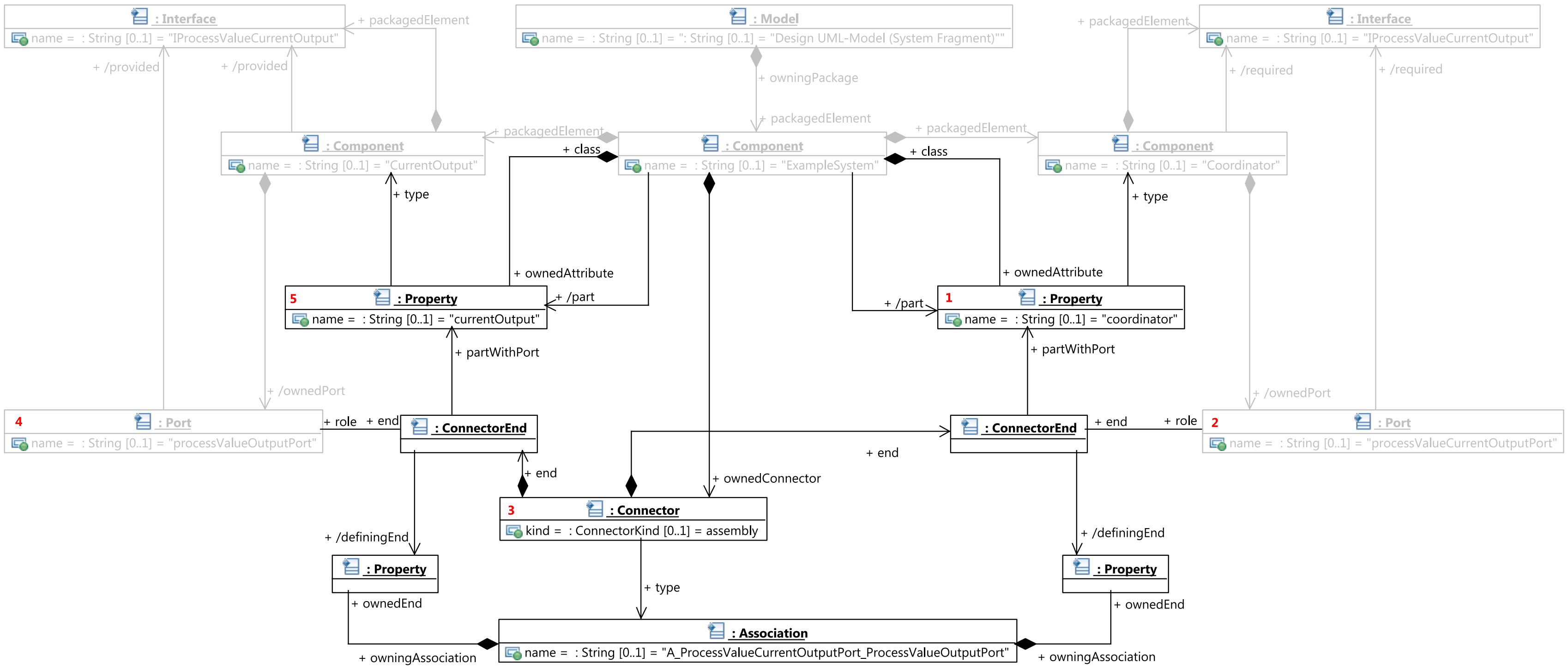
(b) Contributed Model Artefacts

Fig. 53: Design UML-Model - Consolidated Behavioral Subsystem Interface Design Diagram



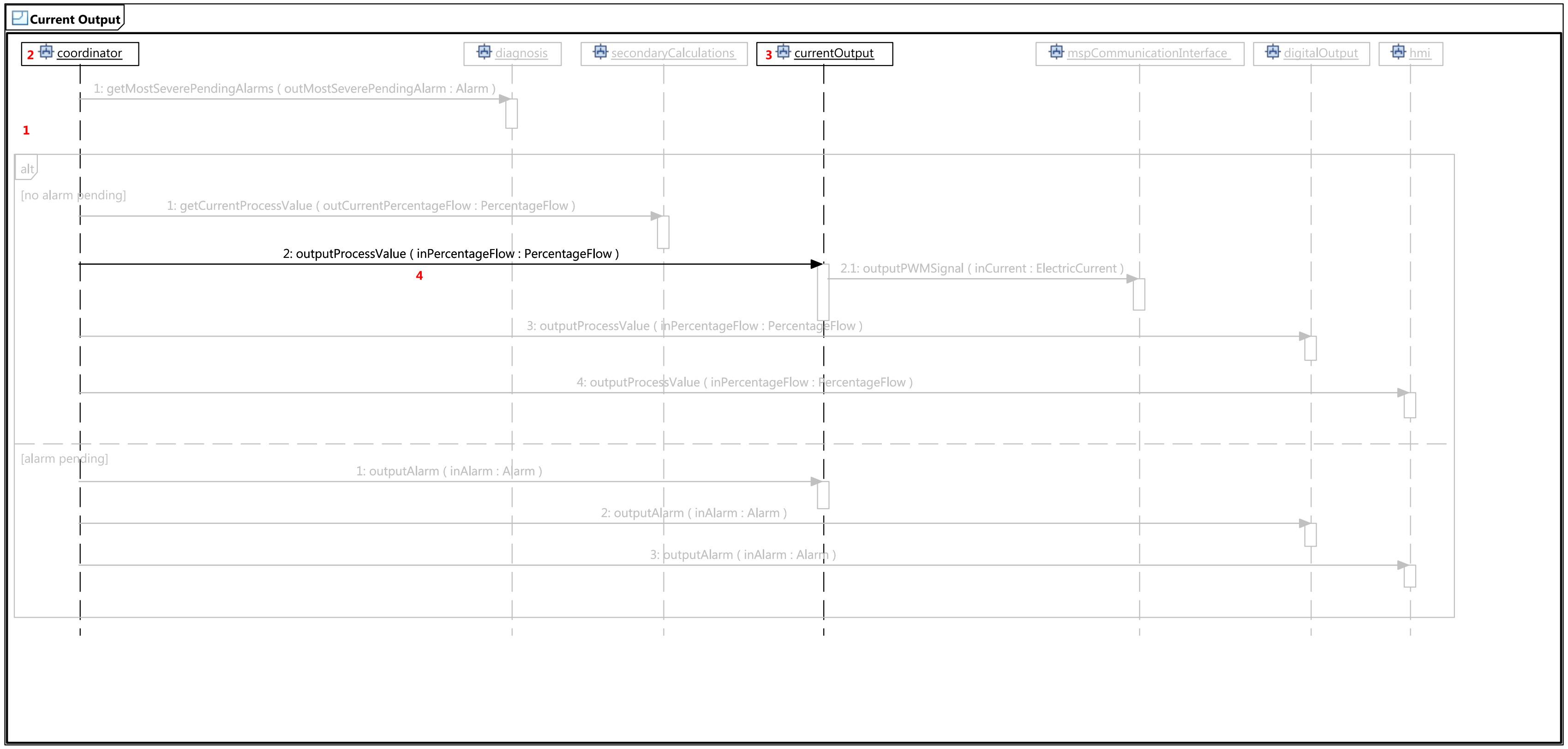
(a) Example Diagram

Fig. 54: Design UML-Model - Structural System Architecture Diagram



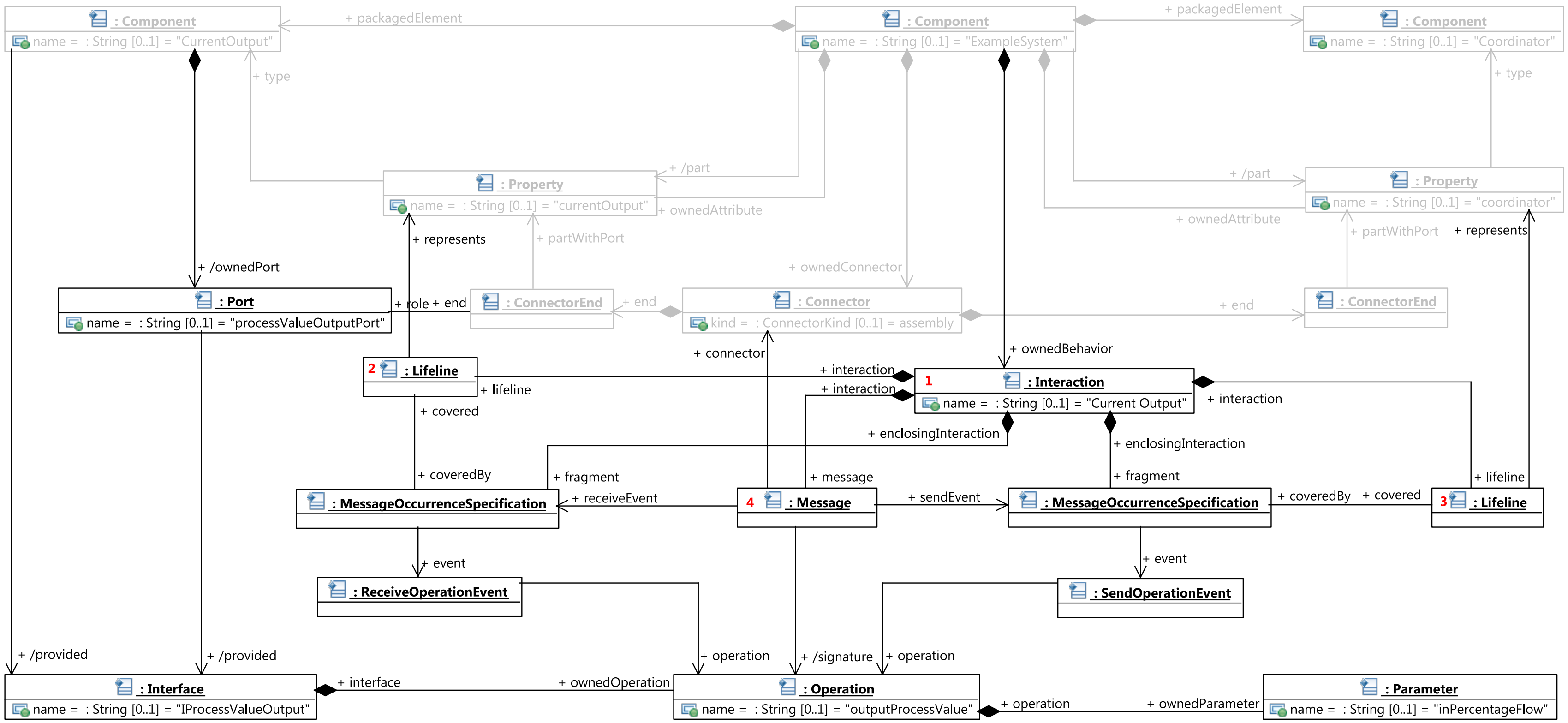
(b) Contributed Model Artefacts

Fig. 54: Design UML-Model - Structural System Architecture Diagram



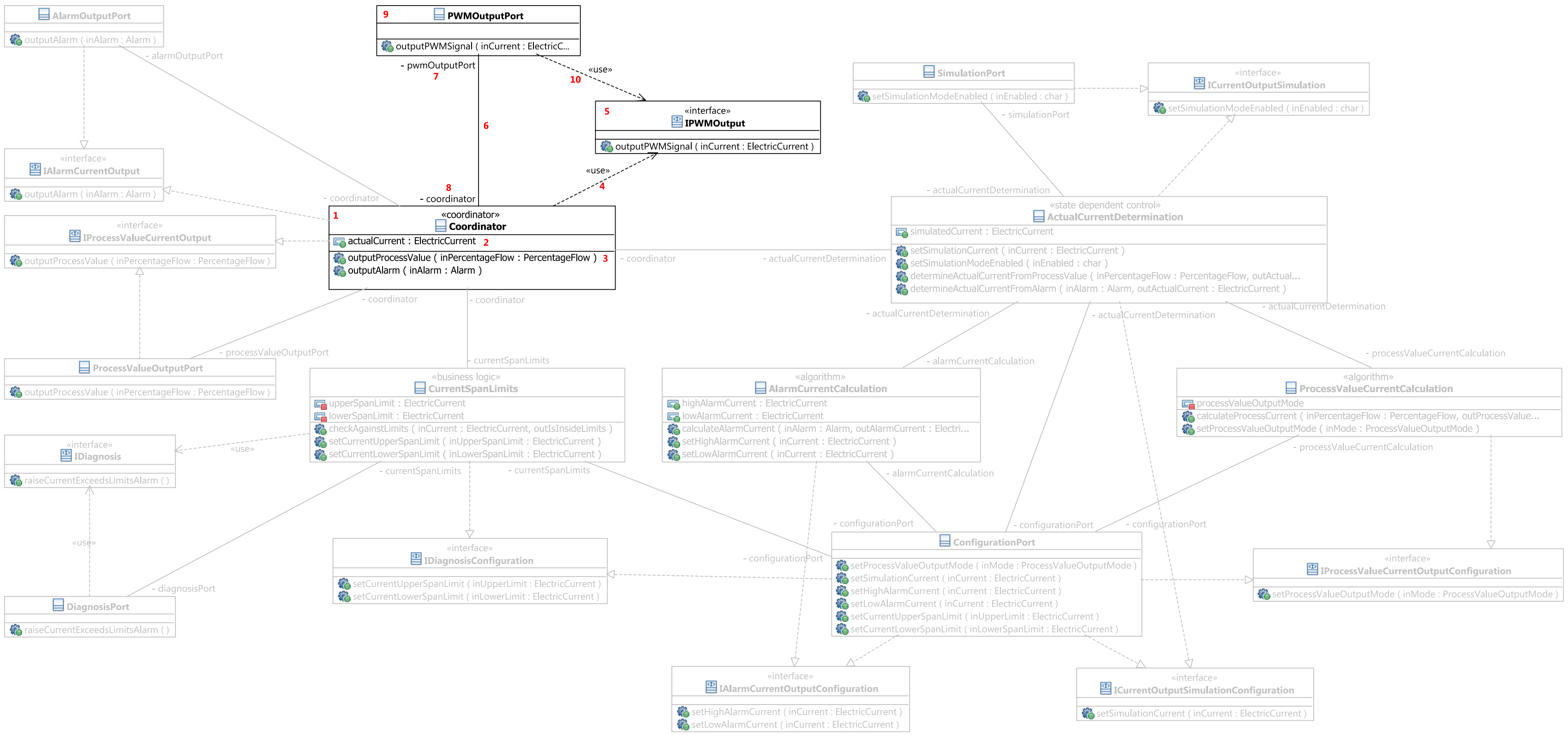
(a) Example Diagram

Fig. 55: Design UML-Model - Behavioral System Architecture Diagram



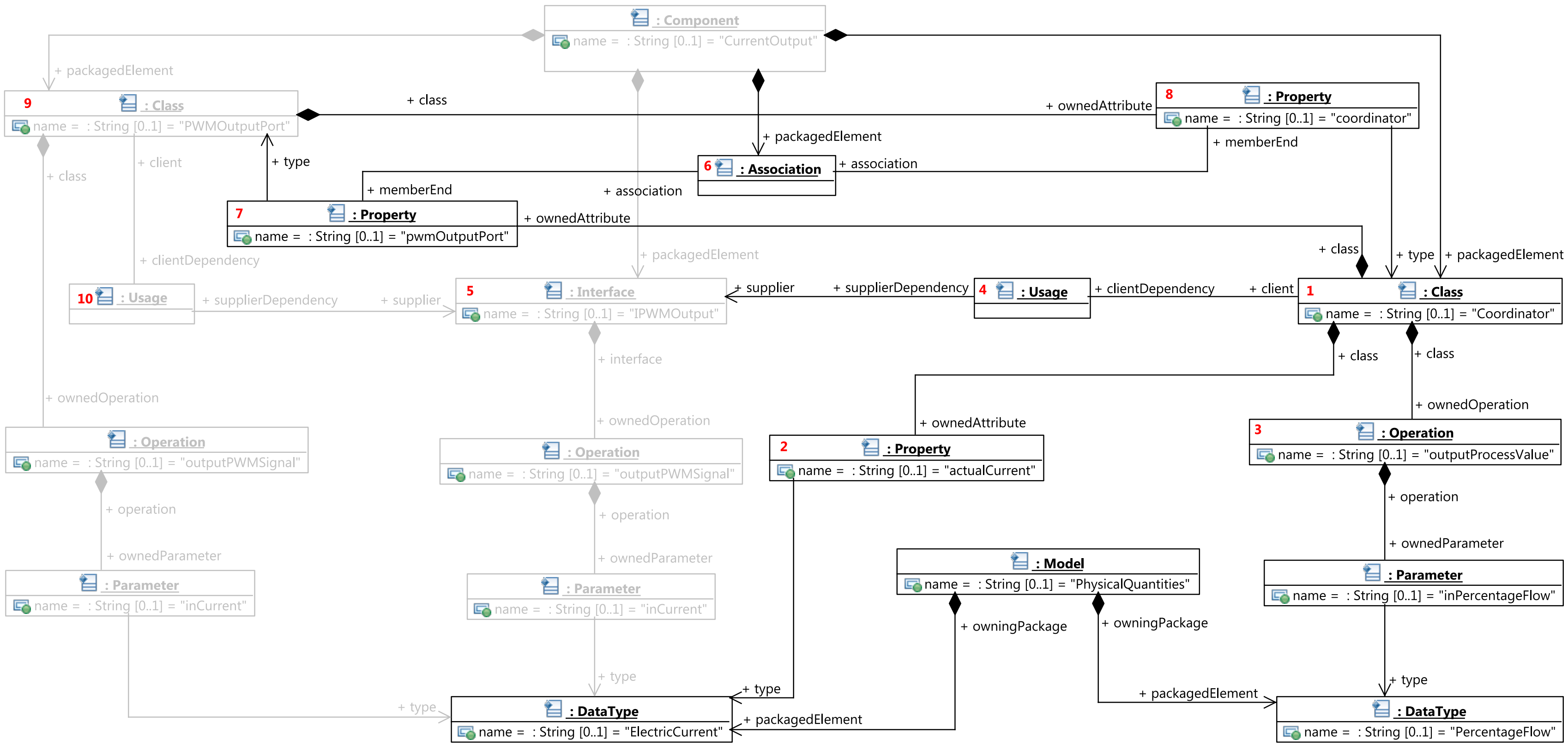
(b) Contributed Model Artefacts

Fig. 55: Design UML-Model - Behavioral System Architecture Diagram



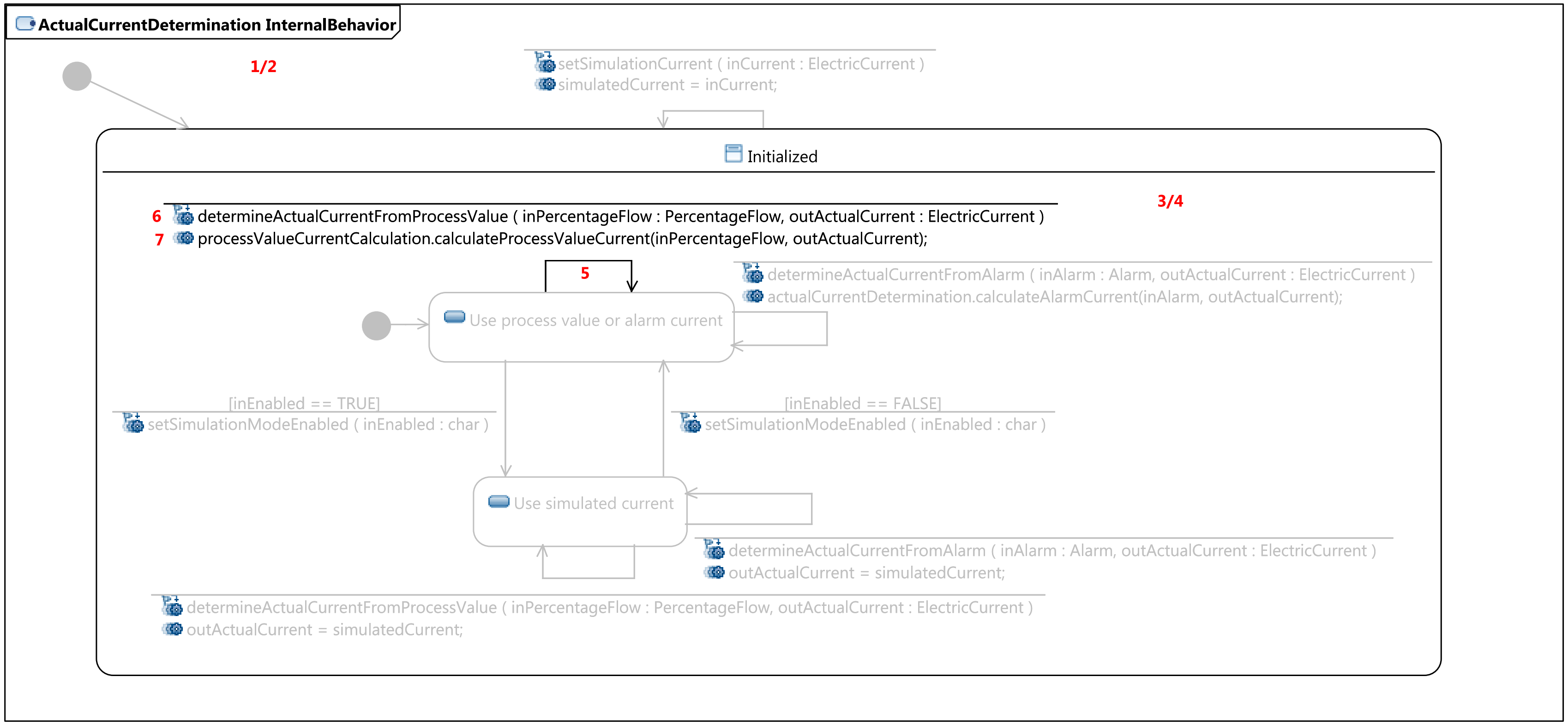
(a) Example Diagram

Fig. 56: Design UML-Model - Structural Detailed Design Diagram



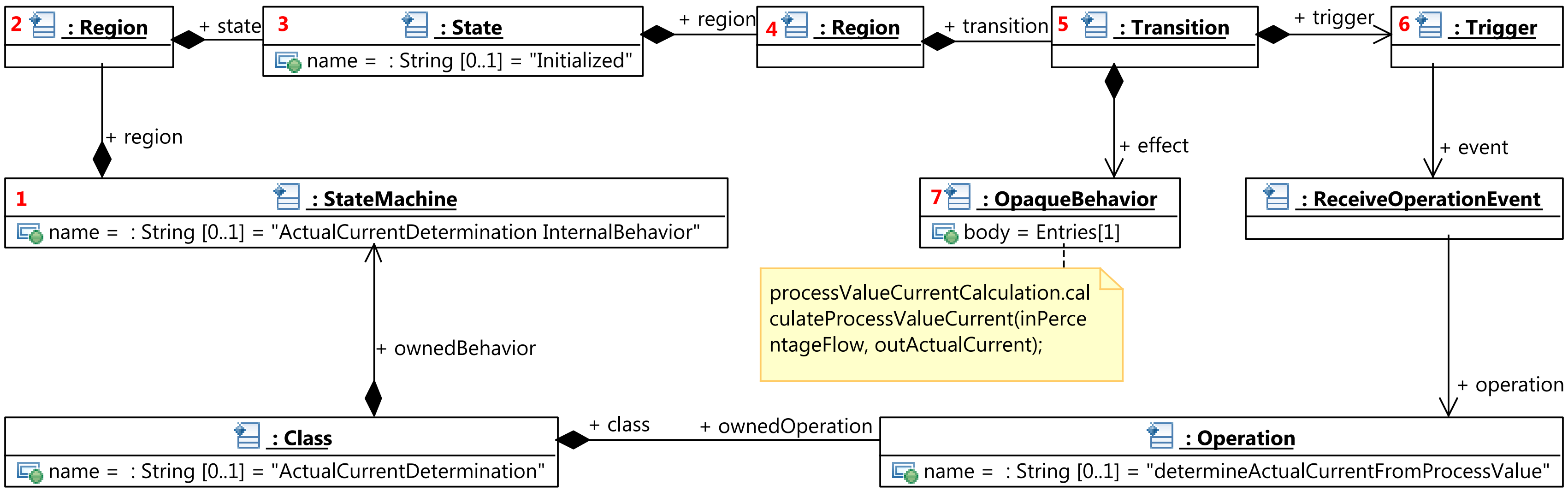
(b) Contributed Model Artefacts

Fig. 56: Design UML-Model - Structural Detailed Design Diagram



(a) Example Diagram

Fig. 57: Design UML-Model - Behavioral Detailed Design Diagram



(b) Contributed Model Artefacts

Fig. 57: Design UML-Model - Behavioral Detailed Design Diagram

B MeDUSA UML-Profiles

The *Actor Taxonomy* and Object Taxonomy introduced in Sections 2.1 and 2.2 are very essential for MeDUSA, as they support the identification of Actors and Objects during *Requirements Modeling* and *Analysis Modeling*. Their use thus has to be properly reflected within the *Requirements* and *Analysis UML-Model*, and of course within the *Design UML-Model* as well (as analysis objects are transferred into corresponding design objects).

It is thus necessary to enrich the UML with the needed expressiveness. This can be done by means of the built-in Profile extension mechanism (compare [OMG07b]). We thus define three Profiles, as outlined by Figures 58, 59, and 60, one for each MeDUSA UML model, offering respective Stereotypes, which may be applied to standard UML model elements to denote their classification according to the MeDUSA taxonomies.

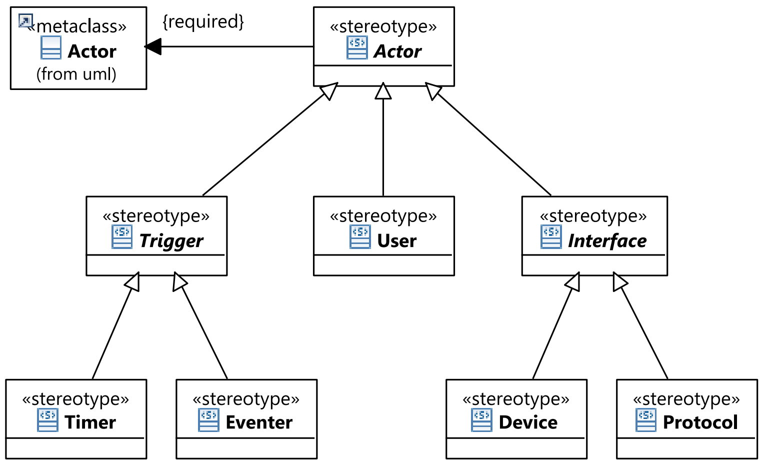


Fig. 58: MeDUSA Requirements UML-Profile

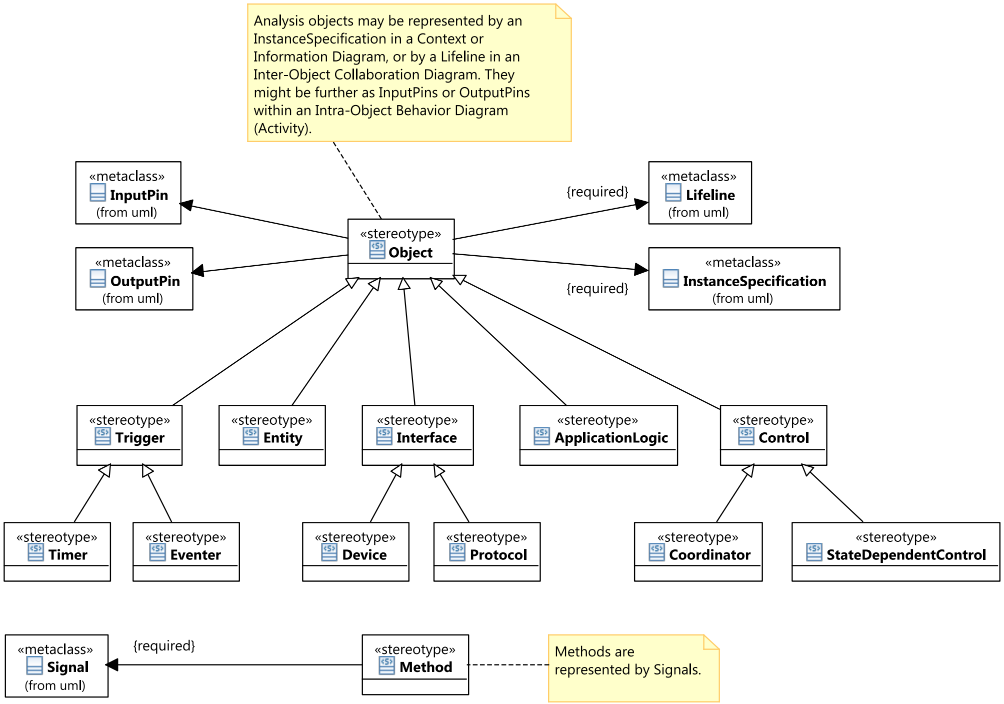


Fig. 59: MeDUSA Analysis UML-Profile

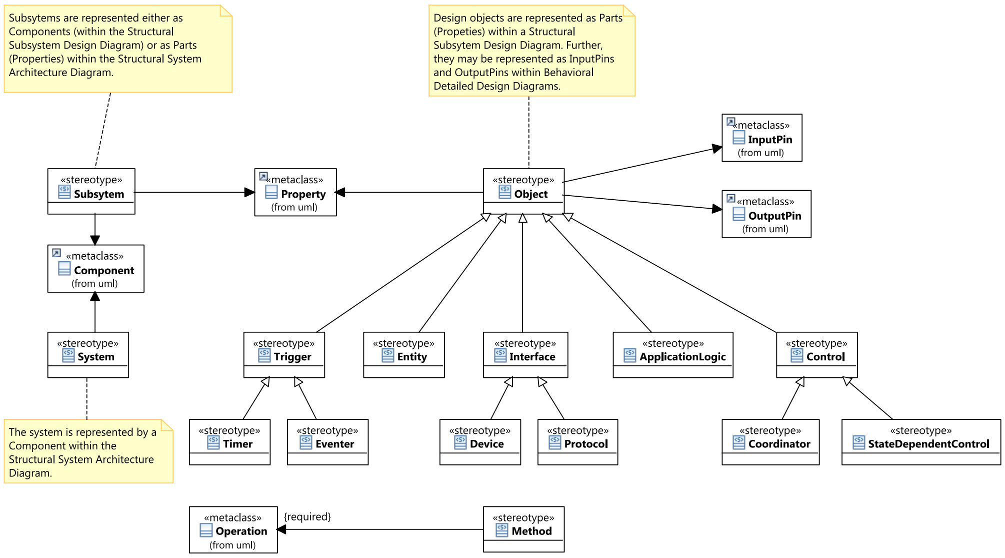


Fig. 60: MeDUSA Design UML-Profile

C MeDUSA ANSI-C Code Generation Schema

What has already been stressed by the fact that MeDUSA was explicitly denoted to be a *construction* rather than a mere *design* method is that a seamless transition between *Architectural Design* and *Implementation* is regarded to be of outstanding importance.

As a matter of fact, this documentation has to include a detailed specification on how ANSI-C conformant source code may be generated from MeDUSA *Design UML-Models*. A general applicable transformation strategy to transform arbitrary UML models into ANSI-C code would be - because of the complexity of the UML - hard to achieve and rather impractical to apply. Based on the detailed specification of a MeDUSA *Design UML-Model*, as it is provided in Appendix A, a customized and tailored code generation strategy may however be provided.

Such a generation may be conceptually and technically split into two parts. The first is the generation of a folder structure to match the coarse-grained structure of a *Design UML-Model*, which manifests itself in the subsystem Components being comprised. This is pretty much straight-forward and will thus not be covered here. The second is a transformation, which is basically oriented along the analogy between the Classifier concept of the UML and the corresponding type concept in the ANSI-C language. That is, UML Classifiers are transformed into corresponding ANSI-C types (and related functions). The concrete transformation in this context depends on the type of Classifier being transformed (i.e. Class, State Machine, Activity, ...) as well as on the concrete usage of the Classifier with a MeDUSA *Design UML-Model* (i.e. a Class may be differently transformed if being used as type of a Part within the internal decomposition of a subsystem, or if it is used as type of a Port). As a starting point, it will thus be outlined in the following, which Classifiers of the UML are used within a MeDUSA *Design UML-Model* and in which concrete usage scenarios. Subsequently, general aspects of the transformation are addressed before the transformation is then outlined for each individual Classifier in each respective situation it is employed in.

C.1 Usage of Classifiers within a MeDUSA Design UML-Model

Figure 61 outlines the hierarchy of Classifiers, as it is defined by the UML [4]. The picture further highlights those Classifiers, which are used in the context of a MeDUSA *Design UML-Model*.

[4] It also shows all *merge increments* of the respective Classifiers. They are a result of the *Package Merge* mechanism, used within the definition of the UML Superstructure [OMG07b]. As stated there, a *"package merge is a directed relationship between two packages that indicates that the contents of the two packages are to be combined. It is very similar to Generalization in the sense that the source element conceptually adds the characteristics of the target element to its own characteristics resulting in an element that combines the characteristics of both. [...] Conceptually, a package merge can be viewed as an operation that takes the contents of two packages and produces a new package that combines the contents of the packages involved in the merge. In terms of model semantics, there is no difference between a model with explicit package merges, and a model in which all the merges have been performed."*

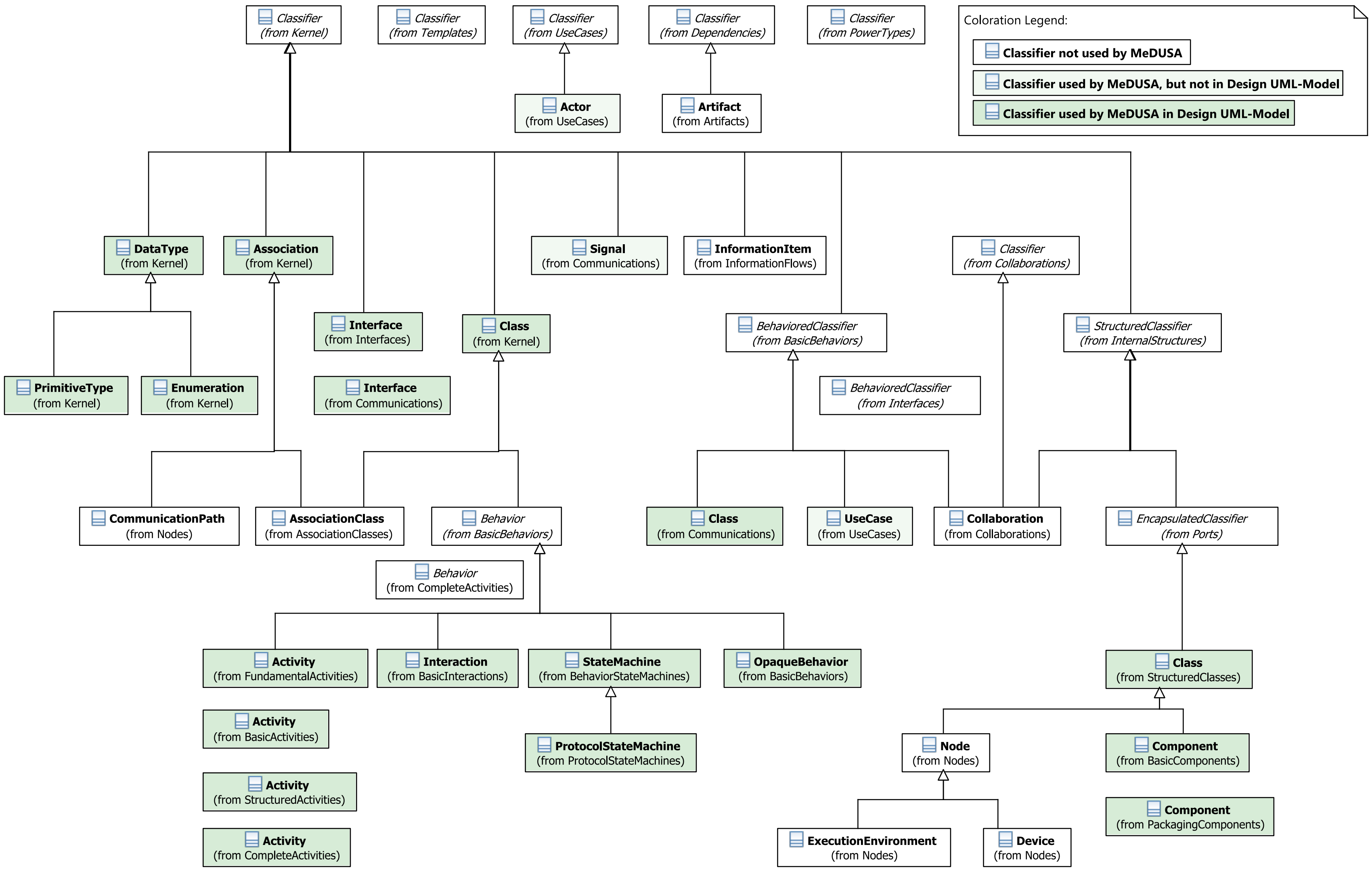


Fig. 61: UML Classifiers Hierarchy, according to [OMG07b]

Out of these Classifiers, the concrete i.e. non abstract ones, which actually have to be transferred into respective ANSI-C equivalents, are the ones listed in Table 2. The table further outlines their respective usages within a MeDUSA *Design UML-Model*.

Component	– Specification of the system, singleton Structured and Encapsulated Classifier with internal structure and explicit external interfaces, directly contained by the system model fragment – Specification of a subsystem, singleton Structured and Encapsulated Classifier with internal structure and explicit external interfaces, directly contained by the subsystem model fragment, and as Type of a Part, representing a subsystem (instance) in the internal decomposition of the system Component
(Kernel) Class	– Type of a Part, representing an object in the internal decomposition of a subsystem Component – Type of Port, to specify (part of) the external interface of a subsystem Component – Type of an Attribute or a Operation Parameter
Interface	– Exposed required or provided Interface of a Port, i.e. realized or used interface of a Port's typing (Kernel) Class
Association	– Type of a Connector in the internal decomposition of a subsystem Component, i.e. between (Kernel) Classes used as types of internally composed Parts and Ports – Type of a Connector in the decomposition of the system Component
Primitive Type, Enumeration, Data Type	– Type of an Attribute or a Operation Parameter
Protocol State Machine	– Owned Behavior of a subsystem Component or a Port's typing (Kernel) Class, as behavioral specification of (parts of) the externally visible interfaces of a subsystem Component
Interaction	– Owned Behavior of a system or subsystem Component, as behavioral specification of its decomposition
State Machine, Activity	– Owned Behavior of a Part's typing Class, as specification of (internal) object behavior.
Opaque Behavior	– Effect of a Transition in the State Machine based specification of (part of the) object behavior. – Behavior of Call Behavior Action in the Activity based specification of (part of the) object behavior.

Table 2: Usage of Classifiers within MeDUSA Design UML-Model

C.2 General Transformation Strategy

The transformation strategy, provided in the following, is based on the transformation of those Classifiers presented in Table 2. That is, all Classifiers within a *Design UML-Model* are transferred into respective ANSI-C types, based on the different usage scenarios they are employed in.

That is, all UML Classifiers are transferred into a respective ANSI-C type, which is usually a struct, and a set of related functions. The declarations and implementations are accordingly generated into a corresponding header and source file, which are created for each individual Classifier, as outlined by Figure 62. For technical reasons (the struct declaration may need to refer to recursively refer to itself), an additional forward declaration header file is also generated, as depicted by Figure 62.

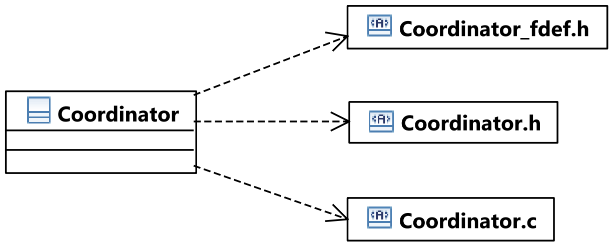


Fig. 62: General Transformation of Classifiers into Translation Units

The concrete file contents of course depends on the individual Classifier, begin transformed, and its respective usage scenario. It will be sketched in the following section. While such a transformation strategy would normally have to be formally defined by means of transformation rules between the UML and the ANSI-C language, here, for the sake of simplicity, the transformation will instead be described with the help of some meaningful examples (similar to as it is done with the specification of the MeDUSA UML model structure, provided in Appendix A).

C.3 Transformation of Classifiers

While all Classifiers are transformed into a respective ANSI-C type, Primitive types are handled somewhat outstanding. While both languages, UML as well as ANSI-C, support the concept of primitive types, the concrete realization differs in both languages. The UML only differentiates between the four different Primitive Types, namely Integer, Boolean, String, and Unlimited Natural, which are used within the definition of the UML meta model. The ANSI-C language standard on the other hand offers a broad set of build-in primitive types. As proposed by the UML specification, we thus advice to not use the UML Primitive Types within a MeDUSA *Design UML-Model*, and to instead use the modeling library of ANSI-C primitive types, depicted by Figure 63, during *Architectural* and *Detailed Design Modeling*.

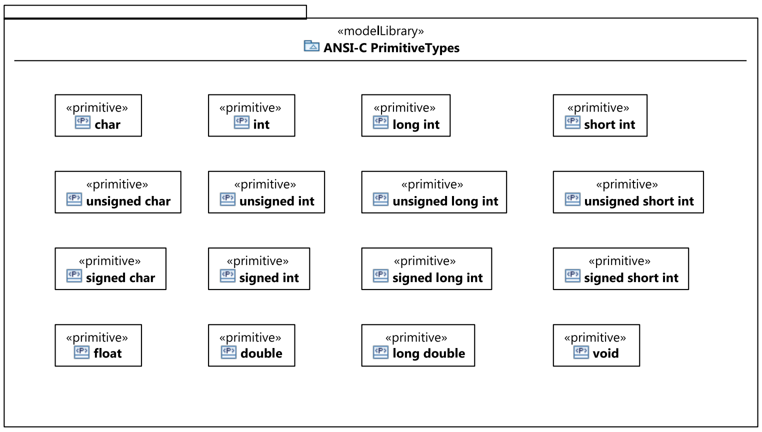


Fig. 63: ANSI-C Primitive Types UML-Library

As they are built-in into the ANSI-C language, those Primitive Types defined by the respective library do then not have to be transferred into respective ANSI-C types. Instead, their name just has to be used within the declaration of those ANSI-C variables and function parameters matching the UML Typed Elements of the input model.

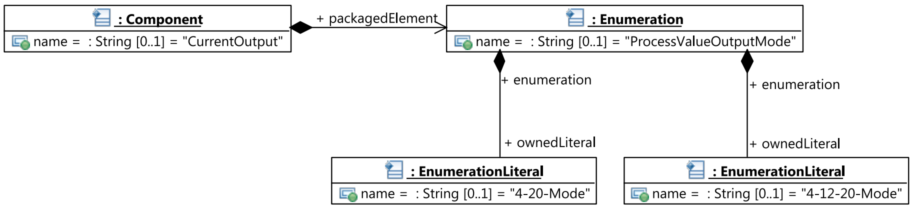
Besides Primitive Types, which are thus not covered by subsequently provided transformation schema for above outlined reasons, Activities and Interactions are further not covered.

The reason, Activities do not get transferred into ANSI-C code is that they are used to specify the behavior of Operations within a *Design UML-Model*, but usually not at an abstraction level that is sufficiently detailed to generate the complete corresponding implementation of a function (i.e. its body). Even while the UML would actually allow to model activities in such a fine grained manner, the effort is regarded to be inappropriately high, so within a MeDUSA *Design UML-Model* this is generally not employed. Generating only fragments of ANSI-C code into a function body is however rather impractical, as - to guarantee robustness of the generated code against re-generation runs - it would have to be anticipated, where a *Subsystem Implementer* has to manually add code, as those sections would have to be protected by guards from being overwritten during re-generation runs (compare [Fun06] for details)), causing the resulting code to be only hard to read and maintain.

The same holds for Interactions, which are developed within a MeDUSA *Design UML-Model* as specification of the collaborative behavior within the internal decomposition of the system or a subsystem. While it would for example be possible to generate ANSI-C code for *coordinator* objects from the external behavior that is depicted by those Interactions that were developed to specify the behavior within the internal decomposition of a subsystem, the resultant code would rather be incomplete. The order between messages, being modeled within different Interactions (there are usually several *Consolidated Behavioral Subsystem Design Diagrams*), can for example not be derived from the *Design UML-Model*.

Interactions are thus regarded to rather serve as some sort of *proof of concept* to depict that the collaborative behavior corresponds to the respective structural specifications. They are meant as a specification and documentation of the respective behavior and can thus serve as direct input to the manual implementation, which is performed by the respective *System* or *Subsystem Implementer*.

All other Classifiers are transformed as outlined above. They are covered in detail in the following, by means of examples. For the different Classifiers and scenarios being covered, an instance specification is first provided by means of a UML object diagram, depicting an example structure, as it would occur within the *Design UML-Model*. This is then followed by the source code, which would be generated accordingly for the Classifier within the respective scenario.

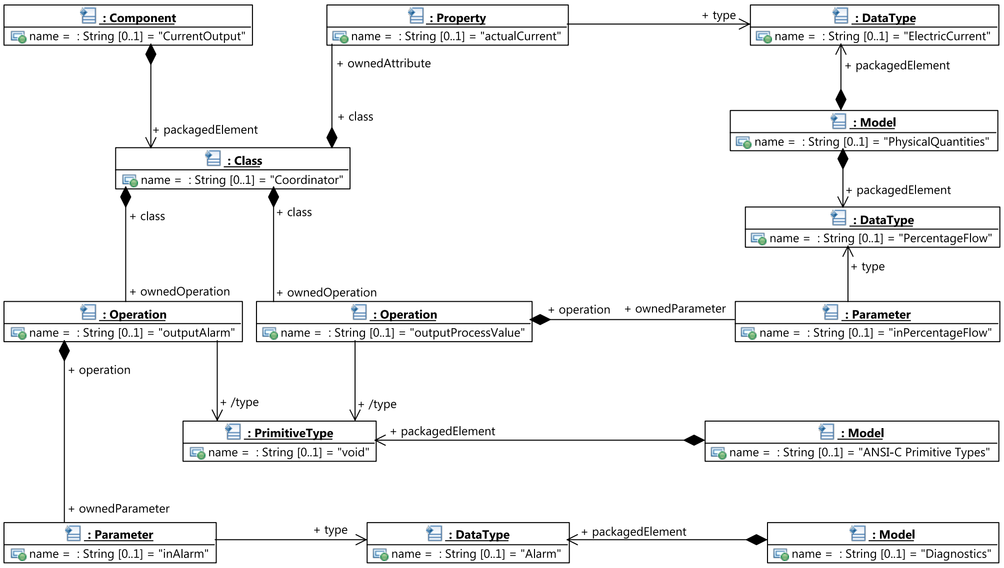


(a) Instance Specification

```
/*************************************************************
* File: ProcessValueOutputMode.h
*************************************************************/
...

/* Enumeration Declaration */
enum CurrentOutput_ProcessValueOutputMode {
 4-20-Mode,
 4-12-20-Mode
};
...
```

(b) Generated Code

Fig. 64: Transformation of Enumeration



(a) Example Instance Specification

Fig. 65: General Transformation of Data Type and (Kernel) Class

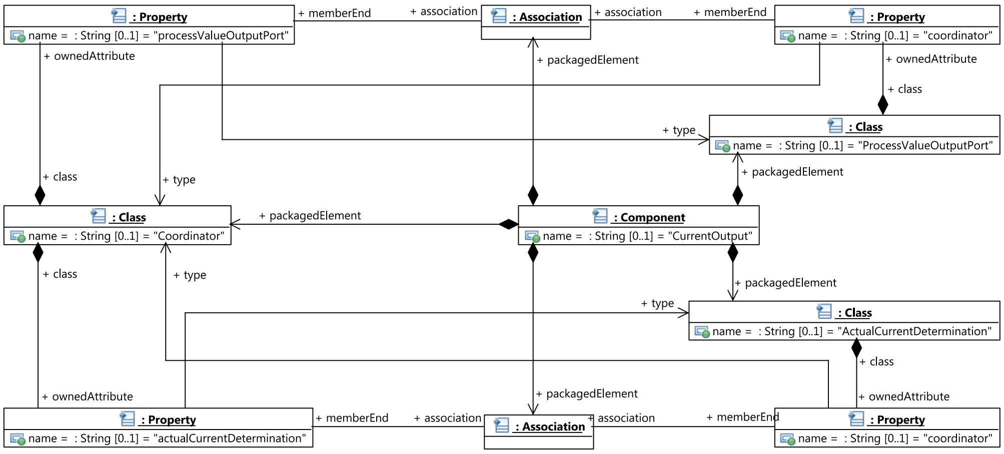
```c
/*************************************************************
* File: Coordinator_fdef.h
*************************************************************/
...

/* Classifier Struct Forward Declaration */
struct _CurrentOutput_Coordinator;

#define Example_System_CurrentOutput_Coordinator \
   struct _CurrentOutput_Coordinator
...
```

```c
/*************************************************************
* File: Coordinator.h
*************************************************************/
...

/* Classifier Struct Declaration */
struct _CurrentOutput_Coordinator {
 /* Owned Attributes */
 PhysicalQuantities_ElectricCurrent actualCurrent[1];
 ...
};

/* Constructors */
void CurrentOutput_Coordinator_create(CurrentOutput_Coordinator* self);

/* Destructors */
void CurrentOutput_Coordinator_destroy(CurrentOutput_Coordinator* self);
...

/* Owned Attributes Selectors */
PhysicalQuantities_ElectricCurrent*
 CurrentOutput_Coordinator_actualCurrrent(CurrentOutput_Coordinator* self);
...

/* Owned Operations */
ERROR_CODE CurrentOutput_Coordinator_outputProcessValue(
 CurrentOutput_Coordinator* self,
 PhysicalQuantities_PercentageFlow inPercentageFlow);

ERROR_CODE CurrentOutput_Coordinator_outputAlarm(
 CurrentOutput_Coordinator* self,
 Diagnostics_Alarm inAlarm);

...
```

```c
/*************************************************************
* File: Coordinator.c
*************************************************************/
...

void CurrentOutput_Coordinator_create(CurrentOutput_Coordinator* self) {
 ...
}

void CurrentOutput_Coordinator_destroy(CurrentOutput_Coordinator* self) {
 ...
}
...

/* Owned Attributes Selectors Implementation*/
PhysicalQuantities_ElectricCurrent*
 CurrentOutput_Coordinator_actualCurrent(CurrentOutput_Coordinator* self) {
   return *self->actualCurrent;
}
...

/* Owned Operations Implementation */
ERROR_CODE CurrentOutput_Coordinator_outputProcessValue(
 CurrentOutput_Coordinator* self,
 PhysicalQuantities_PercentageFlow inPercentageFlow) {
 ...
}

ERROR_CODE CurrentOutput_Coordinator_outputAlarm(
 CurrentOutput_Coordinator* self,
 Diagnostics_Alarm inAlarm) {
 ...
}

...
```

(b) Generated Code - Declaration of Struct and Constructors/Destructors

Fig. 65: General Transformation of Data Type and (Kernel) Class

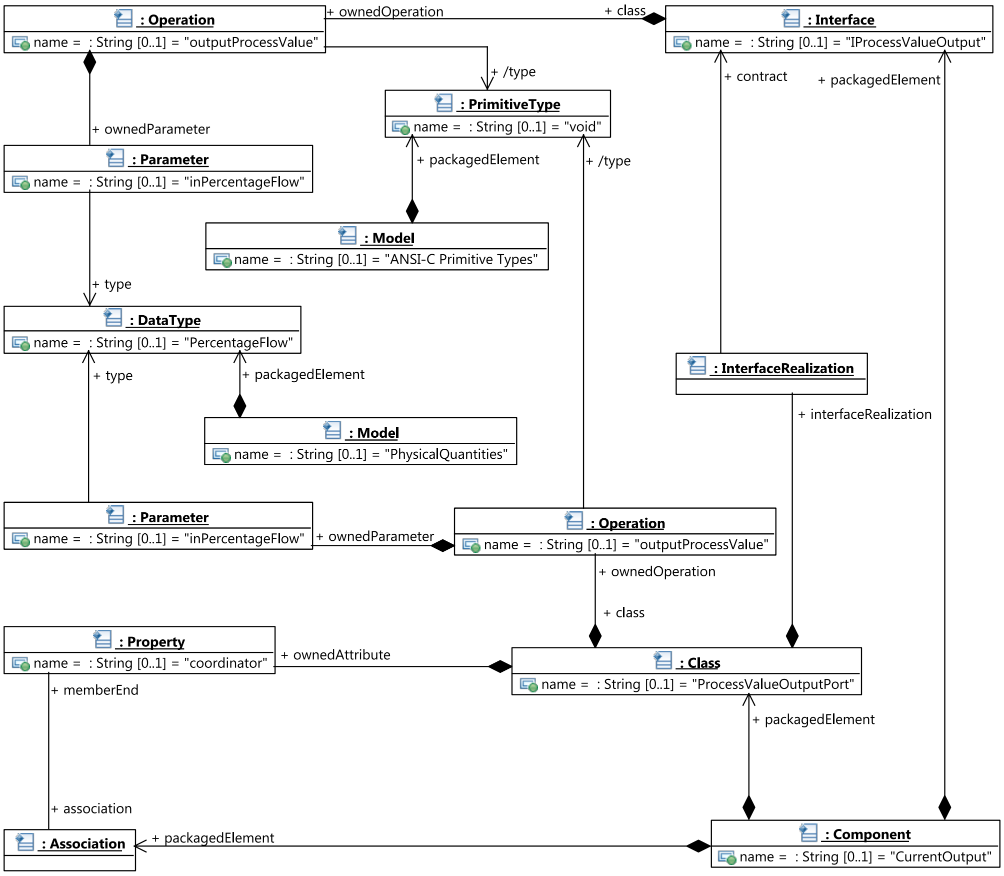


(a) Instance Specification: A Class typing a Part (Subsystem Decomposition)

```
/**************************************************************
* File: Coordinator.h
**************************************************************/
...

/* Classifier Struct Declaration */
struct _CurrentOutput_Coordinator {
 ...
 /* Association Ends */
 CurrentOutput_ProcessValueOutputPort processValueOutputPort[1];
 CurrentOutput_ActualCurrentDetermination actualCurrentDetermination[1];
 ...
};

...
```

(b) Generated Code: Member Declarations corresponding to Association Ends

Fig. 66: Additions for Transformation of (Kernel) Class typing Part - Transformation of Association Ends

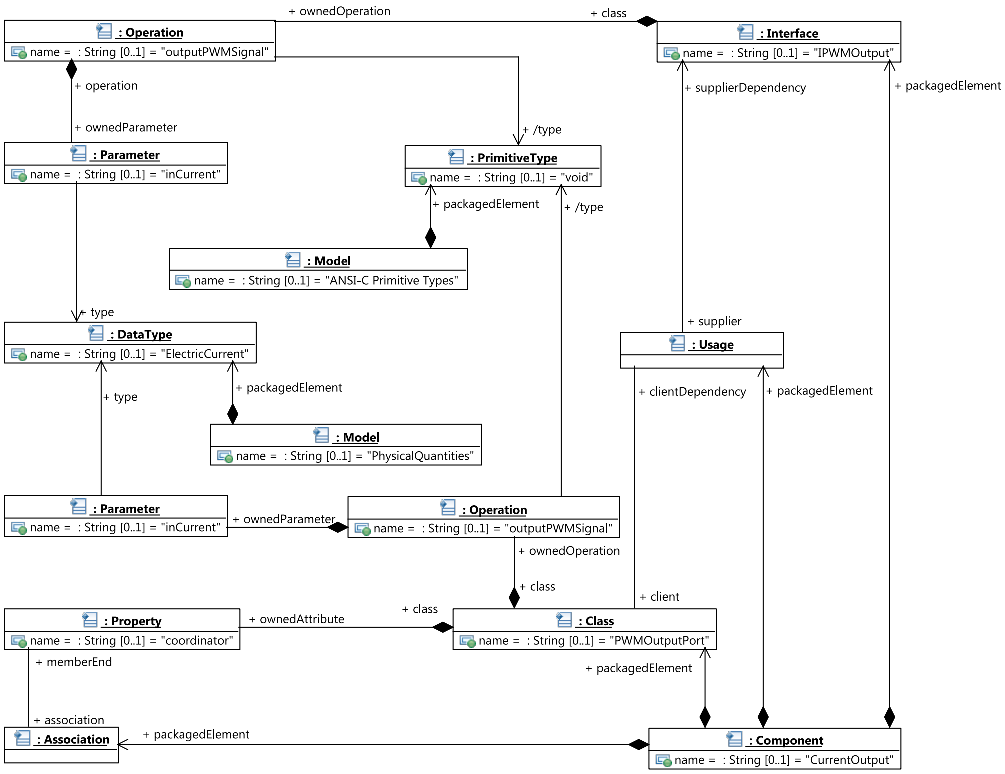


(a) Instance Specification: Class typing Port (Provided Interface)

```
/******************************************************************
* File: ProcessValueOutputPort.c
******************************************************************/
...

ERROR_CODE CurrentOutput_ProcessValueOutputPort_outputProcessValue(
  CurrentOutput_ProcessValueOutputPort* self,
  PhysicalQuantities_PercentageFlow* inPercentageFlow){
    return CurrentOuput_Coordinator_outputProcessValue(self->coordinator, inPercentageFlow);
}

...
```

(b) Generated Code: Function Declarations corresponding to Provided Interfaces

Fig. 67: Additions for Transformation of (Kernel) Class typing Port (Provided Interface)



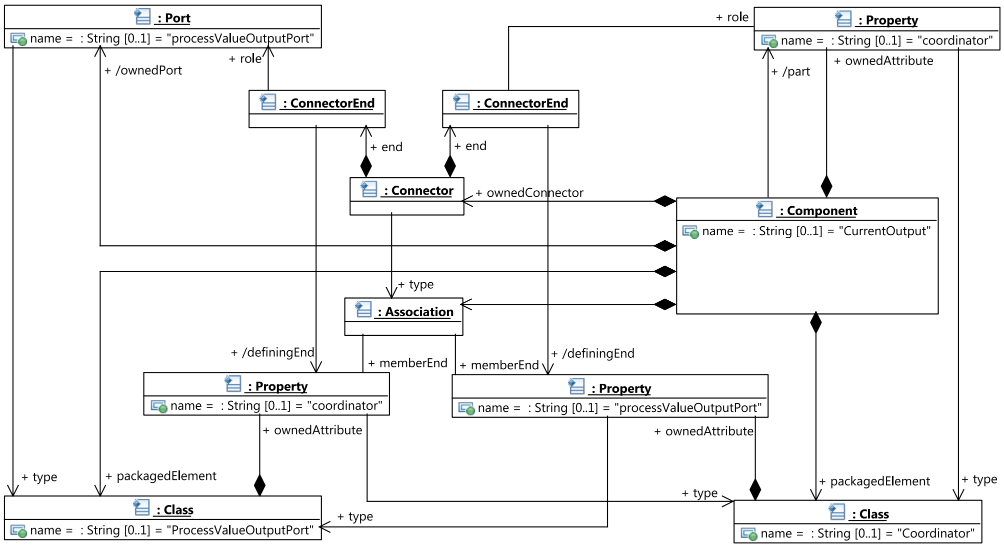
(a) Instance Specification: Instance Specification: Class typing Port (Required Interface)

```
/****************************************************************
* File: PWMOutputPort.h
***************************************************************/
...

/* Classifier Struct Declaration */
struct _CurrentOutput_PWMOutputPort {
  ...
  /* Required Interfaces */
  void* iPWMOutput;
  ERROR_CODE (* iPWMOutput_outputPWMSignal)(void* iPWMOutput, PhysicalQuantities_ElectricCurrent
        inCurrent);
  ...
};

...
```

```
/****************************************************************
* File: PWMOutputPort.c
***************************************************************/
...

ERROR_CODE CurrentOutput_PWMOutputPort_outputPWMSignal(
  CurrentOutput_PWMOutputPort* self,
  PhysicalQuantities_ElectricCurrent* inCurrent){
  return (self->iPWMOutput_outputPWMSignal)(self->iPWMOutput, inCurrent);
}

...
```

(b) Generated Code: Function Pointer and Struct Pointer Member Declarations corresponding to Required Interfaces

Fig. 68: Additions for Transformation of (Kernel) Class typing Port (Required Interface)



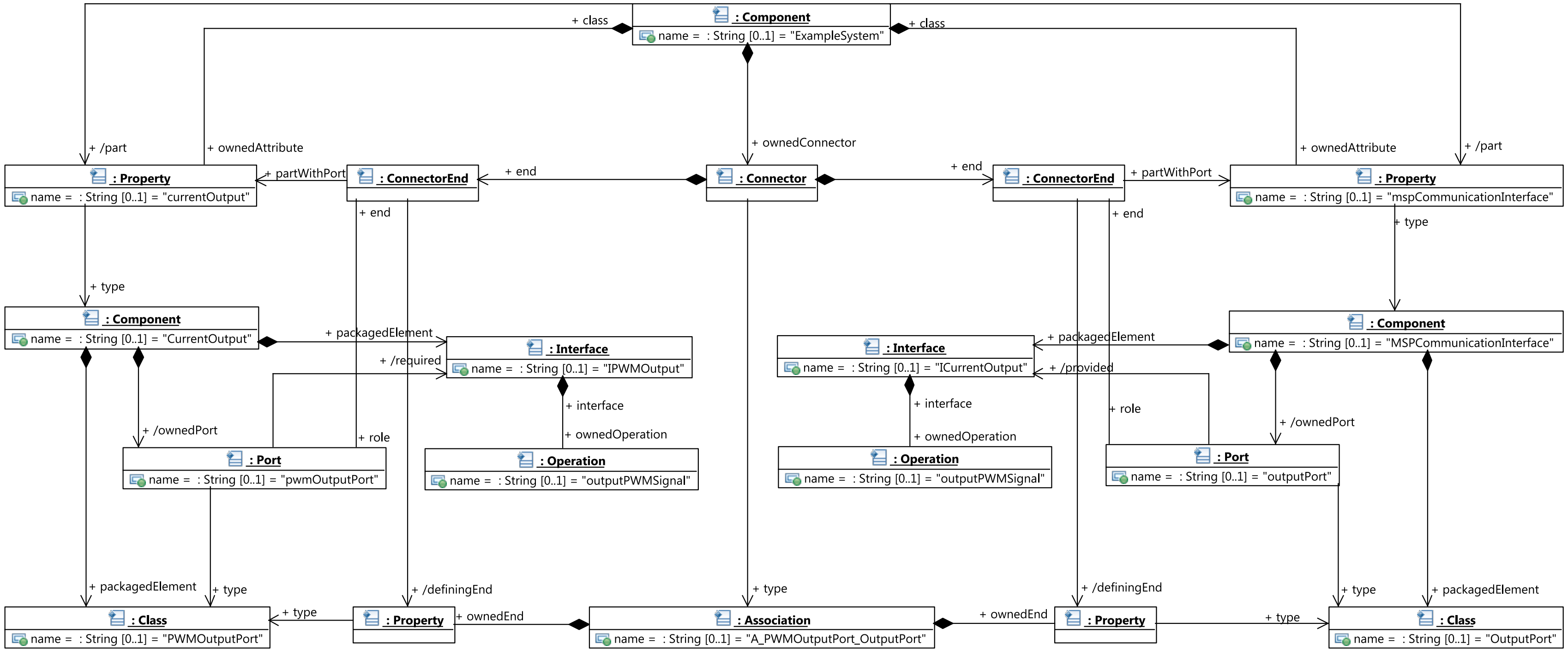
(a) Instance Specification: Subsystem Component

```
/****************************************************************
* File: CurrentOutput.h
****************************************************************/
...

/* Classifier Struct Declaration */
struct _CurrentOutput {
  /* Parts */
  CurrentOutput_Coordinator coordinator[1];
  ...
  /* Ports */
  CurrentOutput_ProcessValueOutputPort processValueOutputPort[1];
  ...
};

...
```

```
/****************************************************************
* File: CurrentOutput.c
****************************************************************/
...

void CurrentOutput_create(CurrentOutput_Coordinator* self) {
  /* Create Parts */
  CREATE_CurrentOutput_Coordinator(&self->coordinator[0]);
  ...
  /* Create Ports */
  CREATE_CurrentOutput_ProcessValueOutputPort(&self->processValueOutputPort[0]);
  ...
  /* Wiring */
  self->coordinator[0].processValueOutputPort[0] = &self->processValueOutputPort[0];
  self->processValueOutputPort[0].coordinator[0] = &self->coordinator[0];
  ...
}

...
```

(b) Generated Code: Struct Member Declarations for Parts and Ports, Wiring

Fig. 69: Transformation of subsystem Component



(a) Instance Specification: System Component

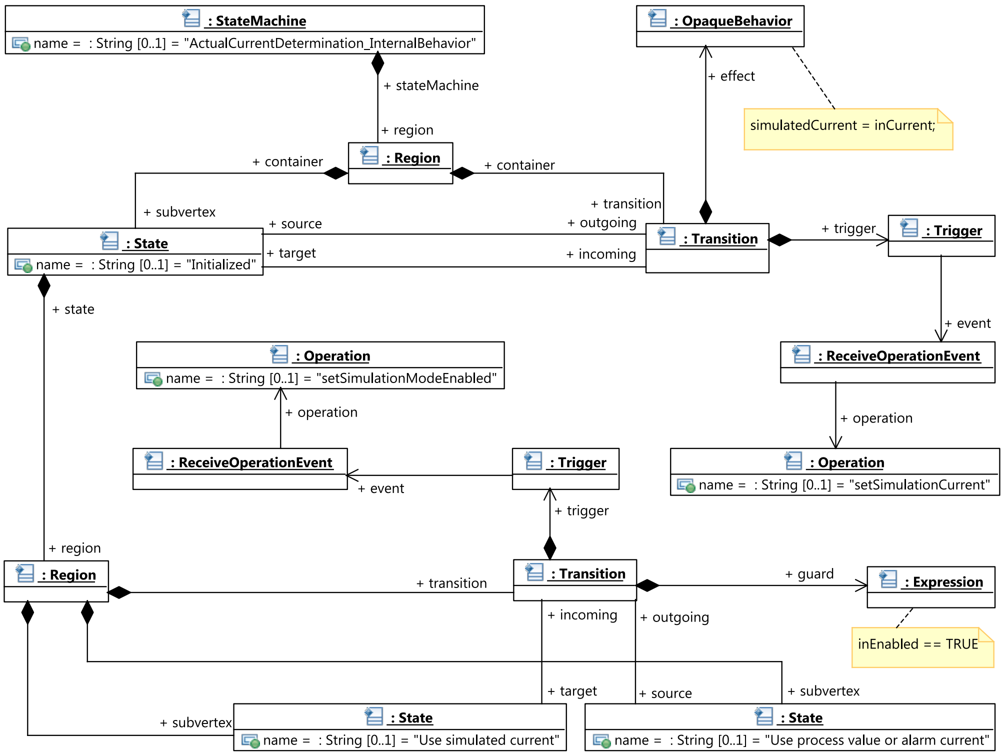
Fig. 70: Transformation of system Component

```c
/**************************************************************
* File: ExampleSystem.h
*************************************************************/
...

/* Classifier Struct Declaration */
struct _ExampleSystem {
 /* Parts (Subsystems) */
 CurrentOutput currentOutput[1];
 MSPCommunicationInterface mspCommunicationInterface[1];
 ...
};

...
```

```c
/**************************************************************
* File: ExampleSystem.c
*************************************************************/
...

void ExampleSystem_create(ExampleSystem* self) {
  /* Create Parts (Subsystems) */
  CREATE_CurrentOutput(&self->currentOutput[0]);
  CREATE_MSPCommunicationInterface(&self->mspCommunicationInterface[0]);
  ...

  /* Wiring (Required Interfaces) */
  self->currentOutput[0].pwmOutputPort[0].iPWMOutputPort =
    &self->mspCommunicationInterface[0].outputPort[0];
  self->currentOutput[0].pwmOutputPort[0].iPWMOutputPort_outputPWMSignal =
    &MSPCommunicationInterface_OutputPort_outputPWMSignal();
  ...
}

...
```

(b) Generated Code: Struct Member Declarations for Parts, Wiring of Parts within Constructor Implementation

Fig. 70: Transformation of system Component



(a) Instance Specification: State Machine

```
/*****************************************************************
* File: ActualCurrentDetermination_InternalBehavior.h
*****************************************************************/

/* (Flattened) States */
#define CurrentOutput_ActualCurrentDetermination_InternalBehavior_\
      STATE_Initialized_UseProcessValueOrAlarmCurrent 0x001
#define CurrentOutput_ActualCurrentDetermination_InternalBehavior_\
      STATE_Initialized_UseSimulatedCurrent 0x002
...

/* Events */
#define CurrentOutput_ActualCurrentDetermination_InternalBehavior_\
      RECEIVE_setSimulationModeEnabled    0x001
#define CurrentOutput_ActualCurrentDetermination_InternalBehavior_\
      RECEIVE_setSimulationCurrent        0x002
...

/* Classifier Struct Declaration */
struct _CurrentOutput_ActualCurrentDetermination_InternalBehavior{
  /* Current State */
  int state;
  ...
}

ERROR_CODE CurrentOutput_ActualCurrentDetermination_InternalBehavior_onEvent(
  CurrentOutput_ActualCurrentDetermination_InternalBehavior* self,
  int event,
  void* parameters[]);
```

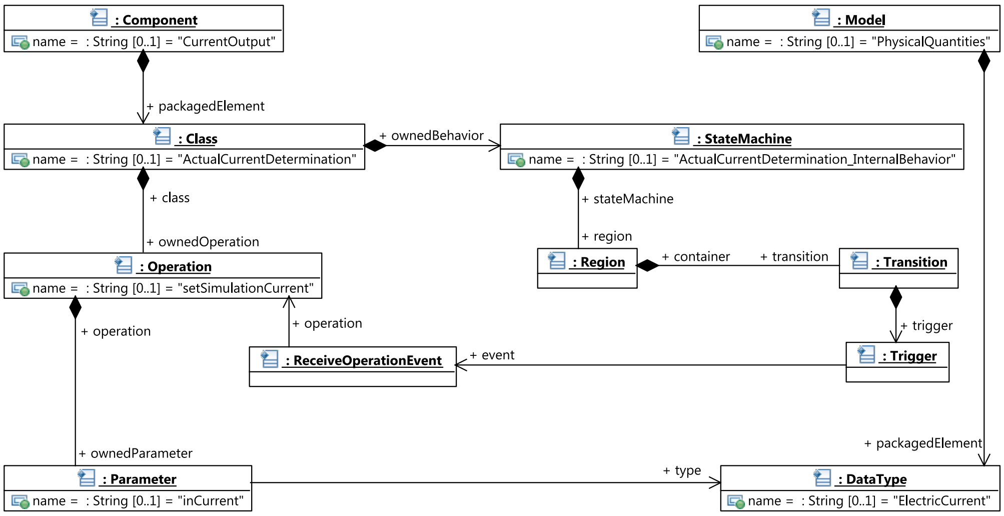
(b) Generated Code: Macro Definitions corresponding to States and Events, Corresponding Struct Declaration

Fig. 71: General Transformation of State Machine

```c
/***********************************************************
* File: ActualCurrentDetermination_InternalBehavior.c
***********************************************************/

/* Guards */
ERROR_CODE GUARD_1(void* parameters[]){
 // inEnabled == TRUE
 ...
}
...

/* Effects */
ERROR_CODE EFFECT_1(void* parameters[]){
 // simulated current = inCurrent;
 ...
}
...

/* Transitions */
ERROR_CODE TRANSITION_1(
 CurrentOutput_ActualCurrentDetermination_InternalBehavior* self, void* parameters[]){
 /* Check Guard */
 // no guard

 /* Execute Effect */
 EFFECT_1(parameters);

 /* Change State */
 //no state change

 return EVENT_CONSUMED;
}
...

ERROR_CODE TRANSITION_5(
 CurrentOutput_ActualCurrentDetermination_InternalBehavior* self, void* parameters[]){
 /* Check Guard */
 if(!GUARD_1(parameters))
   return GUARD_NOT_PASSED;

 /* Execute Effect */
 // no effect

 /* Change State */
 self->state =
 CurrentOutput_ActualCurrentDetermination_InternalBehavior_STATE_Initialized_UseSimulatedCurrent;

 return EVENT_CONSUMED;
}
...

/* Transition Table */
ERROR_CODE (* transitions)(
 CurrentOutput_ActualCurrentDetermination_InternalBehavior* self,
 void* parameters[]) [NUM_STATES][NUM_EVENTS] = {
   { TRANSITION_1, TRANSITION_2, TRANSITION_3, TRANSITION_4},
   { TRANSITION_5, TRANSITION_6, TRANSITION_7, TRANSITION_8}
}

void CurrentOutput_ActualCurrentDetermination_InternalBehavior_create(
 CurrentOutput_ActualCurrentDetermination_InternalBehavior* self){
   self->state =
   CurrentOutput_ActualCurrentDetermination_InternalBehavior_STATE_Initialized_UseProcessValueOrAlarmCurrent
        ;
}

ERROR_CODE CurrentOutput_ActualCurrentDetermination_InternalBehavior_onEvent(
 CurrentOutput_ActualCurrentDetermination_InternalBehavior* self,
 int event, void* parameters[]){
  if(transitions[self->actualState][event] != NULL){
    return transitions[self->actualState][event](self, parameters);
  }
  else{
    return NO_TRANSITION;
  }
}
...
```

(c) Generated Code: Function Implementations corresponding to Guards, Effects, and Transitions, Declaration of State-Transition Table as Function Pointer Array and Declaration of State Transition Function

Fig. 71: General Transformation of State Machine



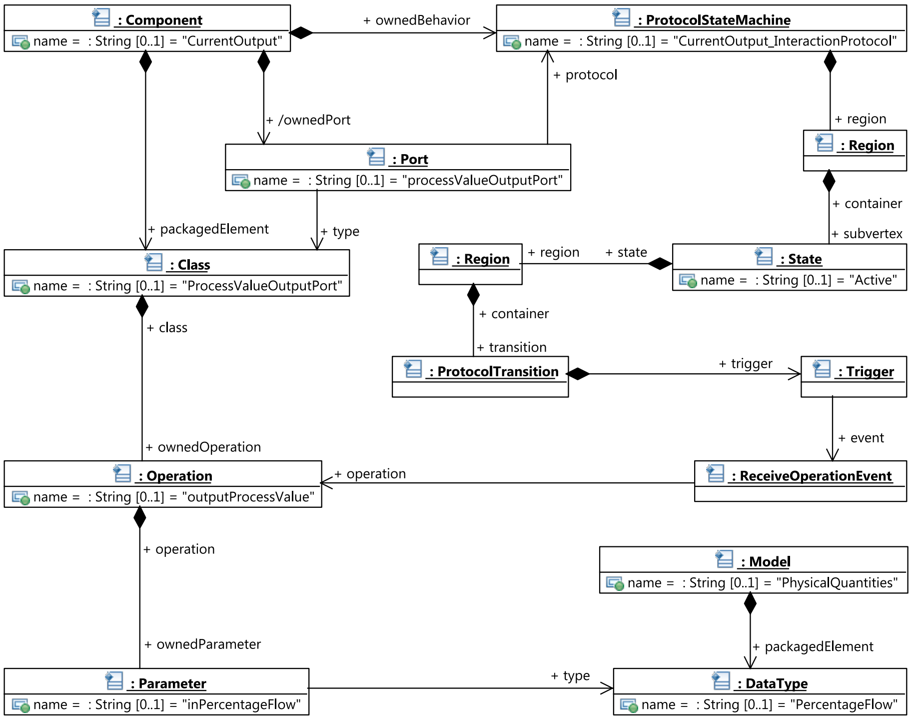
(a) Instance Specification: State Machine as Internal Behavior

```
/*****************************************************************
* File: ActualCurrentDetermination.h
*****************************************************************/
...

/* Classifier Struct Declaration */
struct _CurrentOutput_ActualCurrentDetermination {
  ...
  /* Owned Behavior */
  CurrentOutput_ActualCurrentDetermination_InternalBehavior ownedBehavior[1];
}
...
```

```
/*****************************************************************
* File: ActualCurrentDetermination.c
*****************************************************************/
...

void CurrentOutput_ActualCurrentDetermination_create(
  CurrentOutput_ActualCurrentDetermination* self){
  ...

  /* Initialize Owned Behavior */
  CurrentOutput_ActualCurrentDetermination_InternalBehavior_create(&self->ownedBehavior[0]);
}

ERROR_CODE CurrentOutput_ActualCurrentDetermination_setSimulationCurrent(
  CurrentOutput_ActualCurrentDetermination* self,
  PhysicalQuantities_ElectricCurrent* inCurrent){

  /* Call Owned Behavior */
  void* parameters[1];
  ... // allocate
  parameters[0] = inCurrent;

  CurrentOutput_ActualCurrentDetermination_InternalBehavior_onEvent(
    self->ownedBehavior[0],
    CurrentOutput_ActualCurrentDetermination_InternalBehavior_RECEIVE_setSimulationCurrent,
    parameters);
}

...
```

(b) Generated Code: Struct Member Declaration and Function Implementation within Owning Behaviored Classifier

Fig. 72: Additions for Transformation of a State Machine as Internal Behavior Specification



(a) Instance Specification: Protocol State Machine as Interaction Protocol

```
/****************************************************************
* File: ProcessValueOutputPort.h
****************************************************************/
...

/* Classifier Struct Declaration */
struct _CurrentOutput_ProcessValueOutputPort {
  ...
  CurrentOutput_InteractionProtocol protocol[1];
};

...
```

```
/****************************************************************
* File: ProcessValueOutputPort.c
****************************************************************/
...

ERROR_CODE CurrentOutput_ProcessValueOutputPort_outputProcessValue(
  CurrentOutput_ProcessValueOutputPort* self,
  PhysicalQuantities_PercentageFlow* inPercentageFlow){

  /* Check Protocol State Machine */
  void* parameters[1];
  ... // allocate
  parameters[0] = inPercentageFlow;

  int protocolConformance = CurrentOutput_InteractionProtocol_onEvent(
    self->protocol[0], CurrentOutput_InteractionProtocol_RECEIVE_outputProcessValue, parameters);

  /* Delegate Call */
  if(protocolConformance == EVENT_CONSUMED){
    return CurrentOuput_Coordinator_outputProcessValue(self->coordinator, inPercentageFlow);
  }
  else{
    return protocolConformance;
  }
}

...
```

(b) Generated Code: Struct Member Declaration and Function Implementations within Port's Type

Fig. 73: Additions for Transformation of a State Machine as Interaction Protocol Specification

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.

[Fun06] Mathias Funk. Generierung von effizientem C-Code aus UML2-Strukturdiagrammen, 2006. Diploma Thesis, RWTH Aachen University, http://www.swc.rwth-aachen.de/lufgi/teaching/theses/completed/Mathias_Funk_Thesis_Report.pdf.

[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.

[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. Addison Wesley, 1992.

[Kev07] Özgür Kevinc. Erweiterung des ViPER Codegenerators um Nebenläufigkeit und Zeitverhalten, 2007. Diploma Thesis, RWTH Aachen University, http://www.swc.rwth-aachen.de/lufgi/teaching/theses/completed/Oezguer_Kevinc_Thesis_Report.pdf.

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

[NL07a] Alexander Nyßen and Horst Lichter. MeDUSA - MethoD for Uml2-based Design of Embedded Software Applications. Technical Report AIB-2007-07, RWTH Aachen University, May 2007.

[NL07b] Alexander Nyßen and Horst Lichter. Use Case Modeling for Embedded Software Systems - Deficiencies & Workarounds. In M. Gehrke, H. Giese, and J. Stroop, editors, Proceedings of the 4th Workshop on Object-oriented Modeling of Embedded Real-Time Systems (OMER4), 30.-31. October, HNF Musuems Forum, Paderborn, Germany, pages 63–67, 2007.

[NLS+05] Alexander Nyßen, Horst Lichter, Jan Suchotzki, Peter Müller, and Andreas Stelter. UML2-basierte Architekturmodellierung kleiner eingebetteter Systeme - Erfahrungen einer Feldstudie. In Klein, Rumpe, and Schätz, editors, Tagungsband des Dagstuhl Workshops Modellbasierte Entwicklung eingebetteter Systeme (MBEES), 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.

[OMG07a] Software & Systems Process Engineering Metamodel Specification, v2.0 (Beta 2). OMG Document ptc/07-11-01, November 2007. http://www.omg.org/docs/ptc/07-11-01.pdf.

[OMG07b] UML Superstructure Specification, v2.1.2. OMG Formal Document 07-11-02, November 2007. http://www.omg.org/cgi-bin/doc?formal/07-11-02.

[Wal07] Andreas Walter. Ein Use Case-Modellierungswerkzeug für die ViPER-Plattform, 2007. Diploma Thesis, RWTH Aachen University, http://www.swc.rwth-aachen.de/lufgi/teaching/theses/completed/Andreas_Walter_Thesis_Report.pdf.

[Weg87] Peter Wegner. Dimensions of object-based language design. In OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications, pages 168–182, New York, NY, USA, 1987. ACM Press.

[WNHL08] Andreas Walter, Alexander Nyßen, Veit Hoffmann, and Horst Lichter. Werkzeugunterstützung für die use case-modellierung. In Korbinian Herrmann and Bernd Bruegger, editors, Proceedings of Software Engineering 2008, 18.-22.02.2008, Munich, Germany, page 58, 2008. Demo.

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`

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, and 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

2007-07 Alexander Nyßen, Horst Lichter: MeDUSA - MethoD for UML2-based Design of Embedded Software Applications

2007-08 Falk Salewski and Stefan Kowalewski: Achieving Highly Reliable Embedded Software: An empirical evaluation of different approaches

2007-09 Tina Kraußer, Heiko Mantel, and Henning Sudbrock: A Probabilistic Justification of the Combining Calculus under the Uniform Scheduler Assumption

2007-10 Martin Neuhäußer, Joost-Pieter Katoen: Bisimulation and Logical Preservation for Continuous-Time Markov Decision Processes

2007-11 Klaus Wehrle (editor): 6. Fachgespräch Sensornetzwerke

2007-12 Uwe Naumann: An L-Attributed Grammar for Adjoint Code

2007-13 Uwe Naumann, Michael Maier, Jan Riehme, and Bruce Christianson: Second-Order Adjoints by Source Code Manipulation of Numerical Programs

2007-14 Jean Utke, Uwe Naumann, Mike Fagan, Nathan Tallent, Michelle Strout, Patrick Heimbach, Chris Hill, and Carl Wunsch: OpenAD/F: A Modular, Open-Source Tool for Automatic Differentiation of Fortran Codes

2007-15 Volker Stolz: Temporal assertions for sequential and concurrent programs

2007-16 Sadeq Ali Makram, Mesut Güneç, Martin Wenig, Alexander Zimmermann: Adaptive Channel Assignment to Support QoS and Load Balancing for Wireless Mesh Networks

2007-17 René Thiemann: The DP Framework for Proving Termination of Term Rewriting

2007-18 Uwe Naumann: Call Tree Reversal is NP-Complete

2007-19 Jan Riehme, Andrea Walther, Jörg Stiller, Uwe Naumann: Adjoints for Time-Dependent Optimal Control

2007-20 Joost-Pieter Katoen, Daniel Klink, Martin Leucker, and Verena Wolf: Three-Valued Abstraction for Probabilistic Systems

2007-21 Tingting Han, Joost-Pieter Katoen, and Alexandru Mereacre: Compositional Modeling and Minimization of Time-Inhomogeneous Markov Chains

2007-22 Heiner Ackermann, Paul W. Goldberg, Vahab S. Mirrokni, Heiko Röglin, and Berthold Vöcking: Uncoordinated Two-Sided Markets

2008-01 * Fachgruppe Informatik: Jahresbericht 2007

2008-02 Henrik Bohnenkamp, Marielle Stoelinga: Quantitative Testing

2008-03 Carsten Fuhs, Jürgen Giesl, Aart Middeldorp, Peter Schneider-Kamp, René :Thiemann, Harald Zankl: Maximal Termination

2008-04 Uwe Naumann, Jan Riehme: Sensitivity Analysis in Sisyphe with the AD-Enabled NAGWare Fortran Compiler

2008-05 Frank G. Radmacher: An Automata Theoretic Approach to the Theory of Rational Tree Relations

2008-06 Uwe Naumann, Laurent Hascoet, Chris Hill, Paul Hovland, Jan Riehme, Jean Utke: A Framework for Proving Correctness of Adjoint Message Passing Programs

* These reports are only available as a printed version.
Please contact biblio@informatik.rwth-aachen.de to obtain copies.