Pattern and Component-based Development of Dependable Systems

Der Fakultät für Ingenieurwissenschaften,

Abteilung Informatik und Angewandte Kognitionswissenschaft
Universität Duisburg-Essen
zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften (Doktor-Ingenieur - Dr.-Ing.)

vorgelegte Dissertation

von Denis Hatebur

geboren am 27.09.1977 in Datteln

Datum der Einreichung: 08.05.2012

Contents

List of Figures

List of Tables

Abstract

In this thesis, we present an engineering process for developing software-based dependable systems based on *patterns* and re-usable *components*.

The process starts with expressing dependability requirements, such as confidentiality, integrity, availability, and reliability using patterns that refer to suitable descriptions of the environment. This thesis considers random faults as well as certain attacks and therefore supports a combined safety and security engineering. The patterns - attached to functional requirements - are part of a *pattern system* that can be used to identify missing requirements. The consolidated requirements can be used to develop the specification of the system to be built systematically.

For architectural design, knowledge from the requirements analysis can be used together with *architectural patterns* and *generic components*. We describe step-by-step how the architecture for a dependable system can be developed and analyzed.

For the system - implemented according to this architecture - the functional requirements and the dependability requirements should be verified. We propose a new method for system validation by means of testing, which is based on environment models expressed as UML state machines.

Each aspect is illustrated on small examples of a cooperative adaptive cruise control (CACC) system, and the whole approach is illustrated on a patient care system (PCS) as a comprehensive example.

German Abstract

In dieser Arbeit presentieren wir einen Entwicklungsprozess für verlässliche softwarebasierte Systeme, der auf Mustern und wiederverwendbaren Komponenten basiert.

Der Prozess beginnt mit dem Aufstellen von Verlässlichkeitsanforderungen, wie zum Beispiel für Vertraulichkeit, Integrität, und Verfügbarkeit und Zuverlässigkeit. Die Anforderungen werden mit Hilfe von Mustern beschrieben und beziehen sich auf geeignete Beschreibungen der Umgebung. In dieser Arbeit werden zufällige Fehler aber auch gezielte Angriffe betrachtet. Daher wird ein ingenieurmäßige Entwicklung von sicheren Systemen im Sinne von Safety (Schutz der Umgebung) und Security (Informationssicherheit) unterstützt. Die Muster - funktionalen Anforderungen zugeordnet - sind Teil eines *Mustersystems*, das zur Identifikation von fehlenden Anforderungen genutzt werden kann. Die so konsolidierten Anforderungen können genutzt werden, um die Spezifikation des Systems systematisch zu entwickeln.

Um die Architektur zu entwickeln, kann ebenfalls Wissen aus der Anforderungsanalyse, zusammen mit *Architekturmustern* und *generischen Komponenten* verwendet werden. In einer Schritt-für-Schritt Anleitung beschreiben wir, wie die Architektur für verlässliche Systeme entwickelt und analysiert werden kann.

An dem System - das entsprechend der Architektur implementiert wurde - müssen die funktionalen Anforderungen und die Verlässlichkeitsanforderungen verifiziert werden. Wir schlagen eine neue Testmethode zur Systemverifikation vor, die auf einem als UML Zustandsmachinenen dargestellten Umgebungsmodell basiert.

Jeder Aspekt wird an einem kleinen Beispiel eines kooperativen Tempomaten gezeigt und der gesamte Ansatz wird an einem Patienten-Behandlungs-System als vollständiges Beispiel dargestellt.

Acknowledgements

Special thanks to my wife Michaela and my daughters Jacqueline and Amelie for their extraordinary patience during my work on this thesis.

This work started in the “Software Engineering” working group of Prof. Maritta Heisel (with Azadeh Alebrahim, Kristian Beckers, Vanessa Bembenek, Isabelle Cóte, Stephan Faßbender, Holger Schmidt, and Ina Wentzlaff) at the University Duisburg-Essen. In this work group I got valuable feedback on my work, we had a lot of interesting discussions, and these discussions helped me to get the key ideas for this thesis. Working in this group was always a pleasure.

I am very grateful to Prof. Maritta Heisel, who supervised this thesis and worked with me on many of the topics. She reviewed most of the underlying work, found too many spelling mistakes, had good ideas for improvement, and provided constructive feedback.

I would like to express thanks to Holger Schmidt, who supported my work with joint research sessions on security aspects, and Isabelle Cóte who spent hours with me on developing the OCL integrity conditions and improving ADIT.

My sincere gratitude to Christine Choppy for the interesting work on architectures.

Thanks to the anonymous reviewers for their constructive comments on our papers and the book chapter, which we used as a basis for this thesis.

Chapter 1

Introduction

Dependable systems play an increasingly important role in daily life. More and more tasks are supported or performed by computer systems with software. These systems are often required to be safe, secure, available, reliable, and maintainable.

Safety is the *in*ability of the system to have an undesirable effect on its environment, and **security** is the *in*ability of the environment to have an undesirable effect on the system (Røstad, Tøndel, Line, & Nordland, 2006). To achieve safety, systematic and random faults must be handled. For security, certain attackers must be considered. **Availability** is the readiness for service (up-time vs. down-time) (Laprie, 1995). **Reliability** is a measure of continuous service accomplishment (Laprie, 1995). Maintainability is not considered in this thesis.

This thesis focuses on embedded systems development. Especially, for embedded systems, dependability must be considered, since many embedded systems are used in critical infrastructures. Embedded systems are computer-based systems being part of a product other than a computer (Simon, 2004). They consist of hardware and software components, and are used in the application domains automotive, aviation and space technology, medical technology, traffic guidance technology, industrial automation, telecommunications, business, entertainment, and household. According to Broy and Pree (2003), about 98 % of the CPUs produced worldwide are used in embedded systems. Since embedded systems are usually produced in large numbers, incorrectly functioning systems might cause large damages.

It is a widely accepted opinion in the software engineering community that reusing software development knowledge helps to avoid errors and to speed up the development of a software product. One promising attempt that enables software engineers to systematically construct software using a body of accumulated knowledge are *patterns*.

Patterns have been introduced on the level of detailed object oriented design (Gamma, Helm, Johnson, & Vlissides, 1995a). Today, patterns are defined for different software development activities. *Problem Frames* (Jackson, 2001) are patterns that classify software development *problems*. *Architectural styles* are "architectural patterns" that characterize software architectures (Buschmann, Meunier, Rohnert, Sommerlad, & Stal, 1996). *Design Patterns* are used for finer-grained software design, while *frameworks* (Fayad & Johnson, 1999) are considered as less abstract, more specialized. Finally, *idioms* are low-level patterns related to specific programming languages (Buschmann et al., 1996), and are sometimes called "code patterns".

Dependability requirements must be described and analyzed. Problem frames (Jackson, 2001) are a means to describe and analyze functional requirements, but they can be extended to describe also dependability mechanisms, as shown in earlier papers (Hatebur, Heisel, & Schmidt, 2006, 2007a). Functional requirements expressed with problem frames refer to and constrain parts of the environment. They describe the how the environment should behave when the system to be built is operating. Especially for dependability requirements the environment is of importance, i.e., a system may be safe and secure enough in one environment (in a private home), but not in another environment with e.g, more potential attackers, high electro-magnetic influence, or persons around who are not able to avoid harm that maybe caused by the system.

Another promising attempt to reuse software development knowledge are reusable components. The basic idea of *component-based software development* is to build software systems from smaller (already developed and tested) parts. The component approach tries to apply standard engineering methods to software development.

The developed system has to be implemented in such a way that the intended functionality and also the dependability requirements are fulfilled. This implementation has to be verified and validated. For the verification, we want to make use of the model of the environment in which the system will be operating. This model can be used to test the system to be built against its requirements, which refer to the environment.

Within this thesis, the following goals are pursued:

- We investigate how dependability can be systematically integrated into system development.
- We create an approach that enables a seamless development from the requirements engineering to implementation and testing. Seamless development has the potential to lead to a high quality of the system. High system quality is a necessary condition for dependable systems. Our basic approach is to use the same framework for requirements presentation and architectural design.
- We investigate how to describe dependability requirements in order to be able
  - to analyze them,
  - to use them to derive the specification of the system, and
  - to use them for the architectural design.
- We investigate how to systematically integrate mechanisms into the architecture that address the dependability requirements.
- We collect approaches to achieve a dependable implementation.
- We investigate how the description of the environment, necessary to describe dependability requirements, can be used for testing.
- We investigate to which extend our approaches support the certification of a developed product.

The thesis is structured as follows: In Chapter 2 basic concepts necessary to understand the thesis are presented. Chapter 3 introduces the ADIT software development process (Hatebur & Heisel, 2009a). This process is extended for dependability engineering for embedded systems. Chapter 4 describes a requirements engineering approach using patterns. In Chapter 5, we define a set of patterns that can be used to describe and analyze dependability requirements. Chapter 6 describes how to integrate the use of the dependability patterns into a system development process and analyze the requirements for possible interactions and completeness. In Chapter 7 we present a systematic approach for developing specifications for dependable systems. In Chapter 8 we present a method to systematically derive software architectures from problem descriptions. In Chapter 9 this method is extended to address dependability requirements systematically in the architecture. Chapter 10 summarizes some generic rules for implementing dependable systems. In Chapter 11 we propose a new method for system validation by means of testing, which is based on environment models expressed as UML state machines. Chapter 12 relates the presented work to commonly used standards like Common Criteria for security aspects as well as IEC/ISO 61508 for safety aspects. Additional to the partial case studies in each chapter,

Chapter 13 contains a case study including all steps of the development process. In each chapter, the related work is addressed. The thesis closes with a summary and perspectives in Chapter 14.

Chapter 2

Basic Concepts

In this chapter, we briefly introduce basic concepts, notations, and terminology used in this thesis. Section 2.1 introduces the key concepts and terminology in the field of dependability engineering. Section 2.2 introduces the agenda concept, Section 2.3 context diagrams, and Section 2.4 introduces terminology and problem frames developed by Jackson (2001). In Section 2.5 the UML notations used in this thesis are introduced. Section 2.6 introduces basics about UMLsec. Section 2.7 give a brief introduction to the Object Constraint Language. In Section 2.8 gives an overview on the terminology in the field of architectures and architectural styles.

2.1. Dependability Concepts and Terminology

An important quality attribute for systems is dependability. It comprises safety, security, reliability, and maintainability.

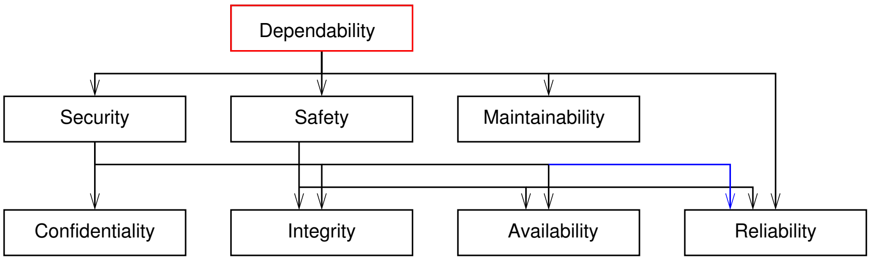


Figure 2.1.: *Dependability Terminology*

For this thesis, we define the terms in Fig. 2.1 as follows:

- **Security** is the inability of the *environment* to have an undesirable effect on the system, considering attackers (Røstad et al., 2006). For security certain attackers must be considered.

- **Safety** is the inability of the *system* to have an undesirable effect on its environment, considering systematic and random faults (Røstad et al., 2006).

- **Maintainability** is the ability to undergo modifications and repairs. (Avizienis, Laprie, Randall, & Landwehr, 2004). Maintainability can be achieved by additional interfaces for updates (of the whole software or components), by a maintainable structure of the software itself (e.g., documentation, appropriate architectures, comments in the source code), and by maintenance plans (e.g., restart the software once a week to reduce memory fragmentation). Maintainability is not considered in this thesis.

Security can be described by confidentiality, integrity and availability requirements. **Confidentiality** is the absence of unauthorized disclosure of information. **Integrity** is the absence of improper system, data, or a service alterations (Pfitzmann & Hansen, 2006). **Availability** is the readiness for service (up-time vs. down-time) (Laprie, 1995). Availability, in contrast to reliability, does not require correct service. Also for safety, integrity and availability must be considered. For safety, integrity and availability mechanisms have to protect against random (and some systematic) faults. **Reliability** is a measure of continuous service accomplishment (Laprie, 1995). A safety-critical system has to perform its safety functions with a defined reliability (or integrity). In this case, reliability describes the probability of correct functionality under stipulated environmental conditions (Courtois, 1997). This terminology is used in the whole thesis and especially in Chapter 5.

2.2. Agenda Concept

The *agenda* concept (Heisel, 1998) is used in this thesis to describe processes. An agenda is a list of steps or phases to be performed when carrying out some tasks in the context of systems and software engineering. Each step results in a document that is expressed in a certain language, e.g., natural language, problem diagrams (cf. Section 2.4), UML (Unified Modeling Language (UML Revision Task Force, 2010c)) diagrams, or even formal languages can be used. Agendas contain informal descriptions of the steps, which may depend on each other. Therefore they are a method to guide systems and software development activities. Additionally, agendas support quality assurance, because the steps may have validation conditions associated with them that help to detect errors as early as possible in the process. These validation conditions state necessary semantic conditions that the developed artifact must fulfill in order to serve its purpose properly. The agenda concept is used to describe our development process (see Appendix A) and parts of our dependability extension described in Chapter 3.

2.3. Context Diagrams

The environment in which the machine (system to be built) will operate can be represented by a context diagram (Jackson, 2001). It is used for structuring of problems by structuring the description of the environment.

A context diagram consists of domains and interfaces. Domains are represented by rectangles. Plain rectangles denote *application domains* (that already exist). A rectangle with a single vertical stripe denotes a *designed domain* physically representing some information. A rectangle with a double vertical stripe denotes the machine to be developed. The connecting lines represent interfaces that consist of *shared phenomena*. A shared phenomenon of an interface is controlled by one domain and it can be observed by other domains. However, a context diagram does not show who is in control of the shared phenomena.

An example of a context diagram is shown in Fig. 2.2. This context diagram shows a patient monitoring system. The Monitor machine is the machine domain in this context. The domain Periods & Ranges is a designed domain. All other domains (e.g., Medical staff, Nurses' station) are application domains. The interfaces in this context diagram are denoted with a, b, c, d, e, and f. Period, Range, PatientName, and Factor are shared phenomena associated with the interface a.

The domain Analog devices is a *connection domain*. Connection domains connect two ore more other domains. They represent a communication medium or device between these domains. Connection domains have to be considered if connections are unreliable, introduce delays that are an essential part of the problem, convert phenomena, or are explicitly mentioned in the

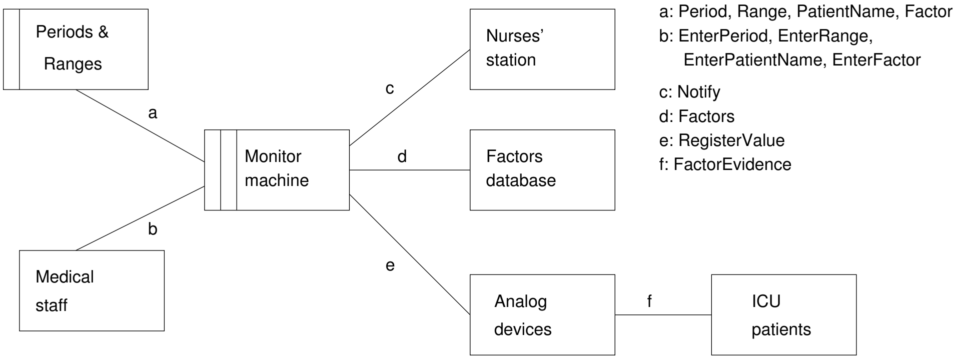


Figure 2.2.: *Context Diagram: Patient Monitoring System (cf. (Jackson, 2001))*

requirements (Jackson, 2001). Other examples for connection domains are network connections, display units, or keyboards that are used for user input.

In Chapter 4, this context diagram notation is extended and context diagrams are used in Chapters 7, 8, and 9.

2.4. Problem Frames

Problem frames are a means to describe software development problems. They were proposed by Michael Jackson (Jackson, 2001), who describes them as follows:

> "A problem frame is a kind of pattern. It defines an intuitively identifiable problem class in terms of its context and the characteristics of its domains, interfaces and requirement."

Problem frames classify *simple problems*. A real-world software development problem is usually a complex problem. Hence, we must decompose a complex problem into subproblems such that problem frames are applicable. Problem frames are described by *frame diagrams*, which basically consist of rectangles, a dashed oval, and different links between them, see Fig. 2.3. The task is to construct a *machine* that establishes the desired behavior of the environment (in which it is integrated) in accordance with the requirements.

Jackson (Jackson, 2001, p. 83f) considers three main domain types:

- *"A **biddable domain** usually consists of people. The most important characteristic of a biddable domain is that it is physical but lacks positive predictable internal causality. That is, in most situations it is impossible to compel a person to initiate an event: the most that can be done is to issue instructions to be followed."*

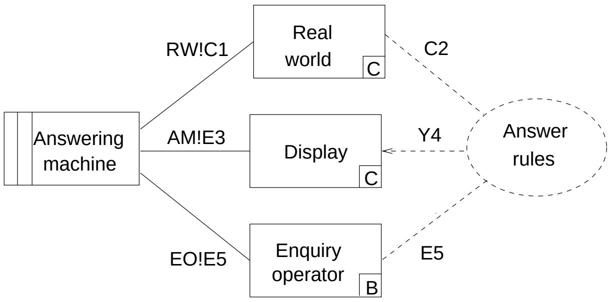


Figure 2.3.: *Commanded Information problem frame (cf. (Jackson, 2001))*

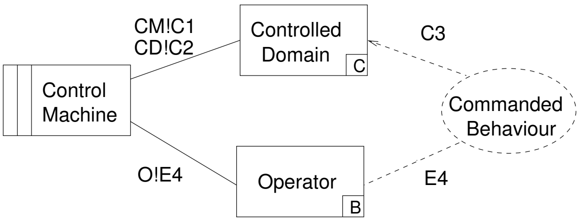


Figure 2.4.: *Commanded Behaviour problem frame (cf. (Jackson, 2001))*

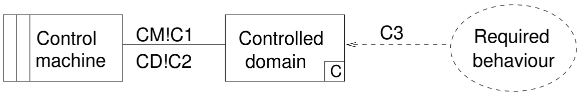


Figure 2.5.: *Required Behaviour problem frame (cf. (Jackson, 2001))*

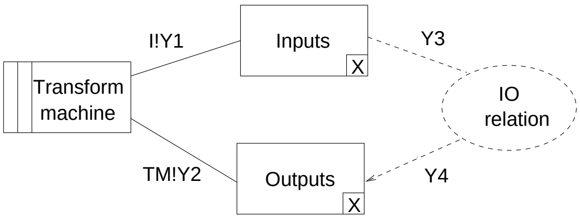


Figure 2.6.: *Transformation problem frame (cf. (Jackson, 2001))*

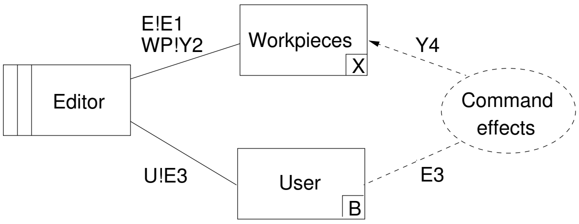


Figure 2.7.: *Simple Workpieces problem frame (cf. (Jackson, 2001))*

Biddable domains are indicated by B (e.g., Enquiry operator in Fig. 2.3).

- "*A **causal domain** is one whose properties include predictable causal relationships among its causal phenomena.*"

  Often, causal domains are mechanical or electrical equipment. They are indicated with a C in frame diagrams (e.g., Display in Fig. 2.3). Their actions and reactions are predictable. Thus, they can be controlled by other domains.

- "*A **lexical domain** is a physical representation of data – that is, of symbolic phenomena. It combines causal and symbolic phenomena in a special way. The causal properties allow the data to be written and read.*"

  Lexical domains are indicated by X.

For domains and interfaces, the same notations as for the context diagram can be used. *Requirements* are denoted with a dashed oval. A dashed line represents a requirement reference, and an arrow indicates that the requirement *constrains* a domain. If a domain is constrained by the requirement, we must develop a machine, which controls this domain accordingly. In Fig. 2.3, the Display domain is constrained, because the Answering machine changes it on behalf of Enquiry operator commands to satisfy the required Answer rules. Shared phenomena are observable by at least two domains, but controlled by only one domain. In the problem diagram and frame, this is indicated by an exclamation mark. For example, in Fig. 2.3 the notation EO!E5 means that the phenomena in the set E5 are *controlled* by the domain Enquiry operator and *observed* by the domain Answering machine.

Jackson identified some basic problem frames: Fig. 2.4 shows the *Commanded Behaviour* frame. That frame addresses the issue of controlling the behavior of the controlled domain according to the commands of the operator. The *Required Behaviour* (Fig. 2.5) frame is similar but without an operator; the control of the behavior has to be achieved in accordance with some rules. Other basic problem frames are the *Transformation* frame in Fig. 2.6 that addresses the production of required outputs from some inputs, and the *Simple Workpieces* frame in Fig. 2.7 that corresponds to tools for creating and editing of computer processable text, graphic objects etc. Additional frames can be developed: For example, the *Commanded Information* frame

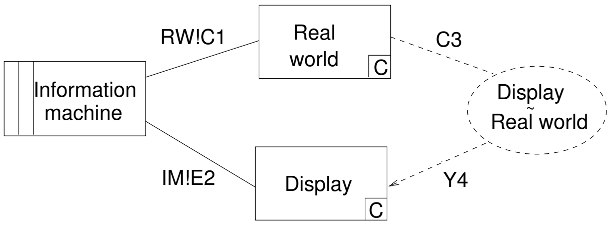


Figure 2.8.: *Information Display problem frame (cf. (Jackson, 2001))*

shown in Fig. 2.3 on Page 7 is a variant of the *Information Display* frame (see Fig. 2.8) where there is no operator, and information about the states and behavior of some parts of the physical world is continuously needed.

It is possible and recommended to extend the list of problem frames. In (Côté, Hatebur, Heisel, Schmidt, & Wentzlaff, 2008), we give an enumeration of possible problem frames, based on domain characteristics, and comment on the usefulness of the obtained frames. In particular, we investigate problem domains and their characteristics in detail. This leads to fine-grained criteria for describing problem domains. As a result, we identify a new type of problem domain (display domain, see Section 4.1.3) and come up with integrity conditions for developing useful problem frames (see Section 4.1.4). Taking a complete enumeration of possible problem frames (with at most three problem domains, of which only one is constrained) as a basis, we find 8 new problem frames. The problem frame *data-based control* is used in this thesis. This frame refers to a lexical domain and constrains a causal domain. It can be regarded as a *required behaviour* frame, because lexical domains are special causal ones. However, a separate frame is appropriate since different aspects (e.g., design of the data) have to be considered to solve such a problem. An example is a calendar-based heating control system (no heating on holidays).

Requirements engineering with problem frames proceeds as follows: first, the environment in which the machine will operate is represented by a *context diagram*. Then, the problem is decomposed into subproblems. Whenever possible, the decomposition is done in such a way that the subproblems fit to given problem frames.

A context diagram forms the basis for problem decomposition through *projection*. A complex problem can be decomposed into subproblems by applying *decomposition operators* to the according context diagram such that the resulting simple subproblems can be fitted to problem frames.

To fit a subproblem to a problem frame, one must *instantiate* its frame diagram, i.e., provide instances for its domains, phenomena, and interfaces. The instantiated frame diagram is called a *problem diagram*. A problem diagram for the odometer according the following description is given in Fig. 2.9 on the next page:

> A microchip computer is required to control a digital electronic speedometer and odometer in a car. One of the car's rear wheels generates pulses as it rotates. The computer can detect these pulses and must use them to set the current speed and total number of miles traveled in the two visible counters on the car fascia. The underlying registers of the counters are shared by the computer and the visible display.

The odometer problem diagram is an instance of the commanded information problem frame in Fig. 2.8. The car on road represents the real world and the Fascia display is an instance of the Display. Furthermore, relevant *domain knowledge* about the domains contained in the frame diagram must be elicited, examined, and documented. Domain knowledge consists of *facts* and *assumptions*. Facts describe fixed properties of the environment irrespective of how the machine is built, e.g., that a network connection is physically secured. Assumptions describe conditions

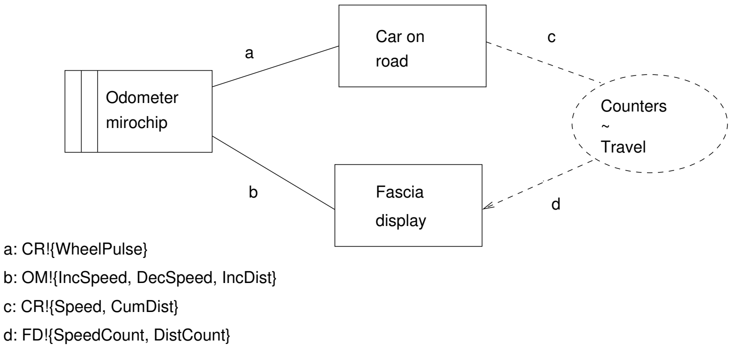


Figure 2.9.: *Odometer problem diagram (cf. (Jackson, 2001))*

that are needed, so that the requirements are accomplishable, e.g., we assume that a password selected by a user is not revealed by this user to other users.

Successfully fitting a problem to a given problem frame means that the concrete problem indeed exhibits the properties that are characteristic for the problem class defined by the problem frame. A problem can only be fitted to a problem frame if the involved problem domains belong to the domain types specified in the frame diagram. For example, the "User" domain of Fig. 2.7 can only be instantiated by persons, but not for example by some physical equipment like an elevator. Since the requirements refer to the environment in which the machine must operate, the next step consists in deriving a *specification* for the machine. A specification describes the behavior of the machine at its external interfaces (see (Jackson & Zave, 1995) for details). The specification describes the machine and is the starting point for its construction.

The problem frames are used for the functional requirements in Chapter 4 an is the basis for the handling of dependability requirements described in Chapter 5.

2.5. UML Diagrams

To specify different aspects of a system, we use the following UML diagram types:

- class diagrams,
- sequence diagrams, and
- composite structure diagrams

Syntax and semantic of these diagrams are given in UML Revision Task Force (2010c). In Chapter 4, class diagrams are used to express context diagrams, problem diagrams, problem frames, domain knowledge diagrams and mapping diagrams. In Chapter 5, class diagrams are used to express problem diagrams. We use sequence diagrams in Chapter 7 to express specifications. Architectures in Chapters 8 and 9 are expressed using composite structure diagrams. Nearly all UML elements of the class diagrams, sequence diagrams, and composite structure diagrams can be used within the associated steps of the development process.

2.6. UMLsec

UMLsec (Jürjens, 2005) constitutes a UML profile used to develop and analyze security models. UMLsec offers new UML language elements and constraints to specify typical security requirements such as secrecy (in this context, the same as confidentiality), integrity, authenticity (truthfulness of origins), and attacker models. The language elements are introduced using a profile with stereotypes Examples for pre-defined UMLsec stereotypes are ≪critical≫ to label security-critical parts of UML diagrams, ≪secure dependency≫ to ensure that dependent parts of models preserve the security requirements relevant for the parts they depend on, ≪secure links≫ to introduce attacker models, and ≪data security≫ to analyze behavior models with respect to confidentiality and integrity requirements. The aforementioned stereotypes are used in Chapter 7 for creating UMLsec specifications based on results from security requirements engineering. A detailed explanation and a formal foundation of the tags and stereotypes defined in UMLsec can be found in Jürjens (2005).

Based on UMLsec models and the semantics defined for the different UMLsec language elements, with UMLsec possible security vulnerabilities can be identified at a very early stage of software development. One can thus verify that the desired security requirements, if fulfilled, enforce a given security policy. This verification is supported by the UMLsec tool suite, which is available online via `http://www.umlsec.de/`. UMLsec is used to express specifications of secure systems in Chapter 7.

2.7. OCL – Object Constraint Language

The *Object Constraint Language* (OCL) (UML Revision Task Force, 2010a; Warmer & Kleppe, 2003) is part of UML (UML Revision Task Force, 2010c). It is a notation to describe constraints on object-oriented modeling artifacts such as class diagrams and sequence diagrams. A constraint is a restriction on one or more elements of an object-oriented model.

We use association ends or role names to navigate from the context to other model elements. Often associations are one-to-many or many-to-many, which means that constraints on a collection of objects are necessary. OCL expressions either state a fact about all objects in the collection using quantification or facts about the collection itself.

We make use of the following operators and keywords to construct constraints:

$boolean\ expression_1$ **implies** $boolean\ expression_2$ is a boolean expression used for logical implications.

$boolean\ expression_1$ **and** $boolean\ expression_2$ is a boolean expression representing the logical conjunction operator.

$boolean\ expression_1$ **or** $boolean\ expression_2$ is a boolean expression representing the logical disjunction operator.

$collection- > forAll(identifier\ :\ type\ |\ boolean\ expression)$ is the universal quantification operator; the result is true if *boolean expression* is true for all elements of *collection*.

$collection- > exists(identifier\ :\ type\ |\ boolean\ expression)$ is the existential quantification operator; the result is true if *boolean expression* is true for at least one element of *collection*.

$collection- > select(identifier\ :\ type\ |\ boolean\ expression)$ returns all elements from *collection* for which *boolean expression* is true.

$collection- > reject(identifier\ :\ type\ |\ boolean\ expression)$ returns all elements from *collection* for which *boolean expression* is false.

$collection- > collect(identifier\ :\ type\ |\ expression\ with\ identifier)$ returns a new collection derived from *collection* by evaluating *expression with identifier*.

$collection_1- > union(collection_2)$ returns the union of $collection_1$ and $collection_2$.

$collection- > size()$ returns the number of elements contained in *collection*

$collection->includes(identifier : type)$ returns true, if *identifier* is contained in *collection*, and false otherwise.

$collection->includesAll(identifier : collection(type))$ returns true, if *identifier* is a subset of *collection*, and false otherwise.

$identifier.oclIsTypeOf(type)$ returns true, if *identifier* has the type *type*, and false otherwise.

$identifier.oclAsType(type)$ returns the *identifier* as *type*, or if the conversion is not possible it returns *oclInvalid*.

$identifier_1 = identifier_2$ equality operator: returns true if $identifier_1$ is equal to $identifier_2$, and false otherwise.

$identifier_1 <> identifier_2$ inequality operator: returns true if $identifier_1$ is not equal to $identifier_2$, and false otherwise.

OCL is used to express integrity conditions on models in Chapters 4, 5 and 8. It is also used to describe pre- and postconditions of operations defining derived attributes in Chapter 5 and model transformation operations in Chapter 7.

2.8. Architectures and Architectural Styles

According to Bass, Clements, and Kazman (Bass, Clements, & Kazman, 1998),

> “the software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.”

Architectures can be given for hardware and software. To express hardware architectures, it is sufficient to describe components with interfaces between the components. This component-based structuring can be provided by hardware description languages like VHDL (Ashenden, 2002) or Verilog (Hoppe, 2006) and also by UML composite structure diagrams. In contrast, the architecture of software is multi-faceted (Kruchten, 1995): there exists a structural view, a process-oriented view, a function-oriented view, an object-oriented view with classes and relations, and a data flow view on a given software architecture. In this thesis a structural view with interfaces is used for the following reasons:

- The process-oriented view can be added by defining components as active or passive.
- The function-oriented view can be extracted from the interfaces and their descriptions.
- The possible data flow is given by the parameters and return values of methods in the interface classes.

Architectural styles are patterns for software architectures. A style is characterized by Bass, Clements, and Kazman (Bass et al., 1998) as

(i) a set of component types (e.g., data repository, process, procedure) that perform some function at runtime,

(ii) a topological layout of these components indicating their runtime interrelationships,

(iii) a set of semantic constraints (for example, a data repository is not allowed to change the values stored in it), and

(iv) a set of connectors (e.g., subroutine call, remote procedure call, data streams, sockets) that mediate communication, coordination, or cooperation among components.

When choosing an architecture for a machine, usually several architectural styles are possible, which means that all of them could be used to implement the functional requirements. Which architectural style is the most appropriate must then be decided using *non-functional* criteria such as efficiency, scalability, or modifiability.

Some important architectural styles are *layered architecture*, *repository architecture*, and the *pipe and filter architecture*.

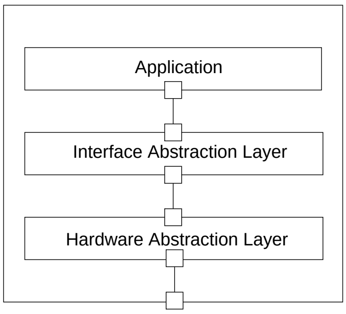


Figure 2.10.: *Layered Architecture*

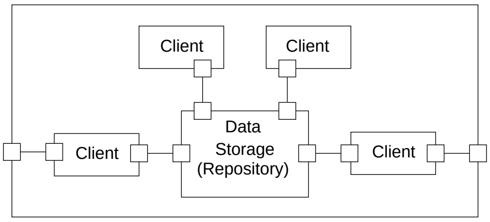


Figure 2.11.: *Repository Architecture*

Layered architecture A layered architecture allows a hierarchical organization of software. "Lower" layers provide services for "higher" layers. A well-known example is the ISO/OSI reference model for communication protocols (Zimmermann, 1980). A layered architectural pattern with only 3 layers is shown in Fig. 2.10. It consists of an *application layer* that processes the signals corresponding to those in the physical environment, an *interface Abstraction Layer* (IAL) with adapter components that transform the signals of the application into signals that can be understood by the *Hardware Abstraction Layer* (HAL). This layer provides abstract interfaces to the hardware components. The layered architecture divides the software into device-dependent and device-independent parts according to the (extended) *four variable model* developed by David Parnas and extended by Bharadwaj and Heitmeyer (1999). The four variables in the model are the monitored variables, the controlled variables, the input data, and the output data. The *monitored variables* are measured quantities (i.e., physical values, monitored by sensors). The *controlled variables* are affected quantities (i.e., physical values, controlled by actuators). The *input data* are resources from which the values of monitored variables must be determined. These are submitted via a technical interface (electrical signals corresponding to digital values). The *output data* are resources available to affect controlled variables. They are submitted via a technical interface by the machine (electrical signals corresponding to digital values). The basic idea of the extended four variable model is, that the application layer software should have the same interfaces as the system, i.e., monitored and controlled variables. Thus, the application layer becomes device-independent. IALs and HALs are used to transform the input data into monitored variables, and to transform the controlled variables into output data. The components in the layered architecture are either *Communicating Processes* (active components) or used with a *Call-and-Return* mechanism (passive components).

Repository architecture The repository architecture consists of a central data storage and several client components that use the central data storage to exchange data. The repository architectural style is shown in Fig. 2.11.

Pipe and filter architecture The pipe and filter architecture consists of several sequential components. Each component performs one step of the complete task (also called filter) and the results are transferred to the next component using a pipe (see Fig. 8.7 on Page 131).

The definitions of architectures and architectural styles are the basis of Chapters 8 and 9.

Chapter 3

Basic Development Process

Dependable systems and software should be developed using a structured development process. Such a process must be extended to handle dependability requirements.

In this chapter, a pattern- and component-based development process for software and system development is presented. The process covers analysis, design, implementation and testing and is called ADIT. It emerged from projects dealing for example with

- smart card operating systems and applets for smart cards in the area of security-critical systems,
- a protocol converter that connects a proprietary RS-485-based bus system with a CAN-bus system, and
- motor control and automatic doors in the area of safety-critical systems.

The process consists of a sequence of steps to be performed. As a means of presentation, we use the *agenda* concept introduced in Section 2.2. In each step, a natural-language description or a model (mostly expressed using UML 2.0, (UML Revision Task Force, 2010c)) is developed. In addition, each step has some *validation conditions* associated with it that help to detect errors as early as possible in the process. A first version of this process was published in the paper (Heisel & Hatebur, 2005) and improved versions can be found in (Hatebur, 2006; Heisel & Hatebur, 2008; Hatebur & Heisel, 2009a; Heisel et al., 2011; Choppy, Hatebur, & Heisel, 2011).

This process is based on problem frames (see Section 2.4) and makes used of architectural patterns (see Section 2.8). It covers hardware and software components. The main focus is on embedded software components and the modeling of problems, specifications, tests and architectures.

The chapter is structured as follows: In Section 3.1 we give an overview of a pattern- and component-based development process to develop systems and software. Details about all steps (input, output, validation and procedure) are described in Appendix A. This section also gives an overview of our extension to develop dependable systems and software. Section 3.2 introduces the case study that is used in the following chapters to illustrate the process with the dependability extension. In Section 3.3 we discuss related work and conclude the chapter in Section 3.4.

3.1. Extended ADIT overview

In the following, we describe the development process called ADIT (Analysis, Design, Implementation, Testing) that serves as a basis for our method to handle dependability requirements, described in the following chapters. ADIT is a model-driven, pattern-based development process also making use of components. It is a joint development of the work group Software Engineering at the University of Duisburg- Essen. ADIT consists of four phases, namely analysis, design, implementation, and testing. Each phase is sub-divided into different steps. These steps detail what needs to be done in order to set up the needed development documents for the respective step. We use the agenda concept (see Section 2.2) to arrange the different steps. The structure for each step is as follows: first, we present its agenda. After presenting the agenda, we mention necessary input followed by a procedure that describes how we can construct the requested output from the provided input. Next, we describe the resulting output. In the following, we briefly state the purpose of each step. Details are described in Appendix A.

3.1.1. Analysis Phases

Concerning the analysis, we consider the following steps:

A1 **Problem Elicitation and Description**: To begin with, we need *requirements* that state our needs. Therefore, we need a description of the desired functionality of the software to be built. Out of this description, we can derive the relevant requirements. The requirements are initially expressed in natural language. In this step, we also state *domain knowledge*, which consists of *facts* and *assumptions*. We must find an answer to the question: “Where is the problem located?”. Therefore, the environment in which the software will operate must be described. Thus, the output of this step is a model describing the environment of the software to be built, the requirements and the domain knowledge. We use a *context diagram* (see Section 2.3) for that purpose. In contrast to Jackson, we allow more than one machine domain in the context diagram to , e.g., take distributed systems into account.

A tight integration of requirements analysis and later development steps helps to avoid mistakes. For dependable systems development, mistakes in the development should be avoided. Using the same notation and re-using elements of the analysis support a tight integration. In Chapter 4, we introduce a UML profile that allows us to describe the environment with the same notation as used in later development phases.

For systems considering random and systematic faults a preliminary hazard analysis has to be performed. For systems considering an attacker a threat analysis has to be performed. The analysis results act as an input for the next steps dealing with dependability requirements. This step can be performed in parallel to Step A1 – Problem Elicitation and Description (see Appendix A.1). The output of Step A1 is the input of the analysis step, and the output of the analysis acts as additional input of Step A1. The agenda of the analysis step is shown in the following table:

Input:	requirements R	optative statements
	domain knowledge $D \equiv F \wedge A$	indicative statements
	context diagram of system to be built	extended UML notation
Output:	requirements addressing the hazards or threats	natural language
	necessary risk reduction to be achieved by these requirements	numbers or natural language
Glossary:	-	-
Validation:	All domains in the environment are considered.	check manually
	Faults in or attacks on all functions of the machine are considered.	check manually

A procedure for creating a preliminary hazard analysis is described, e.g., by Mader et al. (2011). A procedure for threat- and risk-analysis during early security requirements engineering is described by Schmidt (2010b). Both are not part of this thesis.

A2 **Problem Decomposition**: We answer the question: “What is the problem?” in this second step. To answer the question, it is necessary to decompose the overall problem described in Step A1 (Problem Elicitation and Description) into small manageable subproblems. For decomposing the problem into subproblems, related sets of requirements are identified first. Second, the overall problem is decomposed by means of the decomposition operators described in Section 4.1.5. These operators are applied to the context diagram. Afterwards, we have constructed our set of subproblems represented as *problem diagrams* (see Section 2.4) as output.

Subproblems can be described as suggested in Chapter 4.

When the problem is decomposed into smaller subproblems, the dependability requirements can be documented in the same way as the other requirements by using the problem diagram notation we introduce in Chapter 5. In Chapter 5, we also introduce pattern that can be used to express dependability requirements. The patterns allow one to represent each dependability requirements as text, a predicate, and an extension of a problem diagram. If a requirement fits to a certain pattern, a solution approach (generic mechanism) associated to this pattern can be selected in order to identify inconsistencies and missing requirements. We describe the detailed procedure in Chapter 6. The agenda described in Appendix A.2 has to be extended as follows:

Input:	all results of Step Problem Elicitation and Description	
Output:	set of subproblems (for functional requirements)	extended UML notation
	dependability requirements	predicates, natural language, and extended UML notation
	generic mechanisms that solve the dependability problem	natural language
Glossary:	machine domains of subproblems	natural language
	new phenomena and domains (if introduced)	natural language
	name of composed requirements	natural language
Validation:	*see Appendix A.2*	

A3 **Abstract Machine Specification**: In the previous step, we were able to find out what the problem is by means of problem diagrams. However, problem diagrams do not state the order in which the actions, events, or operations occur. Furthermore, we are still talking about requirements. Requirements refer to problem domains, but not to the machine, i.e., the system to be built. Therefore, it is necessary to transform requirements into specifications. The procedure is illustrated in Section 7.1. We use UML *sequence diagrams* (see Section 2.5) as models for our specifications. Sequence diagrams describe the interaction of the machine with its environment. Messages from the environment to the machine correspond to *operations* that must be implemented. These operations will be specified in detail in Step A5 (Operations and Data Specification). The output of this step is a set of specifications for each subproblem represented as sequence diagrams.

The specification can be derived from dependability requirements in the same way as for functional requirements, described in Chapter 7.

For some dependability requirements with selected generic mechanisms, the specification can be derived by using patterns. Section 7.3 describes how to systematically re-use behavioral descriptions that are defined for some of the solution approaches that can be selected in Step A2 – Problem Decomposition. The procedure for creating a specification to fulfill dependability requirements has to be described by detailed rules and patterns. Some rules and patterns are presented in Chapter 7.

A4 **Technical Context Diagram**: In this step, the technical infrastructure in which the machine will be embedded is specified. For example, an adaptive cruise control system may use the speed signal of the car that is provided by the controller processing the wheel pulses. The notation for the model in this step is the same as in Step A1 (Problem

Elicitation and Description), namely a context diagram. As it describes the technical means used by the machine for communicating with its environment, we refer to it as *technical context diagram*. The technical context diagram is introduced in Section 4.1.

For dependability requirements it is important to describe the technical environment since the technical environment may have an influence on dependability. Nevertheless, there is no difference in the procedure compared to functional requirements.

A5 **Operations and Data Specification**: The purpose of this step is to set up the necessary internal data structures represented as analysis *class diagrams* (see Section 2.5). Furthermore, we specify the operations identified in Step A3 (Abstract Machine Specification) by providing pre- and postconditions for each relevant operation. We use OCL (see Section 2.7) to express these operation specifications. The class diagrams as well as the operation specifications constitute the two output elements of this step.

The specification of operations and data supports a structured development process, and a structured development process is necessary for the development of dependable systems.

A6 **Machine Life-Cycle**: In this analysis step, the overall behavior of each machine is specified. We use life-cycle expressions proposed by Coleman et al. (1994) to describe this behavior. This means that we explicitly express the relation between the sequence diagrams associated with the different subproblems. The software life-cycle is the output of this step.

Expressing the life-cycle of the machine supports a structured development process, and a structured development process is necessary for the development of dependable systems.

3.1.2. Design Phases

For the design phase, we propose the following steps:

D1 **Machine Architecture**: Setting up a software architecture is our first design step. We divide it into three sub-steps.

a) The first sub-step, **Initial Architecture**, is used to provide a coarse-grained overview by deriving an initial architecture from the given problem descriptions.

b) In the second sub-step, **Implementable Architecture**, the initial architecture is enriched by further details resulting in an implementable architecture.

c) The third step, **Re-structure Software Architecture**, is optional. It is executed whenever a specific architectural style is desired. The implementable architecture is then re-structured in such a way that it adheres to the desired architectural style.

A procedure for creating the architecture for machines with functional requirements is presented in Chapter 8.

We extend this procedure in Chapter 9. It describes the handling of machines consisting of several hardware components (e.g., for hardware redundancy) and takes dependability requirements explicitly into account. It shows how to make use of design patterns for solving dependability requirements and how to annotate quality stereotypes that serve as hints for the person implementing the system. The solution approach selected in Step Problem Elicitation and Description can also help to find appropriate dependability components to be used for the architecture. In general, input, output, glossary, validation and procedure described in the agenda in Appendix A.7, do not change. The output may be extended by annotated quality stereotypes.

D2 **Inter-Component Interaction**: We describe the interaction of the components contained in the architecture. The signals specified in the interfaces of the architecture (see Step D1 - **Machine Architecture**) are used to annotate the abstract sequence diagrams from Step A3. The sequence diagrams developed in this phase are a concrete basis for the test implementation for all components.

For dependability requirements it is important to describe the component specifications. It may be possible that specific patterns can be applied also for this step. Nevertheless, there is no difference in the procedure compared to functional requirements. Deriving the component specifications is state of the art and not covered by this thesis.

D3 **Intra-Component Interaction**: Some of the components of the architecture may be considered as complex, i.e., they are composed of several other components. Therefore, we specify the structure and the behavior of these complex components in more detail. We use UML composite diagrams or class diagrams to represent the structure, and we use sequence diagrams to represent the behavioral descriptions.

For dependability requirements it is important to describe the detailed design. Nevertheless, there is no difference in the procedure compared to functional requirements. Deriving the detailed design is state of the art and not covered by this thesis.

D4 **Complete Component or Class Behavior**: At the end of this last step of our design phase, we have enough detail to directly transform the results of this step into an implementation. We achieve this by providing a complete internal behavioral description of relevant software components and classes. As a means of representation we use UML *state machines* (see Section 2.5).

Expressing the internal behavioral supports a structured development process, and a structured development process is necessary for the development of dependable systems.

3.1.3. Implementation Phase

For the implementation phase, we propose the following step:

I1 **Implementation and Unit Test**: In this step, the components are implemented and connected according to the specified architecture. The task is to transform the design into a correct implementation (with respect to the specifications). This step is relatively straightforward, as most complex design decisions have already been made in the design steps. Note that ADIT is not directed at a particular programming language. Instead, general rules for implementation with object oriented (OO) programming languages are presented. During implementation, unit test are created in order to perform repeatable tests of the implemented methods. These tests are systematically derived from specifications given in Steps D2 (Inter-Component Interaction) and D3 (Intra-Component Interaction).

In general, the implementation of software with dependability requirements can be performed in the same way as for software with only functional requirements. To avoid systematic faults, standards give dedicated rules for the implementation of safe and secure software. Some rules for the implementation are presented in Chapter 10.

3.1.4. Test Phases

For the testing phase, we propose the following steps:

I1 **Component Tests**: In this step, the implemented components are tested according to the specifications given in Step D2 (Inter-Component Interaction) and D3 (Intra-Component

Interaction). The life-cycle from Step A6 (Machine Life-Cycle) helps to execute the tests in an appropriate order.

I2 **System Test**: The system test verifies the machine as a black box. The integrated components together are tested against the specifications of Step A3 (Abstract Machine Specification) under consideration of the life-cycle from Step A6 (Machine Life-Cycle).

I3 **Acceptance Test**: The acceptance test verifies the machine in the intended usage environment. The test descriptions with acceptance criteria are specified based on Step A2 (Problem Decomposition).

In general, the testing of software with dependability requirements can be performed in the same way as for software with only functional requirements. An advantage of the presented development process is that the environment models created in Step A1 (Problem Elicitation and Description) can also be used for test case generation. If the behavior of the system mainly depends on the states of machine and environment, and many test cases should be performed, an approach as presented in Chapter 11 can be used. Even if additional penetration or fault-injection tests have to be performed to verify dependability requirements, input, output, glossary, validation and procedure described in the agenda in Appendix A.14, A.15, and A.16 do not change.

3.2. Case Study

ADIT for dependability is illustrated by a case study of a cooperative adaptive cruise control (CACC) system maintaining string stability (see Hatebur and Heisel (2009b)). Such a system controls the speed of a car according to the desired speed given by the driver and the measured distance to the car ahead. It also considers information about speed and acceleration of the car ahead which is sent using a wireless network[1]. We selected this case study because for such a system, we have safety requirements as well as security requirements. In the case study, we have to reduce the risk of unintended braking and acceleration caused by internal faults and also the risk of braking and acceleration initiated by an attacker.

3.3. Related work

Many processes for systems and software development were published, e.g., Rational unified Process (Jacobson, Booch, & Rumbaugh, 1999), V-Model (Dröschel & Wiemers, 1999), Cleanroom Software Engineering (Prowell, Trammell, Linger, & Poore, 1999), Scrum (Schwaber, 2004). All these processes are on another level of detail. None of them describes detailed steps with input, procedure, output and validation. Our idea is to provide more detailed guidance by giving this information.

Standards like Common Criteria (International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC), 2009a) and ISO 26262 (International Organization for Standardization (ISO), 2011) define documents to be created for a certification. In contrast to this thesis, they do not integrate safety and security and do not give a detailed procedure how to create the necessary documents.

3.4. Conclusion

In this chapter, we have sketched the development process ADIT for functional requirements and outlined how this development process can be extended for dependability requirements. The

[1] cf. United States Patent 20070083318

development of systems with dependability requirements is mainly supported by the following aspects:

- To express dependability requirements adequately, the environment must be described. This is already done by standard ADIT.
- For dependable systems with safety and security requirements, a preliminary hazard analysis or a threat analysis has to be performed. Such analysis can be easily integrated into ADIT.
- ADIT supports the requirements engineering for functional requirements with patterns (problem frames). In this chapter, we mentioned that patterns can also be used for expressing dependability requirements. If a requirement fits to a certain pattern, a solution approach associated to this pattern can be selected. The solution approach can help to identify conflicting or missing requirements.
- For dependable systems, approved solutions should be re-used. For some solution approaches associated to problem patterns, specifications (behavioral descriptions) are given and can be re-used.
- With the extended ADIT, re-use is not only possible on the level of specifications, but also on the level of implementation. In this chapter, we mentioned that the selected solution approach can also help to find appropriate dependability components.
- The technical infrastructure may have an influence on dependability. The description of technical infrastructure is already done by standard ADIT.
- In this chapter, we mentioned that the implementation can be supported by dedicated rules that avoid systematic faults.
- For dependable systems, the quality needs to be demonstrated. The environment models can also be used for generating a huge number of test cases.
- A structured development process as proposed with ADIT helps to avoid mistakes. This is a great importance for the development of dependable systems.

The process described here is a heavyweight process since many outputs have to be created and these outputs are tightly integrated, i.e., even small changes in one output document lead to many other outputs to be updated. Such a tight integration is necessary to ensure a high degree of correctness and consistency. The problem with the impact of small changes can be addressed by tool support that at least indicates all inconsistencies or by using a single model and reducing redundant model elements.

An important task is to develop a method that also allows the developers to evolve existing systems and not only develop new systems from scratch. It is also necessary to have a better integration of preliminary hazard analysis and threat analysis into the development process.

Chapter 4

UML Profile for Requirements Engineering

The development of dependable systems can be supported by a solid tool support and a tight integration of the requirements to later development steps. Currently, Jackson's requirements engineering process is not equipped with solid tool support. This chapter aims at providing tool support for this requirements engineering process. Instead of developing a new tool from scratch, we decide to use UML (UML Revision Task Force, 2010c) and extend it in a way that it can be used to express at least Jackson's (Jackson, 2001) context diagrams, problem diagrams, and problem frames. This is possible by adding profiles to the UML meta-model by defining stereotypes and constraints. The UML profile allows us to express the different diagrams occurring in the problem frame approach using UML notation. The diagrams are mapped to parts of a global model and a graphical representation.

In addition, we augment the original problem frame notation (see Section 2.4) by concepts known in UML, such as aggregation and multiplicities, to further enhance the requirements engineering process at hand. According to Jackson's approach, it is not possible to constrain a biddable domain. We can only constrain a connection domain between the machine and the biddable domain. This connection domain displays (this also includes acoustical "displaying") the output of the machine. We have to assume, but we cannot guarantee that the biddable domain notices the displayed information. For domains with these properties, we introduced the display domain as a special causal domain, see Côté et al. (2008). In contrast to Jackson, we allow more than one machine domain in the context diagram to be able to model distributed systems. In case of several machine domains, the subproblems have to be assigned to one of the machines and for each machine an architecture have to be developed. In addition to Jackson's diagrams, we express technical knowledge (that we know or can acquire before we proceed to the design phases) about the machine to be built and its environment in a *technical context diagram* (Hatebur & Heisel, 2009a). To express mandatory behavior of domains in the environment we have introduced *domain knowledge diagrams*. Details about these diagrams are given in Section 4.1.

The Eclipse framework (*Eclipse - An Open Development Platform*, 2008) constitutes an integrated development environment that can easily be adapted to meet the different needs of software engineers. The adaptation is usually achieved by adding plug-ins to the basic Eclipse installation. One plug-in which is useful for modeling purposes is the *Eclipse Modeling Framework* (EMF) (*Eclipse Modeling Framework Project (EMF)*, 2008). The EMF enables engineers to create structured data models compliant to UML. The data models are equipped with meta-data, which can be queried and updated via a Java interface. EMF stores the model information using the XML Meta-data Interchange (XMI) (UML Revision Task Force, 2007) format. Another interesting plug-in is the one for the Object Constraint Language (OCL) (UML Revision Task Force, 2010a). With OCL it is possible to formally specify constraints over a given model.

In order to automatically validate the integrity and coherence of the different diagrams in the model, we provide integrity conditions, expressed as OCL constraints. Based on the model information, our tool, which is called UML4PF, can automatically detect semantic errors in the model by evaluating these constraints. UML4PF is based on the Eclipse development environment, extended by an EMF-based UML tool, in our case, Papyrus UML. We also show, how consistency between patterns (problem frames) ans their instances (problem diagrams) can be ensured.

The results of the requirements analysis with problem frames should be easily re-usable in later phases of the development process. Since UML is a widely used notation to express analysis and design artifacts in a software development process, a seamless integration of Jackson's requirements engineering process is desirable. By translating the original problem frame diagrams to UML diagrams, we show how this integration can be achieved. We can also validate that the artifacts of later development steps are consistent with the requirements engineering diagrams.

We illustrate our profile and validation conditions by the case study of a cooperative adaptive cruise control (CACC) system. Parts of this work are published in (Hatebur & Heisel, 2010a). Other parts are based on joint work with Côté, Hatebur, and Heisel (2008).

The chapter is structured as follows: In Section 4.1, our UML profile is presented. It shows how the original diagrams can be translated to UML diagrams and how the integrity and coherence of the model can be checked. Section 4.2 describes the actual realization of our tool. Section 4.3 illustrates our profile and validation conditions by the case study of a CACC. Section 4.4 discusses related work. Finally, Section 4.5 concludes this chapter with a summary, ongoing work, and directions for future research.

4.1. UML extension for Jackson Diagrams

To support problem analysis according to Jackson (Jackson, 2001) and ADIT (Heisel et al., 2011) with UML, we created a new UML profile. In this profile stereotypes are defined. A stereotype extends a UML meta-class from the UML meta-model, such as Association or Class. This is expressed by a filled arrow (see Fig. 4.1 on the facing page). In the following sections, we show how the original elements of Jackson's diagrams can be expressed with UML diagrams using the stereotypes defined in our profile. In addition, we set up OCL constraints to check integrity conditions for the different models.

4.1.1. Diagram Types

The different diagram types make use of the same basic notational elements. As a result, it is necessary to explicitly state the diagram type by appropriate stereotypes. In our case, these stereotypes are ≪ContextDiagram≫, ≪ProblemDiagram≫, ≪ProblemFrame≫, ≪DomainKnowledgeDiagram≫ and ≪TechnicalContextDiagram≫. They extend (some of them indirectly) the meta-class Package in the UML meta-model, as depicted in Fig. 4.1 on the next page.

According to the UML superstructure specification (UML Revision Task Force, 2010c), it is not possible that one UML element is part of several packages. Nevertheless, several UML tools allow one to put the same UML element into several packages within graphical representations. We want to make use of this information from graphical representations and add it to the model (using stereotypes of the profile). Thus, we have to relate the elements inside a package explicitly to the package. For that purpose, we introduce the stereotype ≪isPart≫ for dependencies. The dependencies point from the package to all included elements (e.g., classes, interfaces, comments, dependencies, associations).

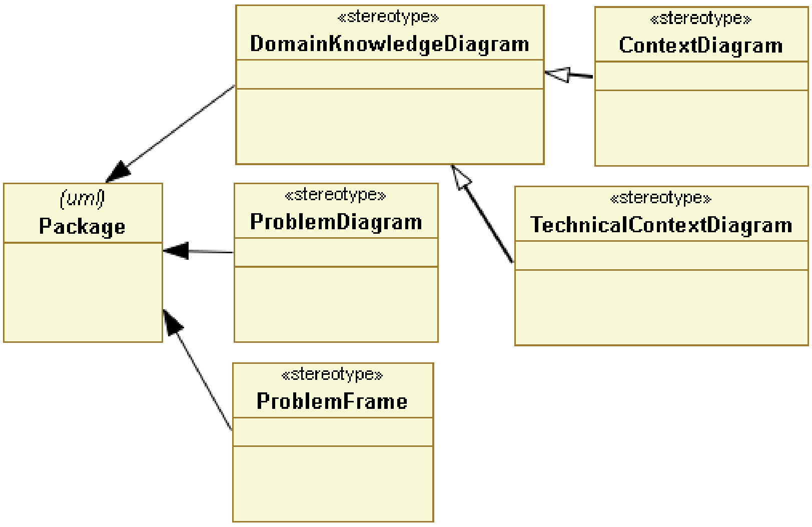


Figure 4.1.: *Diagram Types*

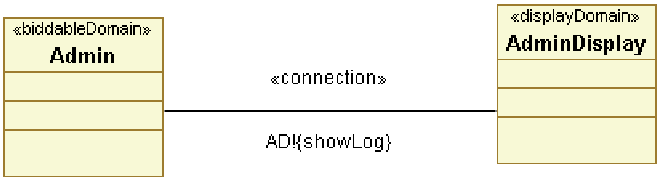


Figure 4.2.: *Interface Class Generation – Drawn*

4.1.2. Associations and Interface Classes

For phenomena between domains, we want to keep the notation introduced by Jackson. Our experience is that this notation is easy to read and easy to maintain. In Jackson's diagrams, interfaces between domains show that there is at least one phenomenon shared by the connected classes. In UML, associations describe that there is some relation between two classes. We decided to use associations to describe the interfaces in Jackson's diagrams. An example for such an interface is depicted in Fig. 4.2. The association AD!{showLog} has the stereotype ≪connection≫ to indicate that there are shared phenomena between the associated domains. The class AdminDisplay controls the phenomenon showLog. In general, the name of the association contains the phenomena and the control direction. To specify the control direction, we use the original notation, i.e., the abbreviation of the domain that controls the phenomena is used, followed by an exclamation mark, and afterwards the set of phenomena is provided.

For large phenomena sets we suggest to use a name instead of the set in the diagram. We then define this set of phenomena in a global comment with the stereotype ≪phenomena≫ (see Fig. 4.19 on Page 44).

Jackson's phenomena can be represented as operations in UML interface classes. The interface classes support the transition from problem analysis to problem solution: Some of the interface classes in problem diagrams become external interfaces of the architecture, and the operations in interface classes must be consistent with the sequence diagram messages. A ≪connection≫ can be transformed into an interface class controlled by a domain and observed by other domains. To express this, the stereotypes ≪observes≫ and ≪controls≫ extend the meta-class Dependency

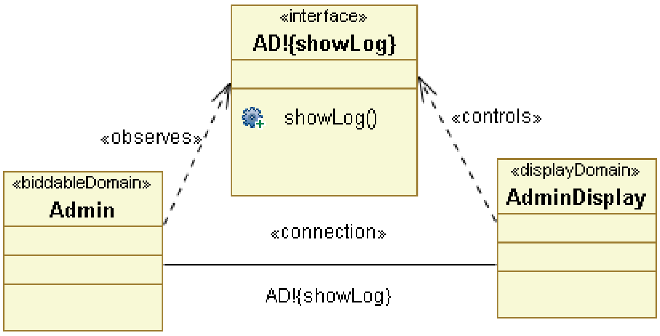


Figure 4.3.: *Interface Class Generation – Transformed*

in the UML meta-model. The interface should contain all phenomena as operations. We use the name of the association (or relevant parts of it) as name for the interface class. Figure 4.3 illustrates how the connection given in Fig. 4.2 on the preceding page can be transformed into such an interface class.

It is possible to add parameters to the operations in the interface classes. The interfaces defined in the analysis can be reused later in the architecture as provided or used interfaces. To support a systematic architectural design, more specific connection types can be annotated. Examples of such stereotypes which can be used instead of ≪connection≫ are, e.g., ≪network_connection≫ for network connections, ≪physical≫ or ≪electrical≫ for physical connections, and ≪ui≫ for user interfaces. The current (extensible) set of connection types is depicted in Fig. 4.4.

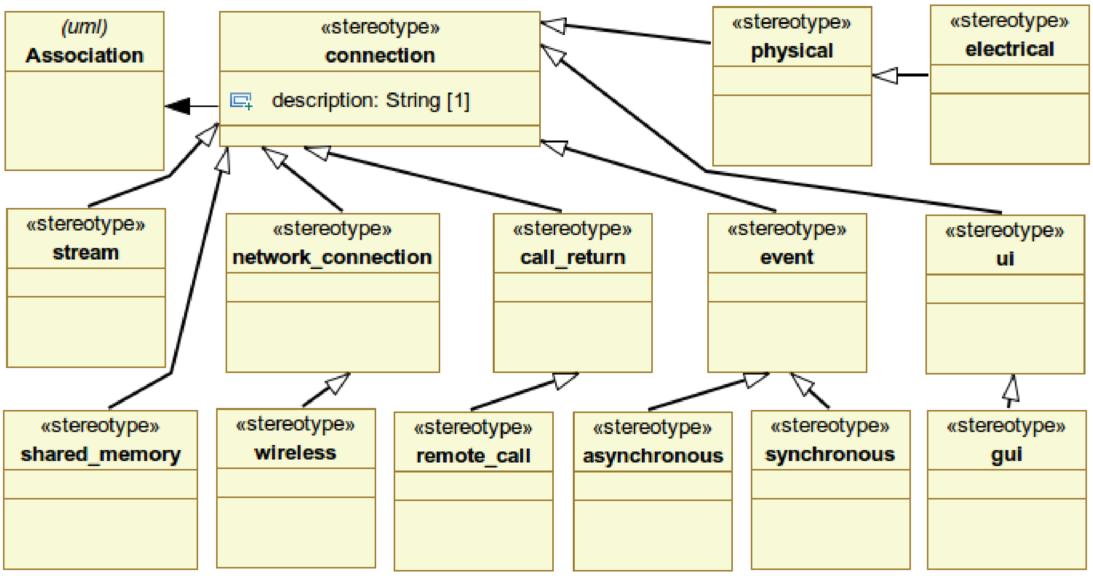


Figure 4.4.: *Connection Types*

To describe given technical realizations, specialized connection types as depicted in Fig. 4.5 on the facing page can be used. Furthermore, it is possible to use multiplicities to add more details about the interfaces.

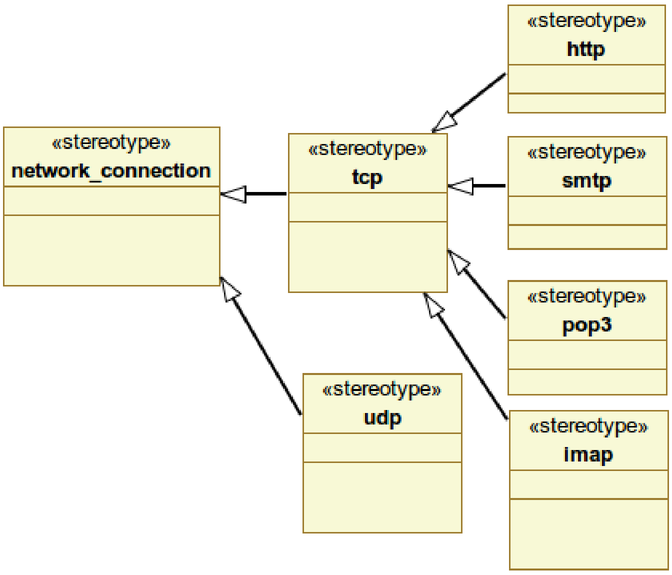


Figure 4.5.: *Technical Connection Types for Network Connections*

4.1.3. Context Diagrams

The context diagram contains the machine domain(s), the relevant domains in the environment, and the interfaces in between. Domains are represented by classes with the stereotype ≪Domain≫, and the machine is marked by the stereotype ≪Machine≫. For each domain, an abbreviation has to be defined to indicate the control of phenomena. Information for the glossary can be inserted into the attribute description of the model elements. Instead of ≪Domain≫, more specific stereotypes such as ≪BiddableDomain≫ and ≪CausalDomain≫ can be used (see Fig. 4.6 on the next page). Each domain is either given, designed or a machine domain. Biddable domains often represent persons, and therefore, they cannot be designed. The machines and designed domains are always causal domains. Domains being neither given nor machine domains should be annotated with the stereotype ≪DesignedDomain≫. For given domains no additional stereotype is used. Other specializations (but not disjoint specializations) of causal domains are lexical domains (data types with storage, marked by ≪LexicalDomain≫) and display domains (used to describe the means used to forward information to a person, introduced in (Côté et al., 2008), marked by ≪DisplayDomain≫). For each domain, the abbreviation and a description should be specified. Since some of the domain types are not disjoint, more than one stereotype can be applied on one class. An overview of the hierarchy of domain types is given in Fig. 4.6 on the following page.

However, not all combinations of stereotypes are permitted. For example, the stereotypes ≪CausalDomain≫ (or subtypes) and ≪BiddableDomain≫ are not allowed to be applied together on one class. Hence, we provide an OCL expression that checks whether this condition is fulfilled. Listing 4.1 depicts the corresponding OCL constraint, which expresses the following: in line 1 and 2, we select the owned elements of the package with the stereotype ≪ContextDiagram≫. Line 3 selects the elements being classes. In line 4, all the classes of the model are selected that satisfy the condition stated within the select-statement. In line 5, we gather the set of stereotypes for each class *cl*. Only those classes should be selected that have the stereotype ≪BiddableDomain≫ or a direct subtype of ≪BiddableDomain≫ **and** the stereotype ≪CausalDomain≫ or a subtype of ≪CausalDomain≫. Unfortunately, it is not possible to iterate through the different inheritance hierarchies of stereotypes with EMF. Therefore, we must explicitly move to each level of inheritance (keyword *general*). As we currently have three hierarchy levels, we

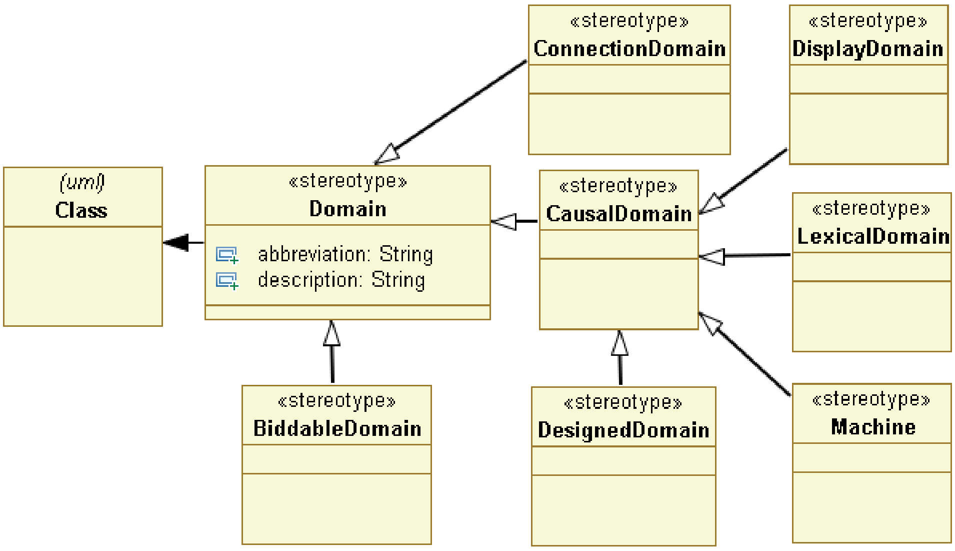


Figure 4.6.: *Domain Types*

limit our constraints to this number (lines 5-10). Note that if new domain types are to be introduced, this limit may need to be adapted. In line 11, we finally check that no class with both stereotypes exist, by comparing the size of the set to 0.

```
1  Package.allInstances() ->select(p |  p.oclAsType(Package).getAppliedStereotypes()
     .name
2          ->includes('ContextDiagram')).clientDependency.target
3  ->select(oclIsTypeOf(Class)).oclAsType(Class)
4  ->select(oe | (
5    oe.oclAsType(Class).getAppliedStereotypes().name ->includes('BiddableDomain') or
6    oe.oclAsType(Class).getAppliedStereotypes().general.name
         ->includes('BiddableDomain')
7  ) and (
8    oe.oclAsType(Class).getAppliedStereotypes().name ->includes('CausalDomain') or
9    oe.oclAsType(Class).getAppliedStereotypes().general.name ->includes('CausalDomain')
         or
10   oe.oclAsType(Class).getAppliedStereotypes().general.general.name
         ->includes('CausalDomain')
11 ) ) ->size()=0
```

Listing 4.1: *The stereotypes ≪biddable≫ and ≪causal≫ are not allowed to appear together*

For a context diagram, the following constraints can be stated:

- We mentioned earlier that all diagram types rely on the same basic notational elements. However, not all existing notational elements are allowed to be used in the different diagram types. In a context diagram, allowed elements are classes with the stereotype domain or a subtype of domain, interfaces, dependencies with the stereotypes ≪isPart≫, ≪observes≫, ≪controls≫ associations with the stereotype ≪connection≫ or a subtype of ≪connection≫, and comments. A corresponding OCL expression is given in Appendix C in Listing C.11.

- In one project we only have one context diagram, as expressed in Appendix C in Listing C.10.

- If we do not have to build anything, the context need not be described. Therefore, the context diagram has at least one machine domain, as expressed in Appendix C in Listing C.12.

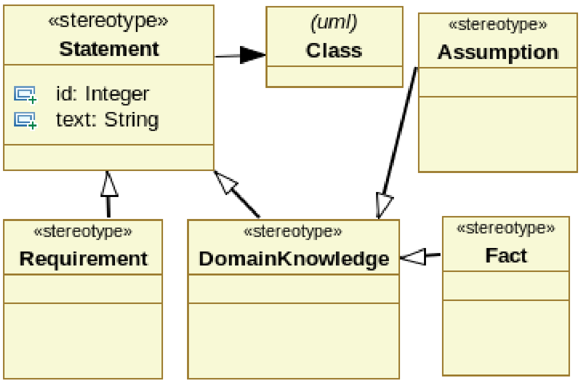


Figure 4.7.: *Statement, Requirement, and Domain Knowledge*

- Since connection domains (see Section 2.3) are used for domains forwarding information, connection domains (in the context diagram) have at least one observed and one controlled interface, as expressed in Appendix C in Listing C.15.

- If a machine should change something in its environment (and otherwise it would not be built), a machine has to control at least one interface, as expressed in Appendix C in Listing C.26.

4.1.4. Problem Diagrams

In a problem diagram, the knowledge for a sub-problem described by a set of requirements is represented. A problem diagram consists of sub-machines of the machines given in the context diagram, the relevant domains, the connections between these domains and a requirement (possibly composed of several related requirements), as well as of the relation between the requirement and the involved domains.

A requirement refers to some domains and constrains at least one domain. This is expressed using the stereotypes ≪refersTo≫ and ≪constrains≫. They extend the UML meta-class Dependency. As shown in Fig. 4.7, domain knowledge and requirements are special statements. Furthermore, any domain knowledge is either a fact (e.g., physical law) or an assumption (usually about a user's behavior).

In a problem diagram, allowed elements are classes, interfaces, associations, dependencies, and comments. In the following, we show an OCL constraint expressing that for some of these elements, only a defined set of stereotypes is allowed. Not allowed in a problem diagram are, e.g. packages, components, or classes without any stereotype. This constraint is shown in Listing 4.2: First, we select all packages that are annotated with the stereotype ≪ProblemDiagram≫ or ≪ProblemFrame≫ (lines 1-3). Second, we select all the elements being part of the package (line 4) that satisfy the conditions for allowed elements using the iterator variable named *el*, i.e., it is a class, an interface, an association, a dependency, or a comment (lines 6, 16, 27, and 37).

- Classes (line 6) being part of the problem diagram package must have a stereotype ≪Domain≫ or a specialized domain stereotype. In lines 7-10 we check that the name of the stereotype is 'Domain' or a subtype of 'Domain'. Classes may also have the stereotype ≪Statement≫ or a subtype such as ≪Requirement≫ (lines 11-15).

- For interface classes (line 16) the usable stereotypes are not restricted.

- Any association (line 17) being part of the problem diagram package represents an interface. Therefore, it must have the stereotype ≪connection≫ or a subtype, e.g., ≪ui≫ for a user interface (lines 18-26).

- The included dependencies (line 27) must be stereotypes ≪observes≫, ≪controls≫, ≪refersTo≫, ≪constrains≫, or ≪isPart≫.
- For technical reasons (PapyrusUML compatibility), Call Events and Profile Application are allowed (lines 35-36).
- For comments (line 37) the usable stereotypes are not restricted.

```
 1 Package.allInstances() ->select(p |
 2   let n: Bag(String) = p.oclAsType(Package).getAppliedStereotypes().name
 3   in  n->includes('ProblemDiagram') or n->includes('ProblemFrame'))
 4       .ownedElement
 5       ->forAll(el | (el.oclIsTypeOf(Class) and
 6         (el.oclAsType(Class).getAppliedStereotypes().name ->includes('Domain') or
 7         el.oclAsType(Class).getAppliedStereotypes().general.name ->includes('Domain')
             or
 8         el.oclAsType(Class).getAppliedStereotypes().general.general.name
             ->includes('Domain') or
 9         el.oclAsType(Class).getAppliedStereotypes().general.general
10                                .general.name ->includes('Domain') or
11         el.oclAsType(Class).getAppliedStereotypes().name ->includes('Statement') or
12         el.oclAsType(Class).getAppliedStereotypes().general.name
             ->includes('Statement') or
13         el.oclAsType(Class).getAppliedStereotypes().general.general.name
             ->includes('Statement') or
14         el.oclAsType(Class).getAppliedStereotypes().general.general
15                        .general.name ->includes('Statement'))) or
16         el.oclIsTypeOf(Interface) or (el.oclIsTypeOf(Association) and
17         (el.oclAsType(Association).getAppliedStereotypes().name
             ->includes('connection') or
18         el.oclAsType(Association).getAppliedStereotypes().general.name
             ->includes('connection') or
19         el.oclAsType(Association).getAppliedStereotypes().general.general
20                                   .name ->includes('connection') or
21         el.oclAsType(Association).getAppliedStereotypes().general.general
22                             .general.name ->includes('connection') or
23         el.oclAsType(Association).endType->forAll(
24           getAppliedStereotypes().name->includes('Statement')  or
25           getAppliedStereotypes().general.name->includes('Statement')))
26         ) or
27     (el.oclIsTypeOf(Dependency) and
28         (el.oclAsType(Dependency).getAppliedStereotypes().name ->includes('refersTo')
             or
29         el.oclAsType(Dependency).getAppliedStereotypes().name
             ->includes('constrains') or
30         el.oclAsType(Dependency).getAppliedStereotypes().name ->includes('controls')
             or
31         el.oclAsType(Dependency).getAppliedStereotypes().name ->includes('observes')
             or
32         el.oclAsType(Dependency).getAppliedStereotypes().name ->includes('isPart') or
33         el.oclAsType(Dependency).getAppliedStereotypes().name
             ->includes('complements'))
34         )or
35     el.oclIsTypeOf(CallEvent) or
36     el.oclIsTypeOf(ProfileApplication) or
37     el.oclIsTypeOf(Comment)
38 )
```

Listing 4.2: *Allowed elements considering the problem diagram/frame*

We also want to ensure, that packages with the stereotypes ≪ProblemDiagram≫ or ≪ProblemFrame≫ contain exactly one machine, represented as a class with stereotype ≪Machine≫. This is expressed in Listing 4.3 using the OCL operator size():

```
Package.allInstances() ->select(p |
  p.oclAsType(Package).getAppliedStereotypes().name ->includes('ProblemDiagram') or
  p.oclAsType(Package).getAppliedStereotypes().name ->includes('ProblemFrame')
) -> forAll (p |
  p.clientDependency ->select(getAppliedStereotypes().name ->includes('isPart'))
  .target->select(pdf_elem |
    pdf_elem .oclIsTypeOf(Class) and pdf_elem
        .oclAsType(Class).getAppliedStereotypes().name ->includes('Machine')
  ) ->size()=1
)
```

Listing 4.3: *A problem diagram/frame has exactly one machine domain*

Since problem diagrams and the corresponding patterns are used to describe requirements, packages with the stereotype ≪ProblemDiagram≫ and ≪ProblemFrame≫ must contain at least one requirement, as stated in the OCL expression in Listing 4.4.

```
Package.allInstances()
       ->select(p | p.oclAsType(Package).getAppliedStereotypes().name
           ->includes('ProblemDiagram') or
                p.oclAsType(Package).getAppliedStereotypes().name
                    ->includes('ProblemFrame') )
                ->forAll(   clientDependency ->select(getAppliedStereotypes().name
                    ->includes('isPart')).target->select(oe |
                    oe.oclIsTypeOf(Class) and
                    oe.oclAsType(Class).getAppliedStereotypes().name
                    ->includes('Requirement'))
                       ->size()>=1 )
```

Listing 4.4: *A problem diagram/frame contains at least one requirement*

Additionally, for problem diagrams and problem frames the following constraints can be stated:

- A requirement does not constrain a machine domain (see Listing C.21 in Appendix C). Requirements should be stated in terms of the environment, and therefore they should not constrain the machine itself.
- Since a machine cannot force a user to do something, a requirement does not constrain a biddable domain (see Listing C.22 in Appendix C).
- As for the context diagram, connection domains in a problem diagram or problem frame have at least one observed and one controlled interface (see Listing C.25 in Appendix C).
- Each machine controls at least one interface (see Listing C.16 in Appendix C).
- The syntactical aspect that dependencies with the stereotypes ≪constrains≫ and ≪refersTo≫ point from statements to domains is stated in Listing C.24 in Appendix C.

4.1.5. Checking the Consistency between Problem Diagrams and Context Diagram

With each problem diagram certain aspects about the system (machine in its environment) are described. The problem diagrams show in which way the requirements refer to domains in the environment and constrain aspects in the environment. The context diagram and the problem diagrams must be consistent. The relation between context diagram and problem diagrams must be made explicitly to support understanding these relations and to allow automatic consistency checking.

Decomposition operators can be used to derive subproblem diagrams from the context diagram. For each operator, we provide means for describing the mapping to express the relation between a subproblem diagram and the context diagram. UML4PF (see Section 4.2) also needs this relation to check the consistency automatically. Therefore, we need **mapping diagrams** to make following operations traceable:[1]

[1] Within these mappings, we use avoid the term 'refine' and use the term 'concretize' because we use none of the well-known refinement relations.

- introduce connection domain (mapping using dependencies with stereotype ≪concretizes≫)
- remove connection domain (mapping using dependencies with stereotype ≪concretizes≫)
- combine domain (mapping using aggregations)
- split domain (mapping using aggregations)
- concretize interface (mapping using dependencies with stereotype ≪concretizes≫)
- abstract interface (mapping using dependencies with stereotype ≪concretizes≫)
- combine interface (mapping using dependencies with stereotype ≪concretizes≫ or aggregations)
- split interface (mapping using dependencies with stereotype ≪concretizes≫ or aggregations)
- leave out domain (no mapping nessecary)

The consistency between context diagram and problem diagrams is checked according to the following rules:

1. Each problem diagram machine is part of the context diagram machine, because a subproblem (described by a problem diagram) describes a part of the overall problem to be solved by the machine in the context diagram.
2. All subproblem diagrams are derived from the context diagram by means of decomposition operators. This is checked by the following rules:
   a) Each domain in each of the problem diagrams corresponds to a domain in the context diagram. This is necessary because we are not allowed to invent a new domain if we want to describe subproblems of the overall problem to be solved by the machine in the context diagram. However, it is allowed to introduce connection domains, combine context diagram domains, or split a context diagram domain.

   b) Each connection in each of the problem diagrams corresponds to a connection in the context diagram. Introducing new connections in the problem diagram is not allowed. Otherwise the problem diagram allows more interaction between domains than the context diagram. This condition can be validated on the level of interfaces since connections are translated into interfaces as described in Section 4.1.2.

   c) Each observed interface in each of the problem diagrams corresponds to an observed interface in the context diagram. This condition refines Condition 2b.

   d) Each controlled interface in each of the problem diagrams corresponds to a controlled interface in the context diagram. This condition also refines Condition 2b.

   e) Each controlled or observed interface of the machine in the context diagram corresponds to at least one interface in one of the problem diagrams because it is not allowed to leave out machine interfaces. We allow to leave out context diagram connection domains, combine context diagram domains, or split context diagram domains. Additionally, interfaces can be abstracted, refined, combined or split.

   Hence, it is allowed to leave out domains in one problem diagram, but each context diagram domain that has a direct interface with the machine must be considered in at least one problem diagram.

In the following paragraph we present the realization of Condition 1.

```
1  let m: Set(Class) =
2    Package.allInstances() ->select(getAppliedStereotypes().name
         ->includes('ContextDiagram')) ->asSequence() ->first()
3    .clientDependency.target
4    ->select(getAppliedStereotypes().name ->includes('Machine') or
5            getAppliedStereotypes().general.name ->includes('Machine'))
6    .oclAsType(Class) ->asSet()
7  in
8    m.oclAsType(Class).member
9    ->select(oclIsTypeOf(Property)).oclAsType(Property).type
10   ->select(cm |
11     cm->select(oclIsTypeOf(Class)) .oclAsType(Class).member
12     ->select(oclIsTypeOf(Property)).oclAsType(Property).type
13     ->select(getAppliedStereotypes().name ->includes('Machine') or
         getAppliedStereotypes().general.name ->includes('Machine'))->size()=0
14   )
15   ->union(
16     m.oclAsType(Class).member
17     ->select(oclIsTypeOf(Property)) .oclAsType(Property).type
18     ->select(oclIsTypeOf(Class)) .oclAsType(Class).member
19     ->select(oclIsTypeOf(Property)) .oclAsType(Property).type
20   )
21   ->select(oclIsTypeOf(Class)).oclAsType(Class)
22   ->select(getAppliedStereotypes().name ->includes('Machine') or
       getAppliedStereotypes().general.name ->includes('Machine')) ->asSet()
23   =
24     Package.allInstances() ->select(getAppliedStereotypes().name
         ->includes('ProblemDiagram'))
25     .clientDependency.target
26     ->select(getAppliedStereotypes().name ->includes('Machine') or
         getAppliedStereotypes().general.name ->includes('Machine'))
27     .oclAsType(Class) ->asSet()
```

Listing 4.5: *The submachines of the problem diagrams must be part of the machine(s) of the context diagram*

Listing 4.5 expresses that each machines in the problem diagrams is a part of a machine in the context diagram to realize Condition 1. The set of machines m (line 1) is defined by selecting the package with the stereotype ≪ContextDiagram≫. Since only one context diagram is allowed in the model, we can access this diagram by converting the selected bag of packages into a sequence and taking the first element (line 2). For this package, we collect the targets of all dependencies (with clientDependency and target in line 3). These dependencies include all dependencies with the stereotype ≪isPart≫ as described in Section 4.2. To get the machines being part of the package, we select all classes with the stereotype ≪Machine≫ (line 4) and all classes with a specialized stereotype ≪Machine≫ [2]. The superclass can be accessed by the EMF keyword general (line 5). The selected elements with these stereotypes are classes and we can convert the bag of elements into a set of classes (line 6). For all machines m, we collect all members (e.g., contained classes, connections, ports, operations, properties) (line 8). We select the properties and collect the type of the properties (elements connected with an aggregation or composition or attribute types) in line 9. From this set of machines only consider the machines that are not aggregated or composed of other machines (using member and size()=0, lines 10-13). To these elements we add the elements aggregated or composed indirectly (using union, lines 15-20). We select the elements being classes with the stereotype ≪Machine≫ or a sub-type and remove double classes (with asSet, lines 21 and 22). Lines 23-27 verify that the set of machines determined above is the same (using '=' in line 23) as the set of machines in the problem diagram. They are retrieved by selecting all packages with the stereotype ≪ProblemDiagram≫ (line 24) and by using the dependencies (with the stereotype ≪isPart≫) pointing to the classes with the stereotype ≪Machine≫ or a sub-type (lines 25-27).

[2] We allow a subtype of the stereotype ≪Machine≫ to allow , e.g., an extensions of the profile that distinguishes between that are only software and machines that are hardware and software.

All the conditions omitted in this chapter can be found in Appendix C, Listings C.34 (Condition 2a), C.35 (Condition 2b), C.36 (Condition 2c), C.37 (Condition 2d), and C.38 (Condition 2e).

4.1.6. Problem Frames

Problem frames have the same kind of elements as problem diagrams. The package has the stereotype ≪ProblemFrame≫ instead of ≪ProblemDiagram≫. To instantiate a problem frame, the domains and the requirements have to be replaced by concrete ones. Figure 4.8 shows the problem frame given in Fig. 2.7 on Page 8 in UML notation, using our profile.[3]

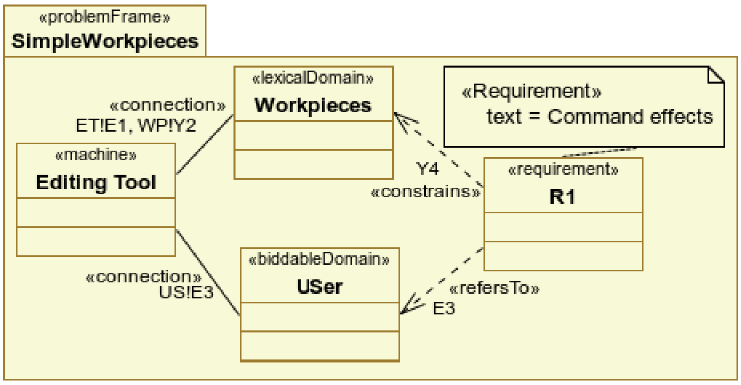


Figure 4.8.: *Simple Workpieces in UML notation*

For problem frames, we have the the same integrity conditions as for problem diagrams (see Section 4.1.4)

4.1.7. Checking the Correct Instantiation of Problem Frames

In this subsection, we present a number of OCL constraints that can be used to check if a given problem diagram is a correct instantiation of a given problem frame. Such checks are very important, because a software development problem only belongs to the problem class characterized by the problem frame if it really exhibits all characteristics required by the frame. Only then can the solution approaches associated with the problem frame be successfully applied.

For each problem diagram, we explicitly state which problem frame it instantiates by using a dependency with the stereotype ≪instanceOf≫. The OCL expression in Listing 4.6 checks if the stereotype ≪instanceOf≫ is used correctly. To this end, all dependencies in the model (Line 1) with the stereotype ≪instanceOf≫ (accessed by the EMF keyword getAppliedStereotypes) (Line 2) are selected. For these dependencies, the source and the target must be a package (checked by the EMF expression oclIsTypeOf(Package) (Lines 3 and 4), the source package has the stereotype ≪ProblemDiagram≫ (Line 5), and the target package has the stereotype ≪ProblemFrame≫ (Line 6).

[3] We use capital letters to indicate the abbreviation, e.g., the abbreviation "US" is used for "USer".

```
Dependency.allInstances() ->select(a |
a.oclAsType(Dependency).getAppliedStereotypes().name ->includes('instanceOf') )
    ->forAll(d |
  d.oclAsType(Dependency).source ->forAll(oclIsTypeOf(Package)) and
  d.oclAsType(Dependency).source.getAppliedStereotypes().name
      ->includes('ProblemDiagram') and
  d.oclAsType(Dependency).target ->forAll(oclIsTypeOf(Package)) and
  d.oclAsType(Dependency).target.getAppliedStereotypes().name
      ->includes('ProblemFrame')
)
```

Listing 4.6: *'instanceOf'-Dependencies are only from ProblemDiagram to ProblemFrame*

If a problem diagram correctly instantiates a problem frame, possible solutions defined for the problem frame can be reused for the concrete problem. For example, corresponding architectural patterns (Choppy, Hatebur, & Heisel, 2005) can be applied.

For security-related problems (see, e.g., (Hatebur et al., 2007a)), we are not allowed to add additional interfaces, whereas for other software development problems, additional domains and interfaces are allowed to be added to the problem diagram. Therefore, we distinguish between two kinds of instances, namely *strict* instances for security-related problems and *weak* instances for other subproblems.

We now present a set of conditions that should evaluate to true if a given problem diagram is a valid instantiation of a given problem frame. These OCL constraints are one of the contributions of this chapter. Some of them we have derived from the informal explanations given by Jackson (Jackson, 2001), for example Conditions 1 and 7 given below. With these conditions (together with the rules given in (Hatebur, Heisel, & Schmidt, 2008b)), we provide the problem frame approach with a formal semantic underpinning. Other conditions express general rules about correctly instantiating patterns, e.g., Conditions 2, 3, and 6. All conditions are decidable, because they check semantic properties of problem descriptions that are expressed as syntactic properties of the corresponding UML model.

Our UML profile is not an exact match of the problem frame approach, but provides several enhancements (e.g., technical context diagram, domain knowledge diagrams, multiplicities). An additional enhancement is the distinction between weak and strict instantiations. Condition 5 states how a weak instance is distinguished from a strict one.

For a given problem diagram to be a valid instance of a given problem frame, the following conditions should evaluate to `true`:

1. The domain types of the constrained domains in the problem frame are the same as in the problem diagram.

2. Each domain referred to by the requirement in the problem frame corresponds to a domain in the problem diagram, i.e., they have the same domain types.

3. Each connection in the problem frame corresponds to a connection in the problem diagram, i.e., they connect the same domain types.

4. For strict instances, each connection in the problem diagram corresponds to a connection in the problem frame, i.e., they connect the same domain types.

5. The domain types in problem diagrams and problem frames are consistent: the number of domains of each type in the problem frame is equal to the number of this type in the problem diagram. In case of a weak instance, the number of domains of each type in the problem frame is smaller than or equal to the number of this type in the problem diagram.

6. For strict instances, the directions of the interfaces (observed vs. controlled) are the same in the problem diagram and the problem frame, i.e., we allow that interfaces are left out.

7. Interfaces cannot be left out if they are controlled by the machine.

In the following, we present the OCL expressions checking a selection of these conditions.

Condition 1. In the OCL expression of Listing 4.7, all dependencies in the model (line 1) with the stereotype ≪instanceOf≫ (line 2) are selected. For these dependencies (line 3) the parts of the target (the problem frame) being requirements (lines 4 and 5) are selected. For these requirements, the dependencies with the stereotype ≪constrains≫ are selected (line 6). dependencies are the constrained classes, and the bag of their stereotype names (line 7) must be the same (line 9) as the bag of stereotype names of constrained domains in the problem diagram (lines 10-4). The domain types ≪ConnectionDomain≫ and ≪DesignedDomain≫ are ignored for this check (lines 8 and 15). We do not allow that a domain subtype is used in the problem diagram, because some known problem frames use a domain subtype of other problem frame. For example, the lexical domain in the Simple Workpieces problem frame can be seen a causal domain like given in a Commanded Behaviour problem frame.

```
1  Dependency.allInstances() -> select(getAppliedStereotypes().name
       ->includes('instanceOf') )
2  ->forAll(
3    target.oclAsType(Package)
4    .clientDependency -> select(getAppliedStereotypes().name ->includes('isPart'))
5    .target -> select(getAppliedStereotypes().name
         ->includes('Requirement')).oclAsType(Class)
6    .clientDependency ->select(getAppliedStereotypes().name ->includes('constrains')).
7    target.getAppliedStereotypes().name
8  =
9    source.oclAsType(Package)
10   .clientDependency -> select(getAppliedStereotypes().name ->includes('isPart'))
11   .target -> select(getAppliedStereotypes().name
         ->includes('Requirement')).oclAsType(Class)
12   .clientDependency -> select(getAppliedStereotypes().name ->includes('constrains')).
13   target.getAppliedStereotypes().name
14 )
```

Listing 4.7: *Constrained domain in the problem frame corresponds to a domain in the problem diagram*

Condition 2 can be checked in a similar way. The OCL condition is given in Expression C.41 in Appendix C.

Condition 5. Line 1 in the OCL expression in Listing 4.8 selects all dependencies with the stereotype ≪instanceOf≫. For these dependencies (line 2), we define the boolean variable weak as the value of the attribute weak of this dependency (lines 3-5). We define the bag pf_domain as all domains of the problem frame the dependency points to (lines 6-10). And we define the bag pd_domains as all domains of the problem diagram (lines 11-15). In the following lines we compare the number of domains of each type: For strict instances, the number of domains with the same domain type are equal in the problem diagram and the problem frame (lines 18 and 19). For weak instances (line 20), the number of domains with the same domain type in the problem frame are lower than or equal to the number of domains with the same domain type in the problem diagram (lines 21 and 22).

```
1  Dependency.allInstances() ->
       select(getAppliedStereotypes().name->includes('instanceOf') )
2  ->forAll( inst_of_dep |
3    let weak: Boolean =
4      inst_of_dep.getValue(inst_of_dep.oclAsType(Dependency) .getAppliedStereotypes()
           ->select(name->includes('instanceOf')) ->asSequence()
           ->first(),'weak').oclAsType(Boolean)
5    in
```

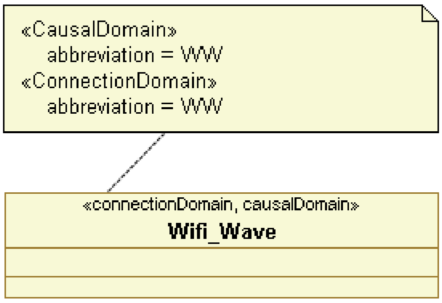


Figure 4.9.: *Domain Abbreviations*

```
let pf_domains: Bag(Class) =
  inst_of_dep.target.oclAsType(Package)
  .clientDependency -> select(getAppliedStereotypes().name->includes('isPart'))
  .target -> select(oclIsTypeOf(Class)) ->reject(getAppliedStereotypes().name
      ->includes('Requirement')).oclAsType(Class)
in
let pd_domains: Bag(Class) =
  inst_of_dep.source.oclAsType(Package)
  .clientDependency -> select(getAppliedStereotypes().name->includes('isPart'))
  .target -> select(oclIsTypeOf(Class)) ->reject(getAppliedStereotypes().name
      ->includes('Requirement')).oclAsType(Class)
in
  pf_domains->forAll(domains |
      domains.getAppliedStereotypes().name ->forAll(stn |
        (pf_domains->select(getAppliedStereotypes().name ->includes(stn))->size() =
        pd_domains->select(getAppliedStereotypes().name ->includes(stn))->size())
        or (weak and
          (pf_domains->select(getAppliedStereotypes().name ->includes(stn))->size()
              <=
          pd_domains->select(getAppliedStereotypes().name ->includes(stn))->size())
        )
      )
  )
```

Listing 4.8: *The domain types in problem diagrams and problem frames are consistent*

The OCL expressions for Conditions 2, 3, 4, and 7 are given in Appendix C, Listings C.41, C.43, C.44, and C.45.

4.1.8. General OCL Constraints

In this section, we present more OCL constraints that should hold for different artifacts of a model:

1. Names of domains, interfaces and statements must be unique. Otherwise, referencing these model elements is quite difficult and it is hard to distinguish if the repeated model elements are the same or different.

2. The abbreviation of the stereotype ≪Domain≫ must be set and identical for all domain stereotypes applied to one class. This is necessary for defining which domain controls a phenomenon annotated at a connector. Figure 4.9 shows a class with two domain stereotypes applied (≪CausalDomain≫ and ≪ConnectionDomain≫). Both have (as required) the same abbreviation (WW).

3. The abbreviation of the domains must be unique. This is also necessary for defining which domain controls a phenomenon annotated at a connector.

4. Dependencies with ≪isPart≫ are only allowed with a package or the model as source. For the composition of classes, the UML composition should be used. For packages and the model, dependencies with ≪isPart≫ should be used to express the composition.

5. A controlled interface must be observed by at least one domain. If an interface is controlled and not observed, it is useless within a model.

6. An observed interface must be controlled by exactly one domain. If several domains control the same interface, priorities need to be defined. Priorities can only be defined by a domain which takes the role of a voter, not by an interface itself. An interface not controlled by a domain is useless within a model.

For the general constrains, the corresponding OCL expressions can be found in Appendix C.1.

We do not claim that the integrity conditions we have defined so far are complete. On the contrary, it is easily possible to identify new conditions and incorporate them into UML4PF. In any case, it is hardly possible to come up with a complete set of semantic integrity conditions that is sufficient for the correctness of the defined models. However, the conditions constitute necessary conditions for the correctness of the defined models. Therefore, a violation of one of the conditions really indicates an error in the development.

4.2. UML4PF - Tool Realization

To work with the profile and the constraints described in Section 4.1, we created the tool *UML4PF*. Figure 4.10 provides an overview of the context of UML4PF. Gray boxes denote re-used components, whereas white boxes describe those components that we created. Basis is the Eclipse platform together with its plug-ins EMF and OCL. Our UML profile is conceived as an eclipse plug-in, extending the EMF meta-model. Eclipse stores profiles in XMI-format. We store all our OCL constraints also in one file in XML-format. This file is generated from the same Latex file that is also the source of Appendix C. With these constraints, we check the validity and consistency of the current model.

The functionality of our tool UML4PF comprises the following:

- It checks if the model is valid and consistent by using our OCL constraints
- It returns the location of invalid parts of the model.
- It automatically generates model elements, e.g., it generates observed and controlled interfaces from association names as well as dependencies with stereotype ≪isPart≫ for all domains and statements being inside a package in the graphical representation of the model.

The graphical representation of the different diagram types can be manipulated by using any EMF-based editor. We selected Papyrus (*Papyrus UML Modelling Tool 1.12*, 2010) as it is available as an Eclipse plug-in, open-source, and EMF-based.

UML4PF can be installed using an Eclipse Update Site located at `http://www.uml4pf.org/plugin/testing` for the development versions and `http://www.uml4pf.org/plugin/uml4pf` for the production releases. The window shown in Fig. 4.11 can be reached with the menu command Help - Install New Software ....

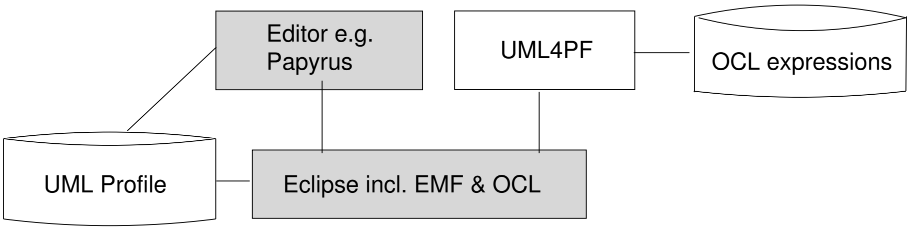
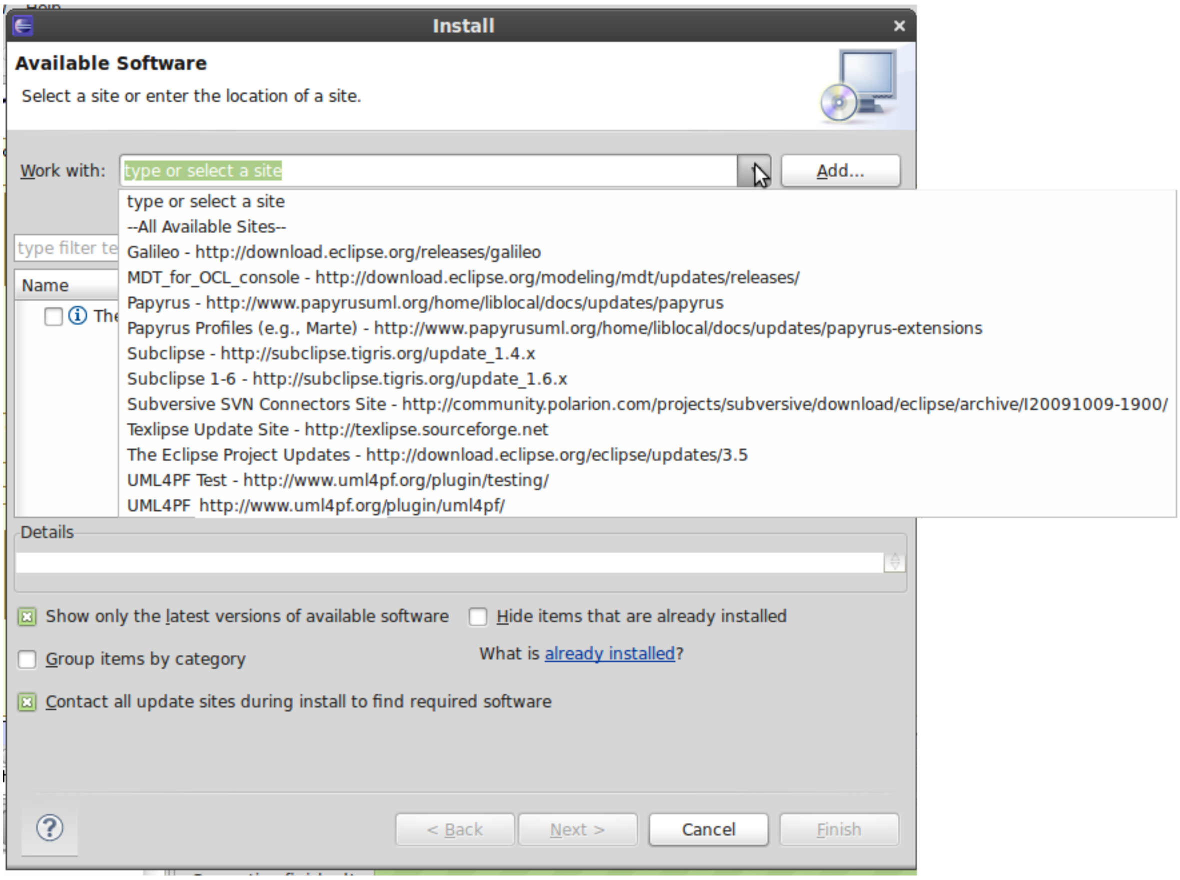


Figure 4.10.: *Tool Realization Overview*


Install

Available Software

Select a site or enter the location of a site.

Work with: type or select a site

type or select a site
--All Available Sites--
Galileo - http://download.eclipse.org/releases/galileo
MDT_for_OCL_console - http://download.eclipse.org/modeling/mdt/updates/releases/
Papyrus - http://www.papyrusuml.org/home/liblocal/docs/updates/papyrus
Papyrus Profiles (e.g., Marte) - http://www.papyrusuml.org/home/liblocal/docs/updates/papyrus-extensions
Subclipse - http://subclipse.tigris.org/update_1.4.x
Subclipse 1-6 - http://subclipse.tigris.org/update_1.6.x
Subversive SVN Connectors Site - http://community.polarion.com/projects/subversive/download/eclipse/archive/I20091009-1900/
Texlipse Update Site - http://texlipse.sourceforge.net
The Eclipse Project Updates - http://download.eclipse.org/eclipse/updates/3.5
UML4PF Test - http://www.uml4pf.org/plugin/testing/
UML4PF http://www.uml4pf.org/plugin/uml4pf/

☒ Show only the latest versions of available software ☐ Hide items that are already installed
☐ Group items by category What is already installed?
☒ Contact all update sites during install to find required software

Figure 4.11.: *UML4PF - Update Site*

When UML4PF is installed, the following new entries appear in the context menu of .uml-files as shown in Fig. 4.12:

- Import UML Package
- Validate Now
- Model Generator

The function to import UML packages can be used to import a problem frame from another UML file, instead of drawing it again for checking if a problem diagram is an instantiation of a problem frame.

When Model Generator is selected in the context menu (Fig. 4.12 on the following page), the different functionalities of the generator can be activated or not. The functionalities are shown in Fig. 4.13.

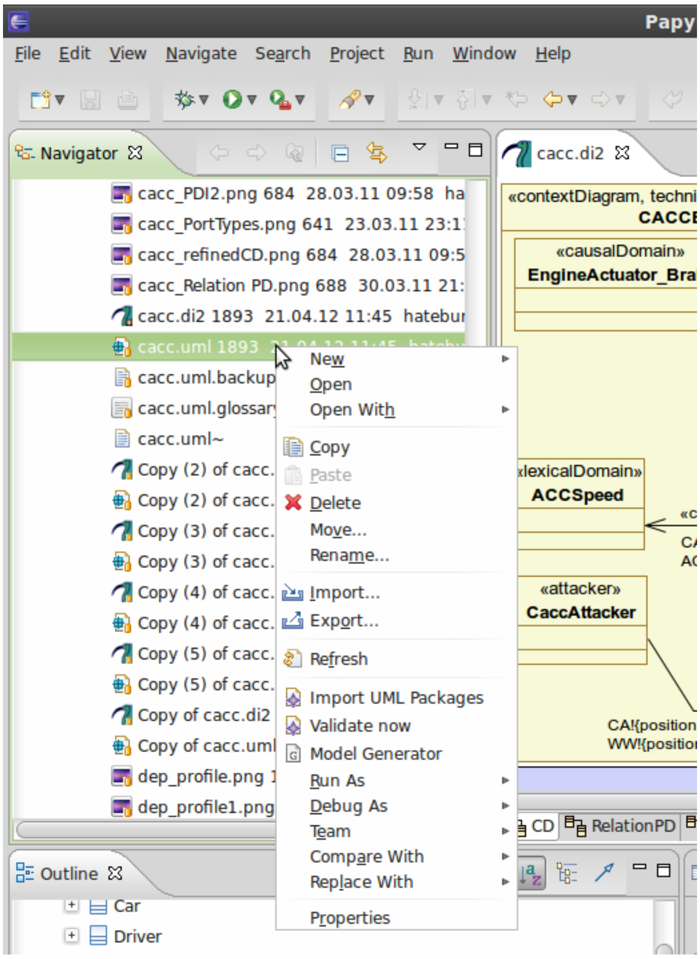
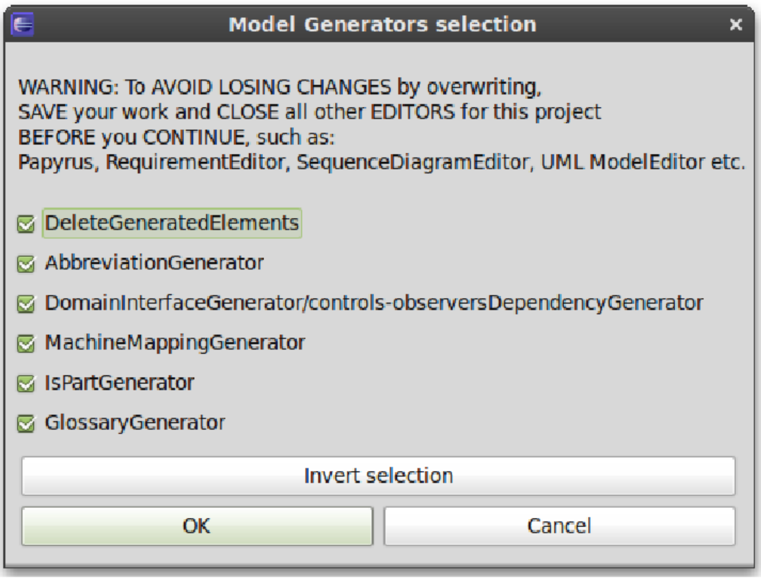


Figure 4.12.: *UML4PF - Context Menu*



Model Generators selection

WARNING: To AVOID LOSING CHANGES by overwriting,
SAVE your work and CLOSE all other EDITORS for this project
BEFORE you CONTINUE, such as:
Papyrus, RequirementEditor, SequenceDiagramEditor, UML ModelEditor etc.

☑ DeleteGeneratedElements

☑ AbbreviationGenerator

☑ DomainInterfaceGenerator/controls-observersDependencyGenerator

☑ MachineMappingGenerator

☑ IsPartGenerator

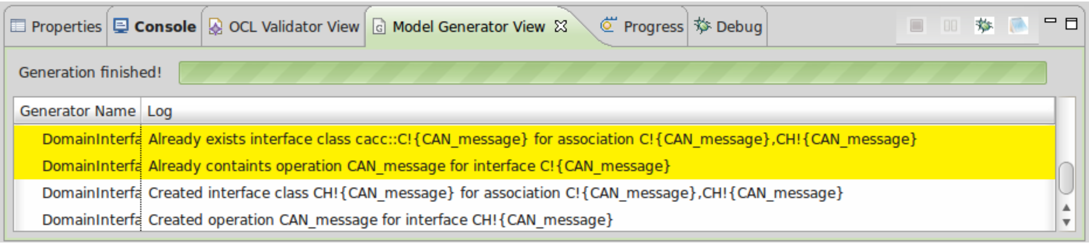
☑ GlossaryGenerator

Figure 4.13.: *UML4PF - Generator Selection*

The AbbreviationGenerator automatically generates abbreviations for classes assigned with the stereotype domain or any of its sub-types. The DomainInterfaceGenerator/controls-observes-

DependencyGenerator generates observed and controlled interfaces from association names. If we have only one machine in the context diagram, the MachineMappingGenerator can automatically create the relations between this machine and those sub-machines found in the problem diagrams. The IsPartGenerator creates dependencies with stereotype ≪isPart≫ for all domains and statements being inside a package in the graphical representation of the model. The GlossaryGenerator generates a glossary containing all relevant model elements.

The performed actions are shown in a separate window as depicted in Fig. 4.14.

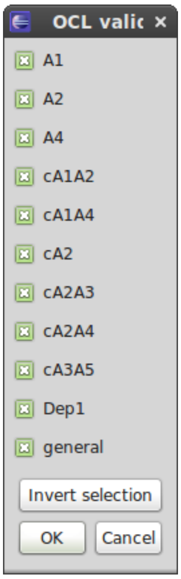


Generation finished!

Generator Name	Log
DomainInterfa	Already exists interface class cacc::C!{CAN_message} for association C!{CAN_message},CH!{CAN_message}
DomainInterfa	Already containts operation CAN_message for interface C!{CAN_message}
DomainInterfa	Created interface class CH!{CAN_message} for association C!{CAN_message},CH!{CAN_message}
DomainInterfa	Created operation CAN_message for interface CH!{CAN_message}

Figure 4.14.: *UML4PF - Generator Output*

When Validate Now is selected in the context menu (Fig. 4.12 on the preceding page), in a selection dialogue (Fig. 4.15), checks for the different ADIT phases and additional checks can be activated or not. With the checkbox A1 in Fig. 4.15 the checks for the ADIT Step A1 – Problem Elicitation and Description (see Appendix A.1) can be activated. With the checkbox cA2 the consistency between problem frames and problem diagrams checks for the ADIT Step A2 – Problem Decomposition can be activated. And with the checkbox cA2A3 the consistency between ADIT Step A2 – Problem Decomposition and Step A3 – Abstract Machine Specification can be activated.


OCL valic

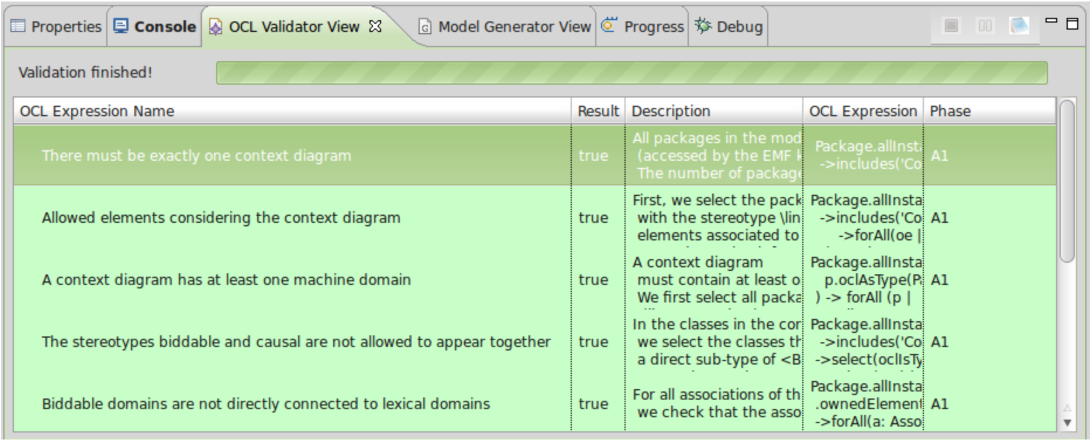
☒ A1
☒ A2
☒ A4
☒ cA1A2
☒ cA1A4
☒ cA2
☒ cA2A3
☒ cA2A4
☒ cA3A5
☒ Dep1
☒ general

Figure 4.15.: *UML4PF - Validator Selection*

When OK is pressed, the validator checks the OCL expressions associated to the selected phases and outputs the results in the window (shown in Fig. 4.16).

If an expressions evaluates to *false*is, additional expressions are evaluated. These expressions return information about the wrong model elements. In Fig. 4.17, the result of an expression that returns all requirements constraining biddable domains is printed.

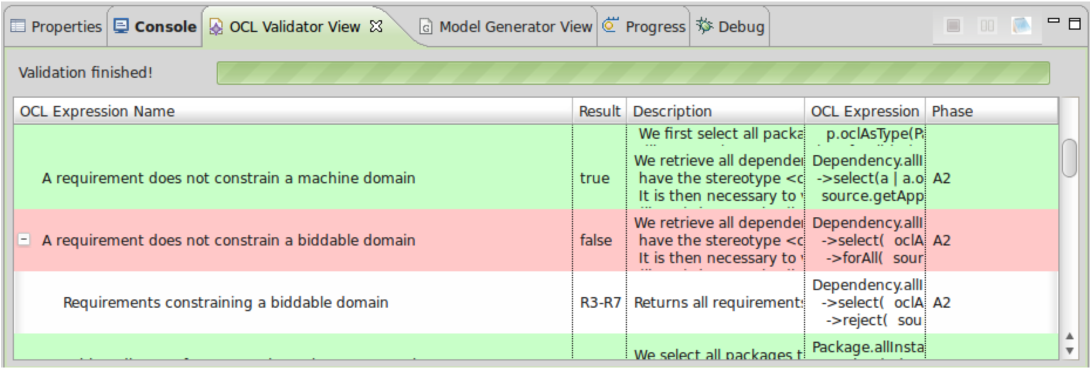
When the developer selects a line in this output, details are shown in a separate window as



Validation finished!

OCL Expression Name	Result	Description	OCL Expression	Phase
There must be exactly one context diagram	true	All packages in the mod (accessed by the EMF k The number of package	Package.allInst ->includes('Co	A1
Allowed elements considering the context diagram	true	First, we select the pack with the stereotype \lin elements associated to	Package.allInsta ->includes('Co ->forAll(oe |	A1
A context diagram has at least one machine domain	true	A context diagram must contain at least o We first select all packa	Package.allInsta p.oclAsType(P) -> forAll (p |	A1
The stereotypes biddable and causal are not allowed to appear together	true	In the classes in the cor we select the classes th a direct sub-type of <B	Package.allInsta ->includes('Co ->select(oclIsTy	A1
Biddable domains are not directly connected to lexical domains	true	For all associations of th we check that the asso	Package.allInsta .ownedElement ->forAll(a: Asso	A1

Figure 4.16.: *UML4PF - Validator Output*

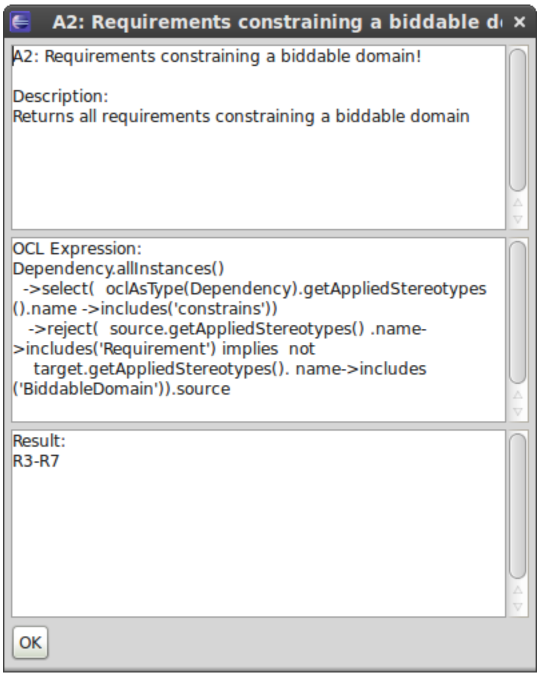


Validation finished!

OCL Expression Name	Result	Description	OCL Expression	Phase
		We first select all packa	p.oclAsType(P	
A requirement does not constrain a machine domain	true	We retrieve all depende have the stereotype <c It is then necessary to	Dependency.all ->select(a | a.o source.getApp	A2
⊟ A requirement does not constrain a biddable domain	false	We retrieve all depende have the stereotype <c It is then necessary to	Dependency.all ->select(oclA ->forAll(sour	A2
Requirements constraining a biddable domain	R3-R7	Returns all requirement	Dependency.all ->select(oclA ->reject(sou	A2
		We select all packages t	Package.allInsta	

Figure 4.17.: *UML4PF - Validator Error*

depicted in Fig. 4.18. This window contains the ADIT phase, the name of the rule, the detailed description, the OCL expression, and the result of this expression.



A2: Requirements constraining a biddable domain!

Description:
Returns all requirements constraining a biddable domain

```
OCL Expression:
Dependency.allInstances()
 ->select( oclAsType(Dependency).getAppliedStereotypes
().name ->includes('constrains'))
  ->reject( source.getAppliedStereotypes() .name-
>includes('Requirement') implies not
   target.getAppliedStereotypes(). name->includes
('BiddableDomain')).source
```

Result:
R3-R7

Figure 4.18.: *UML4PF - Validator Error Details*

4.3. CACC Case Study

In this section, we apply the requirements engineering method and the notations presented in this chapter on the case study introduced in Section 3.2.

The context diagram for the CACC is shown in Fig. 4.19. It also contains the type of connection as stereotypes at the associations between domains (e.g. ≪wireless≫ for wireless connections).

It contains the CACC as the machine to be built. The CACC is part of the Car. The car is connected with a controller area network (CAN) connection (depicted with the stereotype ≪network_connection≫) in order to send messages to the CACC and to receive messages from the CACC. The CACC receives CAN_messages from the Car and sends CAN_messages from the Car. Note that these messages are not identical. The EngineActuator_Brake is also a relevant part of the car. It combines the brake to decelerate (brake) the car and an actuator for the engine to accelerate the car. Corresponding CAN messages can be sent by the CACC. In the environment, we can find the Driver performing the actions defined in the comment with the stereotype ≪phenomena≫. A driver can

- push the brake pedal (brake_pedal),
- push the accelerate pedal (accelerate_pedal),
- press the button to set the current speed as a desired speed (set_speed),
- press the button to increase the desired speed by 5 km/h (increase_speed),
- press the button to decrease the desired speed by 5 km/h (decrease_speed),
- press the button to deacticate the CACC (deactivate), or
- press the button to resume to the last desired speed before deactivation (resume).

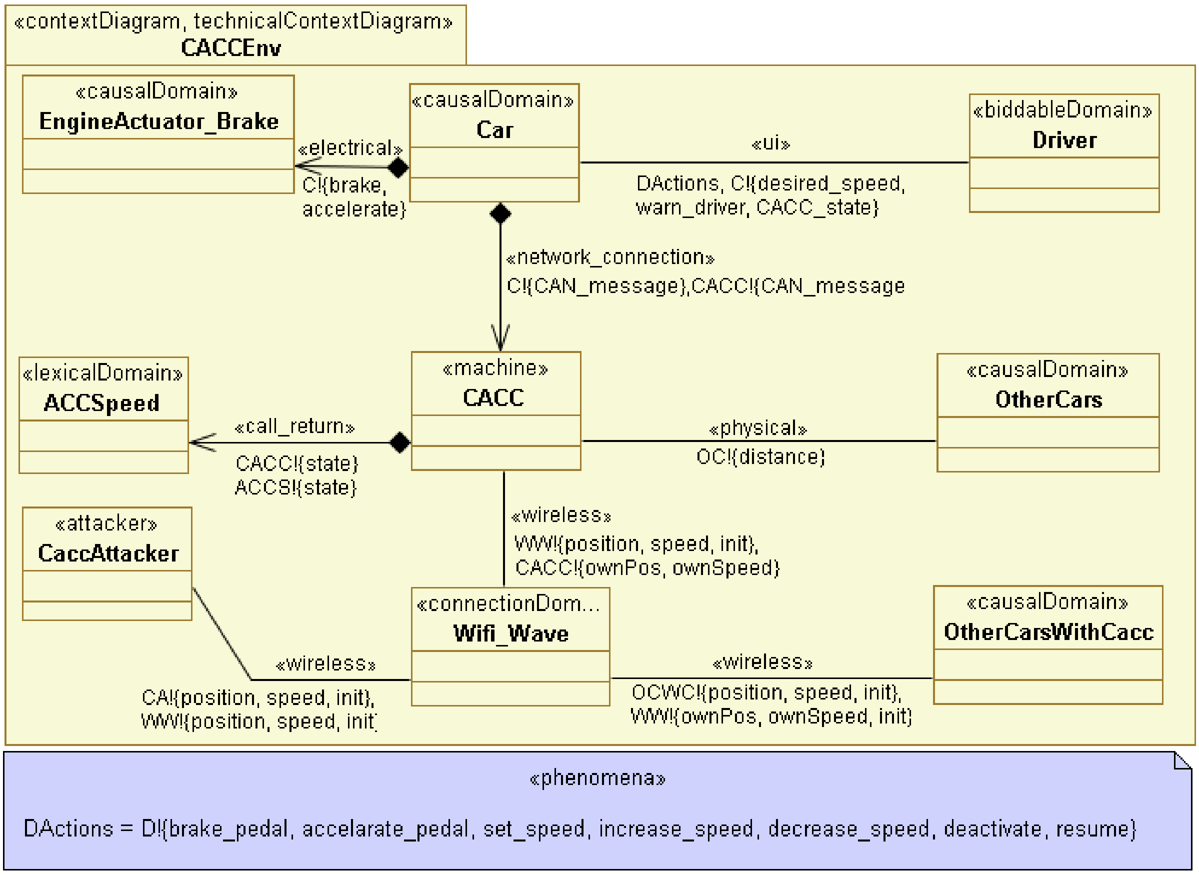


Figure 4.19.: *CACC Context Diagram*

The car displays the desired speed (desired_speed), can warn the driver (warn_driver), and displays the state of the CACC (CACC_state). The state of the CACC and the desired speed are stored in the domain ACCSpeed, which is the internal representation of the driver intention. Storing the state and reading the state is represented by the phenomena named state. Messages corresponding to information at the drivers user interface (≪ui≫) are sent or received by the CACC via CAN. The CACC also measures the distance to the car ahead using a radar. This is modeled as a physical connection with the phenomenon controlled by the car ahead (OC!distance). Other cars that are also equipped with a CACC (OtherCarWithCACC) send their position and speed (OCWC!{position, speed}) using the wireless (≪wireless≫) WIFI or WAVE connection. Additionally, an attacker and the connection domain Wifi_WAVE are introduced here, because this connection may be used by an attacker to control the speed of the car. Details can be found in Section 5.3.

The requirements for the CACC are:

R1 The CACC should accelerate the car if the desired_speed is higher than the current_speed, the CACC is activated and the measured distance and the calculated distance (calculated from the positions of the car itself and the car ahead) to the car(s) ahead is safe.
The CACC should not accelerate if this condition is not given.

R2 The CACC should brake the car if the desired_speed is much (30 km/h) lower than the current_speed, the CACC is activated and the measured or calculated distance to the car(s) ahead is decreasing towards the safe limit.
The CACC should not brake if this condition is not given.

R3 When brake_pedal or deactivate CACC is pressed, the CACC is deactivated and last_speed is set to desired_speed.

R4 When resume is pressed (and resume_speed exists), the CACC is activated and desired_speed is set to last_speed.

R5 When increase_speed is pressed and CACC is activated, the desired_speed is increased by 5 km/h (max.: 200 km/h).

R6 When decrease_speed is pressed and CACC is activated, desired_speed is decreased by 5 km/h (min: 30 km/h).

R7 When set_speed is pressed, desired_speed is set to current_speed.

R8 At that point of time when the CACC is deactivated the driver should be warned, the desired_speed and the CACC_state (CACC is activated or not) should be displayed when the CACC is powered.

R9 CACC should send its own position and speed via Wifi_Wave to OtherCarsWithCACC. [4]

These requirements can be expressed by the problem diagrams depicted in Figures 4.20 (4.21, corrected version), 4.22, and 4.23.

The problem diagram in Fig. 4.20 describes the interfaces between the machine and the environment necessary to implement requirements R1 and R2, e.g., it describes that the machine (a submachine of the CACC in the context diagram) can accelerate the car (CA!{accelerate}), and it describes the relation of the requirements R1 and R2 to the domains in the environment. The requirements constrain the current speed of the car and therefore indirectly its position. The requirements refer to the information in the domains necessary for the described decision, e.g., it refers to the desired speed in the domain ACCSpeed, and it refers to the distance to the car ahead with and without CACC (OtherCar, OtherCarsWithCacc).

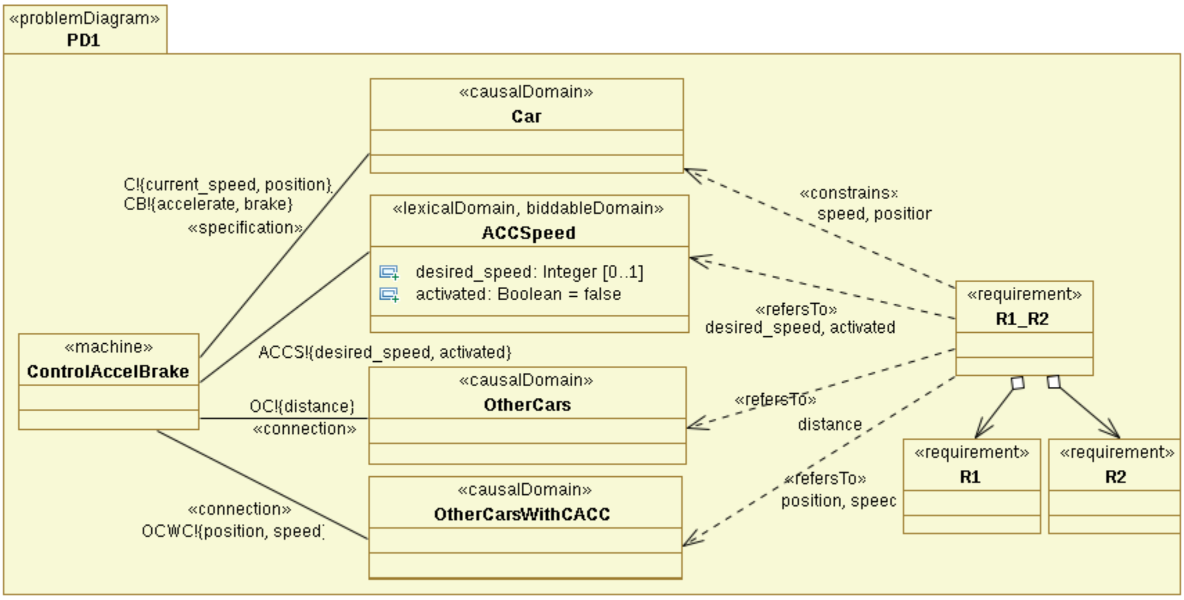


Figure 4.20.: *Erroneous CACC Problem Diagram*

[4] Compared to (Hatebur & Heisel, 2009b), requirements are detailed.

As described in Section 4.1.2, we generated the interface classes as well as the required dependencies using the stereotype ≪isPart≫ (described in Section 4.1.1). After that, we check the OCL constraints given in Section 4.1 on our CACC model. The expressions given in Listings 4.1 on Page 28, C.22 on Page 242 and 4.2 on Page 30 fail. The reasons for that are:

- Listing 4.1 states that the stereotypes ≪CausalDomain≫ (and subtypes) and ≪BiddableDomain≫ are not allowed together in one class. Additionally, it is not allowed that a requirement constrains a biddable domain (see Listing C.22 on Page 242 in Appendix C). Hence, we remove the wrong additional stereotype ≪BiddableDomain≫ from the domain ACCState in the problem diagram.

- The expression in Listing 4.2 defines the allowed elements in problem diagrams. To find out what is wrong in the model, we created debug expressions. For example, we use the expression in Listing 4.2 and replace the forAll with reject. The OCL operation reject removes all elements from a set that fulfill the given condition. Hence, the new debug expression returns all wrong elements in problem diagrams. By inspecting the constraint in Listing 4.2 and the list of wrong elements from the debug expression, we can deduce that the stereotype ≪specification≫ is not allowed for an association (here: C!{current_speed, positions}, CB!{accelerate, brake}) and for the association ACCS!{desired_speed, activated} a stereotype is missing.

A corrected problem diagram is depicted in Fig. 4.21.

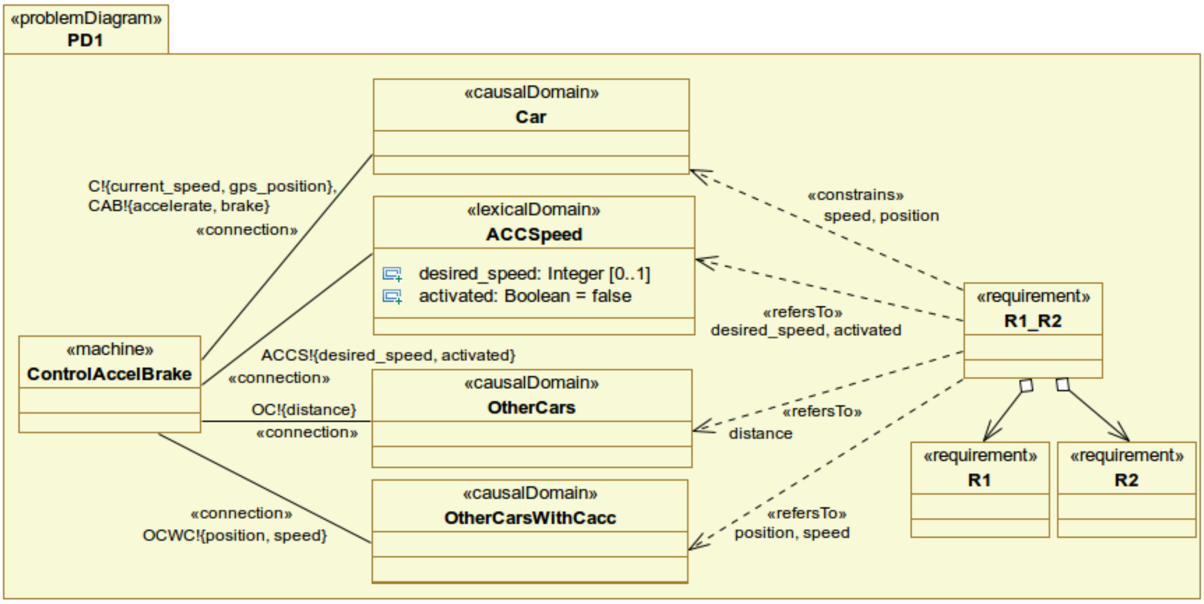


Figure 4.21.: *CACC Problem Diagram 1*

To show the consistency of the problem diagram in Fig. 4.21 to the context diagram, mapping diagrams needed to be provided. Without these mappings, some validation conditions described in Section 4.1.5 fail. The complete graphical representation of the mapping is given in Fig B.1 on Page 232 in Appendix B. The following aspects are relevant for the problem diagram in Fig. 4.21.

- The machine ControlAccelerateBrake is part of the CACC software. Otherwise, Condition 1 of Section 4.1.5 would be violated.

- All domains (Car, ACCSpeed, OtherCars, and OtherCarsWithCACC) are directly taken from the context diagram. Therefore, no mapping is necessary to fulfill Condition 2a of Section 4.1.5

- CB!{accelerate, brake} concretizes CACC!{CAN_message} and ACCS!{desired_speed, activated} concretizes ACCS!{state}. Otherwise, Conditions 2b, 2c, and 2d of Section 4.1.5 would be violated.

The subproblem describing the driver controls (R3 – R7, see Fig. 4.22) constrains the internal representation of the driver intention (ACCSpeed) and refers to the Driver. It also refers to the Car forwarding the driver commands and providing the current speed.

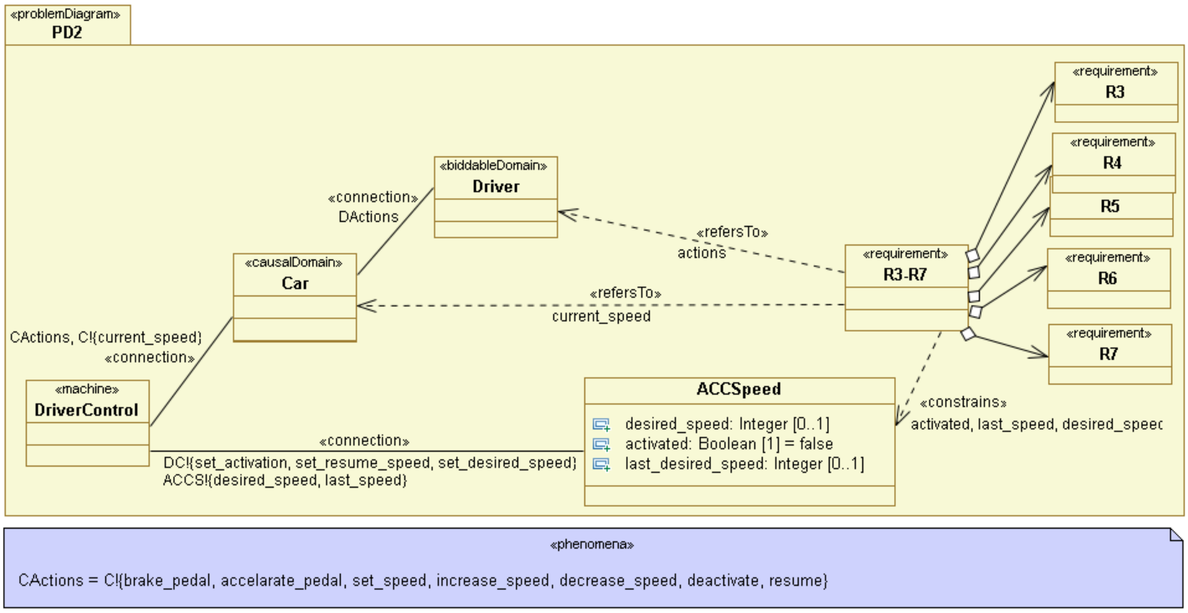

«problemDiagram»
PD2
«requirement»
R3
«requirement»
R4
R5
«biddableDomain»
Driver
«connection»
DActions
«refersTo»
actions
«requirement»
R3-R7
«requirement»
R6
«causalDomain»
Car
«refersTo»
current_speed
CActions, C!{current_speed}
«connection»
«requirement»
R7
«machine»
DriverControl
ACCSpeed
desired_speed: Integer [0..1]
activated: Boolean [1] = false
last_desired_speed: Integer [0..1]
«constrains»
activated, last_speed, desired_speec
«connection»
DC!{set_activation, set_resume_speed, set_desired_speed}
ACCS!{desired_speed, last_speed}
«phenomena»
CActions = C!{brake_pedal, accelarate_pedal, set_speed, increase_speed, decrease_speed, deactivate, resume}

Figure 4.22.: *CACC Problem Diagram 2*

To show the consistency of the problem diagram in Fig. 4.22 to the context diagram, mapping diagrams needed to be provided.

- DriverControl is part of the CACC software. Otherwise, Condition 1 of Section 4.1.5 would be violated.

- All domains (Car, ACCSpeed, and Driver) are directly taken from the context diagram. Therefore, no mapping is necessary to fulfill Condition 2a of Section 4.1.5

- DC!{set_activation, set_resume_speed, set_desired_speed} concretizes CACC!{state}, ACCS!{desired_speed, last_speed} concretizes ACCS!{state}, CActions concretizes C!{CAN_message}, and C!{current_speed} concretizes C!{CAN_message}. Otherwise, Conditions 2b, 2c, and 2d of Section 4.1.5 would be violated.

The subproblem describing the warning and monitoring problem (R8, see Fig. 4.23) refers to the internal representation of the CACC state (ACCSpeed) and constrains the Car. The car has to display the CACC state for the driver.

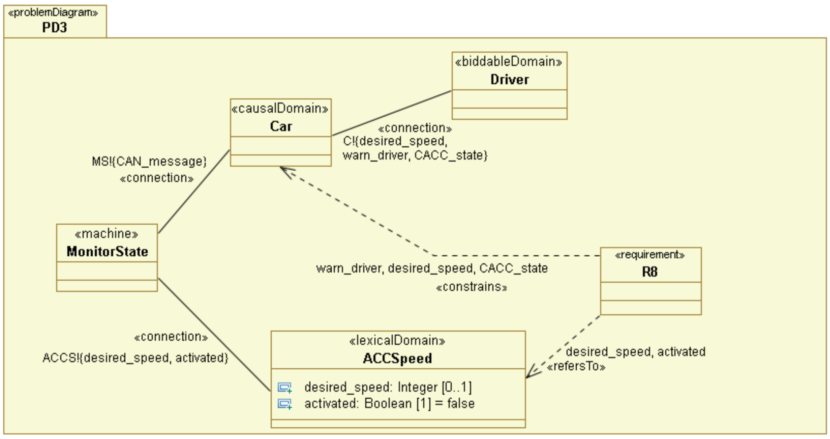


Figure 4.23.: *CACC Problem Diagram 3*

To show the consistency of the problem diagram in Fig. 4.23 to the context diagram, mapping diagrams needed to be provided.

- MonitorState is part of the CACC software. Otherwise, Condition 1 of Section 4.1.5 would be violated.
- All domains (Car, ACCSpeed, and Driver) are directly taken from the context diagram. Therefore, no mapping is necessary to fulfill Condition 2a of Section 4.1.5
- MS!{desired_speed, activated} concretizes CACCS!{state} and MS!{CAN_message} concretizes CACC!{CAN_message}. Otherwise, Conditions 2b, 2c, and 2d of Section 4.1.5 would be violated.

The subproblem describing the sending speed and position problem (R9, see Fig. 4.24) refers to the current speed and position of the car and constrains the information sent via Wifi_Wave.

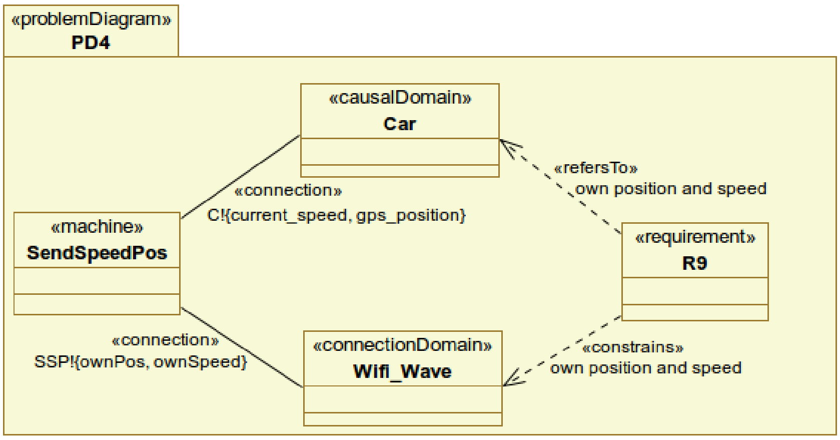


Figure 4.24.: *CACC Problem Diagram 4*

To show the consistency of the problem diagram in Fig. 4.24 to the context diagram, mapping diagrams needed to be provided.

- SendSpeedPos is part of the CACC software. Otherwise, Condition 1 of Section 4.1.5 would be violated.

- All domains (Car and Wifi_Wave) are directly taken from the context diagram. Therefore, no mapping is necessary to fulfill Condition 2a of Section 4.1.5

- SSP!{ownSpeed, ownPos} concretizes CACC!{ownSpeed, ownPos}. Otherwise, Conditions 2b, 2c, and 2d of Section 4.1.5 would be violated.

With all mapping diagrams together (see Fig B.1 on Page 232 in Appendix B), we can show that the context diagram is consistent to the problem diagrams given in Figures 4.21, 4.22, 4.23, and 4.24:

- Each controlled or observed interface of the machine in the context diagram corresponds to at least one interface in one of the problem diagrams (see Condition 2e)

The problem diagram depicted in Fig. 4.21 on Page 46 is a variant of a Required Behaviour problem frame. To fit this problem diagram directly into that frame, we need to merge all problem domains (Car, ACCState, OtherCar, and OtherCarsWithCACC). If ACCState is not merged, it is an instance of the Data-Based Control problem frame, introduced in Section 2.4 on Page 7. If the connection domain Car is removed, the problem diagram depicted in Fig. 4.22 is an instance of the Simple Workpieces problem frame. The problem diagram depicted in Fig. 4.23 on the preceding page is an instance of the Model Display problem frame.

More than 40 OCL constraints were checked using our tool. As a final result, the CACC problem analysis model has been successfully validated.

4.4. Related Work

Lencastre et al. (Lencastre, Botelho, Clericuzzi, & Araújo, 2005) define a meta-model for problem frames using UML. Their meta-model considers Jackson's whole software development approach based on context diagrams, problem frames, and problem decomposition. In contrast to our meta-model, it only consists of a UML class model. Hence, the OCL integrity conditions of our meta-model are not considered in their meta-model. Their approach does not qualify for a meta-model in terms of Model Driven Architecture (MDA) (Warmer & Kleppe, 2003) because, e.g., the class Domain has subclasses Biddable and Given, but an object cannot belong to two classes at the same time (c.f. Figs. 5 and 11 in (Lencastre et al., 2005)).

Hall et al. (Hall, Rapanotti, & Jackson, 2005) provide a formal semantics for the problem frame approach. They introduce a formal specification language to describe problem frames and problem diagrams. As compared to our meta-model, their approach does not consider integrity conditions.

Seater et al. (Seater, Jackson, & Gheyi, 2007) present a meta-model for problem frame instances. In addition to the diagram elements formalized in our meta-model, they formalize requirements and specifications. Consequently, their integrity conditions ("wellformedness predicate") focus on correctly deriving specifications from requirements. In contrast, our meta-model concentrates on the structure of problem frames and the different domain and phenomena types.

We agree with Haley (Haley, 2003) on adding cardinality to standard problem frames to enhance the detailing of shared phenomena at the interfaces. In contrast to Haley though, we do not extend the problem frames notation by introducing a new notational element. We adopt the means provided by UML to annotate problem frames in our meta-model instead.

Van Lamsweerde (Lamsweerde, 2009) considers the relationships between problem worlds and machine solutions. He makes a distinction between different statement subtypes. In our profile we cover a subset of these statement subtypes. Namely requirements, domain knowledge, and specification. Furthermore, he introduces *Satisfaction Arguments*. This satisfaction arguments is similar to the procedure we present in Chapter 7.

Charfi et al. (Charfi, Gamatié, Honoré, Dekeyser, & Abid, 2008) use a modeling framework called *Gaspard2* to design high-performance embedded systems-on-chip. They use model transformations to move from one level of abstraction to the next. To validate that their transformations have been correctly performed, they use the OCL language to specify the properties that must be checked in order to be considered as correct with respect to Gaspard2. We have been inspired by this approach. However, we do not focus on high-performance embedded systems-on-chip. Instead, we target dependable systems development challenges.

Colombo et al. (Colombo, Bianco, & Lavazza, 2008) model problem frames and problem diagrams with SysML. They state that "*UML is too oriented to software design; it does not support a seamless representation of characteristics of the real world like time, phenomena sharing [...]*". We do not agree with this statement. So far, we have been able to model all necessary means of the requirements engineering process using UML.

SysML (UML Revision Task Force, 2010b) also provides the stereotype ≪Requirement≫ for classes. It can be used to express dependabilites between requirements and the relation to realization and tests (e.g., with the stereotypes ≪refine≫, ≪trace≫, ≪satisfy≫). We relate the requirements to domains of the environment to make their pupose explicit and provide support for requirements interaction analysis.

We are not aware of other tools supporting the work with problem frames on the semantic level, as does UML4PF.

4.5. Conclusions and Future Work

In this chapter, we have presented a support tool for requirements engineering using problem frames. This is supported by the UML profile for problem frames we have presented in this chapter. In this profile, we have introduced stereotypes corresponding to the basic notational elements used in problem frame. To automatically check the integrity between the different diagrams we have provided OCL constraints.

In summary, the advantages of our approach are:

- The requirements engineering approach of Jackson is enhanced by making use of UML concepts, such as multiplicities and aggregations, as well as adding diagram types.
- Integrity conditions using OCL are provided, e.g. to check if a problem diagram is consistent with a given problem frame.
- Several artifacts generated in an earlier development step can be re-used in later steps, e.g.
  - interfaces in the context diagram become external interfaces in the architecture
  - machine domains from the subproblems may become components in the architecture
- Existing tools are adapted, which reduces the effort of learning because UML4PF does not have an unknown "look-and-feel".

Problem frames are currently scarcely applied in industry. One reason is the number of figures to be created for a real project. Another reason is the missing support by tools suitable for industrial applications. Nevertheless, practitioners knowing problem frames apply them without showing the diagrams since even applying problem frames in the background substantially increases the quality of the requirements.

In the future, the profile should be used in industrial case studies. Additional integrity conditions may be defined and the existing integrity conditions may be refined. A challenging task is to improve performance and stability of the underlying tool (PapyrusUML). Alternatively, it is possible to apply the approach on another EMF-based UML tool. Another interesting topic is to investigate how to merge the separated requirements later in the development process.

Chapter 5

Expressing Dependability Requirements

For dependable systems, it is of utmost importance to thoroughly analyze, understand, and consolidate the requirements.

Dependability requirements should be described and analyzed. Problem frames can be extended to describe also dependability requirements and domain knowledge, as also shown in (Hatebur et al., 2007a).

This chapter is based on Hatebur and Heisel (2009b), where we have presented a foundation for requirements analysis of dependable systems, based on problem frames (Jackson, 2001). It is also based on Hatebur and Heisel (2010b). In this paper we show how the approach of (Hatebur & Heisel, 2009b) can be tool supported. To this end, we have defined a Unified Modeling Language (UML) profile (UML Revision Task Force, 2010c) that allows us to represent problem frames in UML. This UML profile is then augmented with stereotypes that allows one to expres of dependability requirements. The stereotypes are complemented by constraints expressed in the Object Constraint Language (OCL) (UML Revision Task Force, 2010a) that can be checked by existing UML tools. These constraints express important integrity conditions, for example, that security requirements must explicitly address a potential attacker. By checking the different OCL constraints, we can substantially aid system and software engineers in analyzing dependability requirements.

Within this chapter, the profile and the OCL constraints, defined in Chapter 4 are extended to express dependability requirements. We define a set of patterns that can be used to describe and analyze dependability requirements. These patterns are represented by three parts:

- A text where problem-specific aspects have to be selected or references to domains have to be inserted. This text can be used to discuss the model with persons without a technical background.
- UML elements and stereotypes that can be used to extend a problem diagram with non-functional requirements. With these elements, a seamless integration into the requirements engineering with problem diagrams is possible. The meta-model defining the stereotypes describes the important aspects for defining dependability requirements.
- A predicate that can be used to express dependencies between dependability requirements as described in the next chapter.

In earlier work (Hatebur et al., 2008b), we also defined a meta-model for problem frames. This meta-model is also defined using UML classes, but is not a UML profile, i.e., it does not extend UML meta-classes in order to define stereotypes. It focuses on problem frames and does not consider problem diagrams, dependability and later phases. In contrast, the profile described in this chapter defines stereotypes extends and is the basis for a tool support for the whole problem frames approach.

Section 5.1 contains our profile extension to describe dependability, and it also describes the OCL constraints for applying the elements introduced to describe dependability. In Section 5.4, we demonstrate how to to express already published security problem frames using our profile extension. Section 5.2 describes the process to work with our UML profile for requirements engineering for dependable systems. The case study in Section 5.3 applies the patterns and stereotypes of the profile extension to the cooperative adaptive cruise control system. Section 5.5 discusses related work, and the chapter closes with a summary and perspectives in Section 5.6.

5.1. Profile Extension to Describe Dependability

We developed a set of patterns for expressing and analyzing dependability features (requirements and domain knowledge). Our patterns consist of UML classes with stereotypes and a set of rules describing possible relations to other model elements. The stereotypes contain specific properties of the dependability feature (e.g., probabilities), a unique identifier, and a textual description that can be derived from the properties and the relations to other model elements. The patterns can be directly translated into logical predicates. These predicates are helpful to analyze conflicting requirements and the interaction of different dependability requirements, as well as to find missing dependability requirements (see Chapter 6).

An important advantage of our patterns is that they allow dependability requirements to be expressed without anticipating solutions. For example, we may require data to be kept confidential during transmission without being obliged to mention encryption, which is a means to achieve confidentiality. The benefit of considering dependability requirements without reference to potential solutions is the clear separation of problems from their solutions, which leads to a better understanding of the problems and enhances the re-usability of the problem descriptions, since they are completely independent of solution technologies.

Dependability features and functional features can be described independently. This approach limits the number of patterns, and allows one to apply these patterns to a wide range of problems. For example, the functional requirements for data transmission or automated control can be expressed using a problem diagram. Dependability requirements for confidentiality, integrity, availability and reliability can be added to that description of the functional requirement. The independence of these dependability requirements from functional requirements also allows one to express cross-cutting requirement with these pattern. To support this aspect, the references to the functional requirements, the constrained domains, and other attributes are modeled as sets.

```
1 Class.allInstances()->select(
2 (getAppliedStereotypes().name->includes('Dependability') or
3 getAppliedStereotypes().general.name->includes('Dependability') or
4 getAppliedStereotypes().general.general.name->includes('Dependability') )
5 and getAppliedStereotypes().name->includes('Requirement'))
6 ->forAll(clientDependency->select(d |
7     d.oclAsType(Dependency).getAppliedStereotypes().name -> includes('complements'))
8     .oclAsType(Dependency).target.getAppliedStereotypes().name ->
          includes('Requirement')->count(true)>=1 )
```

Listing 5.1: *Each Dependability Requirement Complements another Requirement*

A dependability requirement always complements (stereotype ≪complements≫) at least one functional requirement. This can be validated with the OCL expression in Listing 5.1. In this OCL expression, all classes with a stereotype indicating a dependability statement or a subtype (e.g., ≪Confidentiality≫ or ≪Availability_rnd≫) and additionally the stereotype ≪Requirement≫ are selected in Lines 1-5. In all of these requirement classes, it is checked that their dependencies (Line 6) with the stereotype ≪complements≫ (Line 7) point to at least one class with the stereotype ≪Requirement≫ (Line 8).

Our patterns help to structure and classify dependability requirements. For example, requirements considering integrity can be easily distinguished from the availability requirements. It is also possible to trace all dependability requirements that refer to a given domain.

The patterns for integrity, reliability, and availability considering random faults are expressed using probabilities, while for the security requirements no probabilities are defined. We are aware of the fact that no security mechanism provides a 100 % protection and that an attacker can break the mechanism to gain data with a certain probability (Santen, 2006). But in contrast to the random faults considered for the other requirements, no probability distribution can be assumed, because, e.g., new technologies may dramatically increase the probability that an attacker is successful. For this reason we suggest to describe a possible attacker and ensure that this attacker is not able to be successful in a reasonable amount of time.

In the following, we present our dependability profile extension and dependability analysis patterns.

5.1.1. Confidentiality

Confidentiality is the absence of unauthorized disclosure of information (Pfitzmann & Hansen, 2006). A typical confidentiality statement is to

> Preserve confidentiality of domain constrained in the functional statement representing the asset for stakeholders and prevent disclosure by attackers.

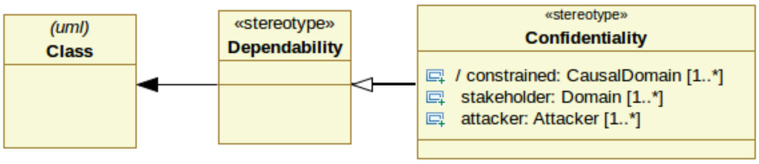

(uml)
Class
«stereotype»
Dependability
«stereotype»
Confidentiality
/ constrained: CausalDomain [1..*]
stakeholder: Domain [1..*]
attacker: Attacker [1..*]

Figure 5.1.: *UML Dependability Problem Frames Profile - Confidentiality*

A statement about confidentiality is modeled as a class with the stereotype ≪Confidentiality≫ in our profile. This stereotype is a specialization of the stereotype ≪Dependability≫, as shown in Fig. 5.1.

Three aspects have to be specified for a confidentiality requirement:

1. The constrained domain is a causal domain representing a stored, shown or transmitted asset. This domains has to be constrained using a dependency with the stereotype ≪constrains≫. Even if assets usually a lexical domains, we may model the asset as a CausalDomain, because in some cases the storage device, the display or the connection and not the asset itself is modeled.

   The same domain as in the complemented functional requirement is constrained because additional functionalities to achieve confidentiality are about the same domain. For example, if a display is constrained by the functional requirement, confidentiality is achieved by not showing the asset; if a storage or a connection are constrained by the functional requirement, confidentiality is achieved by not storing or sending the asset in readable form (e.g., by encryption). This condition can be expressed and checked with the OCL condition in Listing 5.2.

   The attribute constrained is modeled as a derived attribute and it is derived from the dependencies with the stereotype ≪constrains≫. It can be set with an operation, defined by the following precondition and postcondition (Listings 5.2 and 5.3).

In the precondition, all classes in the model with the stereotypes ≪Confidentiality≫ and also ≪Statement≫ or ≪Requirement≫ are selected, and for all confidentiality statements the following condition is checked (lines 1-3). The dependencies starting at this class (clientDependency) with the stereotype ≪constrains≫ (line 4) are considered. The targets of the 'constrains'-dependencies are checked to have the stereotype ≪CausalDomain≫ or a subtype, and the boolean results are collected (lines 5-8). It is checked by counting the positive results if there is at least one causal domain (or a subtype) constrained (line 9).

```
1  Class.allInstances()->select(getAppliedStereotypes().name ->
       includes('Confidentiality'))
2  ->select(getAppliedStereotypes().name -> includes('Statement') or
3    getAppliedStereotypes().name -> includes('Requirement'))->forAll(
4    clientDependency -> select(r |
         r.oclAsType(Dependency).getAppliedStereotypes().name ->
         includes('constrains'))
5  .oclAsType(Dependency).target.getAppliedStereotypes() -> collect(
6    name->includes('CausalDomain') or
7    general.name->includes('CausalDomain') or
8    general.general.name->includes('CausalDomain')
9  )->count(true)>=1)
```

Listing 5.2: *Precondition for operation setting the derived attribute 'constrained'*

After the operation has been performed, the postcondition in Listing 5.3 has to hold. For all classes in the model with the stereotypes ≪Confidentiality≫ and also ≪Statement≫ or ≪Requirement≫ (lines 1-3), the attribute asset (lines 4-6) has the same elements (line 7) as the target of the dependencies starting at this class (clientDependency) with the stereotype ≪constrains≫ (line 8-9).

```
1  Class.allInstances()->select(getAppliedStereotypes().name ->
       includes('Confidentiality'))
2  ->select(getAppliedStereotypes().name -> includes('Statement') or
3  getAppliedStereotypes().name -> includes('Requirement'))->forAll(c|
4    c.oclAsType(Class).getValue(c.oclAsType(Class) .getAppliedStereotypes() ->
         select(s |
5              s.oclAsType(Stereotype).name -> includes('Confidentiality') )
6              ->asSequence()->first(),'constrained')
                   .oclAsType(ProblemFrames::CausalDomain).base_Class->asSet()
7    =
8    clientDependency -> select(r |
         r.oclAsType(Dependency).getAppliedStereotypes().name ->
         includes('constrains'))
9    .oclAsType(Dependency).target.oclAsType(Class)->asSet()
10 )
```

Listing 5.3: *Postcondition for operation setting the derived attribute 'constrained'*

2. For confidentiality, we need to consider an attacker. This attacker must be described in detail. We suggest to describe at least the objective of the attackers, their skills, equipment, knowledge, and the time the attackers have to prepare and to perform the attack (see Fig. 5.2). A similar kind of description is suggested in the Common Methodology for Information Technology Security Evaluation (CEM) (International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC), 2009b). As shown in Fig. 5.2, the stereotype ≪Attacker≫ is a specialized ≪BiddableDomain≫. The reference to an Attacker is necessary, because we can only ensure confidentiality with respect to an Attacker with given properties. The reference from the stereotype ≪Confidentiality≫ to the attacker is given by an attribute of the stereotype. The multiplicity of [1..*] ensures that at least one attacker is referenced. An OCL expression checking this aspect is given in Appendix C, Listing C.59.

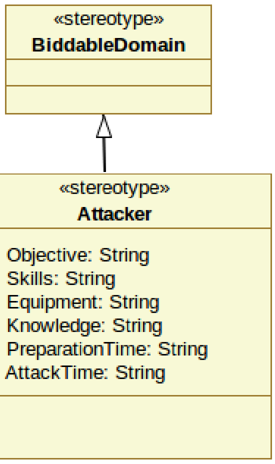


Figure 5.2.: *Attacker in UML Dependability Problem Frames Profile*

3. For confidentiality, we also need to consider the data's stakeholder (see Appendix C, Listing C.60). The stakeholder is referred to, because we want to allow the access only to stakeholders with legitimate interest (Gürses et al., 2005). The instances of stakeholder and attacker must be disjoint.

It is possible to generate the text of the confidentiality statement from other model information: In the typical confidentiality statement the constrained domain (e.g., containing the stored asset) can be obtained from the names of the domains constrained by this statement, the Attacker can be instantiated with the value of the attribute attacker in the stereotype ≪Confidentiality≫, and the Stakeholder can be instantiated with the value of the attribute stakeholder in the stereotype ≪Confidentiality≫. Additionally, the names of complemented functional requirements can be added to the statement text if they exist. They only exist if the statement is a requirement.

The security requirement pattern can be expressed by the following confidentiality predicate:

$$conf_{att} : \mathbb{P}\ ClassWithCausalDomainStereotype \times \mathbb{P}\ ClassWithDomainStereotype \times \\ \mathbb{P}\ ClassWithAttackerStereotype \rightarrow Bool$$

The suffix "att" indicates that this predicate describes a requirement considering a certain *att*acker. The symbol 'ℙ' denotes a powerset. The type *ClassWithCausalDomainStereotype* represents a class with the stereotype CausalDomain or a stereotype derived from CausalDomain (e.g., LexicalDomain) applied. The other types (e.g., *ClassWithDomainStereotype*) are defined in the same way. The predicate $cont_{att}(cd, s, att)$ means that the asset of the constrained domains in the set cd have to be kept confidential for the stakeholders in the set s against the attackers in the set att.

For example, a constrained domain may be the PIN of a bank account, a special stakeholder may be the bank account owner, and a special attacker may be the class of all persons with no permission, who want to withdraw money and have access to all external interfaces of the machine. The following pattern can be used to define confidentiality requirements:

$$\forall\ constrained : ClassWithCausalDomainStereotype; \\ stakeholder : ClassWithDomainStereotype; \\ attacker : ClassWithAttackerStereotype \bullet \\ conf_{att}(\{constrained\}, \{stakeholder\}, \{attacker\})$$

If two or more assets are constrained, the following pattern can be used. The purpose of this rule is to allow to minimize the number of predicates to describe dependability. When a set of a predicate parameter contains several elements, this is equivalent to several AND-linked predicates of this type with subsets in the corresponding parameter:

$$\begin{aligned}
&\forall\, constrained_1 : ClassWithCausalDomainStereotype;\\
&\quad constrained_2 : AnotherClassWithCausalDomainStereotype;\\
&\quad stakeholder : ClassWithBiddableDomainStereotype;\\
&\quad attacker : ClassWithAttackerStereotype \bullet\\
&conf_{att}(\{constrained_1, constrained_2\}, \{stakeholder\}, \{attacker\})\\
&\Leftrightarrow conf_{att}(\{constrained_1\}, \{stakeholder\}, \{attacker\})\\
&\wedge conf_{att}(\{constrained_2\}, \{stakeholder\}, \{attacker\})
\end{aligned}$$

On the other hand, this rule allows to use the pattern system in the next chapter with predicates where the set in the parameter has more than one element.

A confidentiality requirement is often used together with functional requirements for data transmission and data storage. As an example, in Fig. 5.3, the confidentiality requirement is applied to the simple workpieces problem frame.

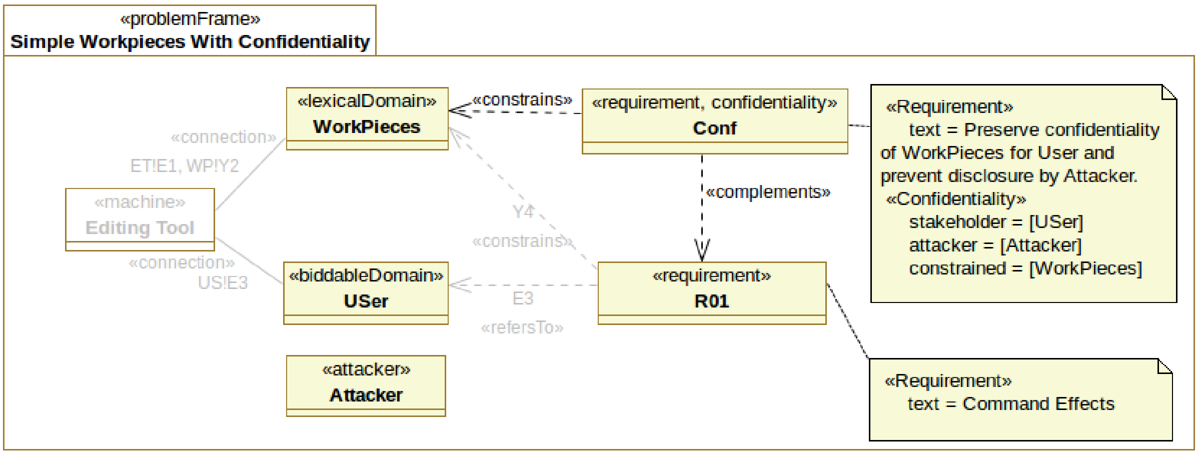


Figure 5.3.: *Simple Workpieces Problem Frame with Confidentiality Requirement*

This confidentiality requirement can be described with the predicate as follows:

$$\begin{aligned}
&\forall\, asset : Workpieces, stakeholder : USer, attacker : Attacker \bullet\\
&\quad conf_{att}(\{asset\}, \{stakeholder\}, \{attacker\})
\end{aligned}$$

5.1.2. Integrity

Integrity is the absence of improper system, data, or a service alterations (Pfitzmann & Hansen, 2006). Typical integrity statements considering random faults are:

> With a probability of P_i, one of the following things should happen: service (as described in the functional statement) with influence on / of the domain constrained in the functional statement must be either correct, or domain influenced in case of a violation (influencedIfViolation) shall perform a specific action.

Typical security integrity statements are:

> For stakeholder, the influence (as described in the functional statement) on / content of the domain constrained in the functional statement must be either correct, or in case of any modifications by Attackers the domain influenced in case of a violation (influencedIfViolation) shall perform a specific action.

In contrast to the integrity statement considering random faults, the security integrity requirement can refer to the content of a domain, because security engineering usually focuses on data. For security, the domain constrained in the functional requirement is usually a display or some plain data. The specific action for both security and safety could be, e.g.:

- Write a log entry.
- Switch off an actuator.
- Do **not** influence the domain constrained in the functional statement (e.g., deny modification).
- perform the same action as defined in the functional statement on domain constrained in the functional statement. In this case integrity coincidence with reliability.
- inform stakeholder. In this case the stakeholder is a biddable domain and cannot be directly constrained. Therefore, the stakeholder must be informed by some technical means that can be constrained, e.g. a display. The assumption that the stakeholder sees the display (being necessary to derive a specification from the requirements) must be checked later for validity.

Integrity statements are modeled as classes with the stereotype ≪Integrity_att≫ or ≪Integrity_rnd≫. In our profile, this stereotype is an indirect specialization of the stereotype ≪Dependability≫, as shown in Fig. 5.4.

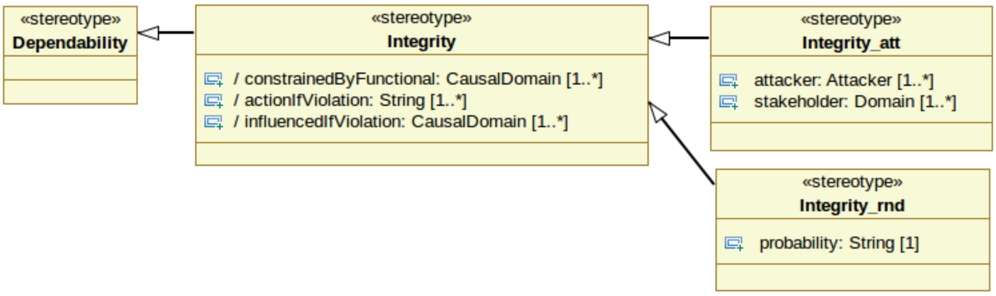

«stereotype»
Dependability
«stereotype»
Integrity
/ constrainedByFunctional: CausalDomain [1..*]
/ actionIfViolation: String [1..*]
/ influencedIfViolation: CausalDomain [1..*]
«stereotype»
Integrity_att
attacker: Attacker [1..*]
stakeholder: Domain [1..*]
«stereotype»
Integrity_rnd
probability: String [1]

Figure 5.4.: *UML Dependability Problem Frames Profile - Integrity*

The stereotype ≪Integrity≫ should not be used directly in a model. Instead the derived stereotypes ≪Integrity_att≫ or ≪Integrity_rnd≫ should be used. The OCL expression in Appendix C, Listing C.62 checks that no class with the stereotype ≪Integrity≫ exists.

The domains mentioned in the specific action must be constrained by the integrity statement. The attribute actionIfViolation of the stereotype ≪Integrity≫ contains the textual descriptions of the specific actions as a set of strings. From the dependencies with the stereotype ≪constrains≫ the attributes influencedIfViolation and actionIfViolation are derived. The precondition of the operation is given in in Appendix C, Listing C.65. The postcondition is given in in Appendix C, Listing C.66.

In contrast to the other dependability requirements, an integrity requirement needs to refer to the domain constrained by the complemented functional requirement: it does not necessarily constrain this requirement. This can be validated with the constraint in Appendix C, Listing C.61.

The attribute constrainedByFunctional of the stereotype ≪Integrity≫ is modeled as a derived attribute. It is derived from the dependencies with the stereotype ≪constrains≫ of the complemented functional requirement. It can be set with an operation, defined by the precondition given in Appendix C, Listing C.63 and the postcondition is given in in Appendix C, Listing C.64.

For requirements considering random faults, the stereotype ≪Integrity_rnd≫ can be used. For each integrity requirement considering random faults, exactly one probability must be specified (see Appendix C, Listing C.67). The probability is a constant, determined by risk analysis. The standard ISO/IEC 61508 (International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC), 2000) provides a range of failure rates for each defined safety integrity level (SIL). The probability P_i could be, e.g., for SIL 3 systems operating on demand $1 - 10^{-3}$ to $1 - 10^{-4}$. For continuous (or high-demand) systems no (dangerous) integrity fault with a probability of $1 - 10^{-7}$ to $1 - 10^{-8}$ per year is necessary.

For requirements considering an attacker, the attribute attacker needs to be defined. The attacker must be described in the same way as for confidentiality in Section 5.1.1 (see Appendix C, Listing C.68).

It is possible to generate the text of the integrity statement from other model information. The domain constrained by the functional statement can be obtained from the causal domain referred to in the integrity statement. The specific action can be obtained from the value of the stereotype attribute actionIfViolation and the name of the domain constrained by the statement. The probability P_i can be obtained from the value of the stereotype attribute probability. The attacker can be obtained from the domain in the integrity stereotype attribute attacker. It s a class with the stereotype ≪attacker≫ or a subtype of ≪attacker≫.

The statement considering random faults can be expressed by the following integrity predicate:

$$\begin{aligned}&int_{rnd} : \mathbb{P}\, ClassWithCausalDomainStereotype \times \mathbb{P}\, String \times \\ &\quad \mathbb{P}\, ClassWithCausalDomainStereotype \times Probability \rightarrow Bool.\end{aligned}$$

The suffix "rnd" indicates that this predicate describes a requirement considering random faults. The predicate $int_{rnd}(cd, aiv, iiv, P_i)$ means that with a probability of P_i, one of the following things should happen: service (as described in the functional statement) of (or with influence on) the constrained domains in the set cd or the domains influenced in case of a violation in the set iiv shall perform the specific actions in the set aiv.

The security statement can be expressed by the following integrity predicate:

$$\begin{aligned}&int_{att} : \mathbb{P}\, ClassWithCausalDomainStereotype \times \mathbb{P}\, String \times \mathbb{P}\, ClassWithCausalDomainStereotype \times \\ &\quad \mathbb{P}\, ClassWithDomainStereotype \times \mathbb{P}\, ClassWithAttackerStereotype \rightarrow Bool\end{aligned}$$

The predicate $int_{att}(cd, aiv, iiv, s, a)$ means that for the stakeholders in the set s influence (as described in the functional statement) on or the content of the constrained domains in the set cd must be either correct or in case of any modifications by an attacker in the set a the domains influenced in case of a violation in the set iiv shall perform the specific actions in the set aiv.

The following pattern can be used to define the integrity statements for a given probability P_i and a given $actionIfViolation$:

$$\begin{aligned}&\forall\, constrainedByFunctional : ClassWithCausalDomainStereotype; \\ &\quad influencedIfViolation : ClassWithCausalDomainStereotype \bullet \\ &int_{rnd}(\{constrainedByFunctional\}, \{actionIfViolation\}, \{influencedIfViolation\}, P_i)\end{aligned}$$

The following pattern can be used to define integrity requirements considering an attacker for a given *actionIfViolation*:

$$\begin{aligned}
&\forall\, constrainedByFunctional : ClassWithCausalDomainStereotype;\\
&\quad influencedIfViolation : ClassWithCausalDomainStereotype;\\
&\quad stakeholder : ClassWithDomainStereotype;\\
&\quad attacker : ClassWithAttackerStereotype \bullet\\
&int_{att}(\{constrainedByFunctional\}, \{actionIfViolation\}, \{influencedIfViolation\},\\
&\quad \{stakeholder\}, \{attacker\})
\end{aligned}$$

For integrity considering an attacker with the specific action to deny any modification, the predicate int_{att_d} can be used.

$$\begin{aligned}
&\forall\, constrainedByFunctional : ClassWithCausalDomainStereotype;\\
&\quad stakeholder : ClassWithDomainStereotype \bullet\\
&\quad attacker : ClassWithAttackerStereotype \bullet\\
&int_{att\_d}(\{constrainedByFunctional\}, \{stakeholder\}, \{attacker\})\\
&\Leftrightarrow int_{att}(\{constrainedByFunctional\}, \{\text{'deny modification'}\}, \{constrainedByFunctional\},\\
&\quad \{stakeholder\}, \{attacker\})
\end{aligned}$$

As an example for integrity requirement considering random faults, in Fig. 5.5, the integrity requirement is applied to the commanded behaviour problem frame.

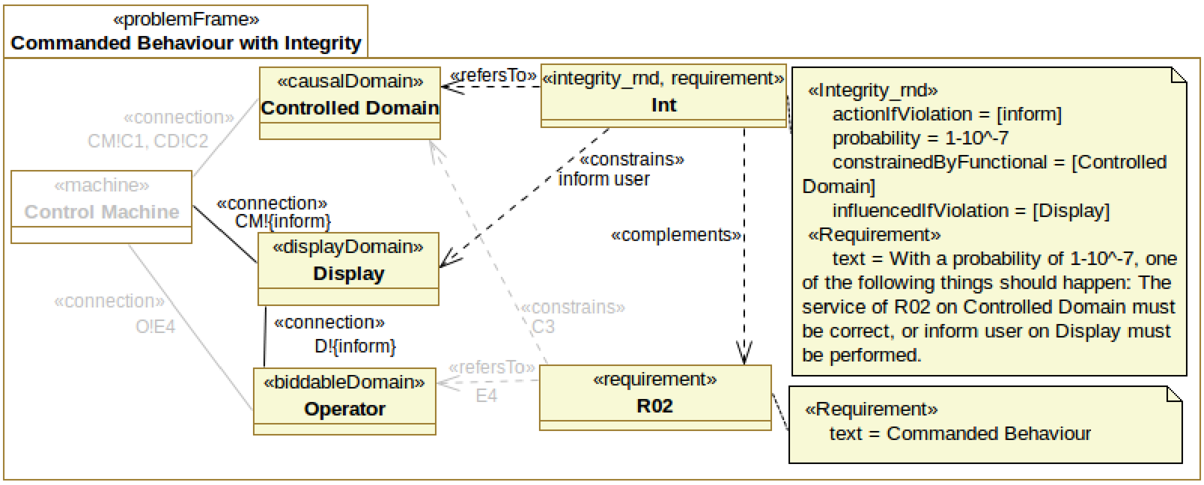

«problemFrame»
Commanded Behaviour with Integrity
«causalDomain»
Controlled Domain
«refersTo»
«integrity_rnd, requirement»
Int
«Integrity_rnd»
actionIfViolation = [inform]
probability = 1-10^-7
constrainedByFunctional = [Controlled Domain]
influencedIfViolation = [Display]
«Requirement»
text = With a probability of 1-10^-7, one of the following things should happen: The service of R02 on Controlled Domain must be correct, or inform user on Display must be performed.
«connection»
CM!C1, CD!C2
«machine»
Control Machine
«connection»
CM!{inform}
«constrains»
inform user
«complements»
«displayDomain»
Display
«connection»
O!E4
«connection»
D!{inform}
«constrains»
C3
«refersTo»
E4
«biddableDomain»
Operator
«requirement»
R02
«Requirement»
text = Commanded Behaviour

Figure 5.5.: *Commanded Behaviour Problem Frame with Integrity Requirement*

This integrity requirement can be described with the predicate as follows:

$$\begin{aligned}
&\forall\, constrainedByFunctional : ControlledDomain;\\
&\quad influencedIfViolation : Display \bullet\\
&int_{rnd}(\{constrainedByFunctional\}, \{\text{'inform user'}\}, \{influencedIfViolation\}, P_i)
\end{aligned}$$

An integrity requirement considering an attacker is often used together with functional requirements for data transmission and data storage. As an example, in Fig. 5.6, the integrity requirement is applied to the simple workpieces problem frame.

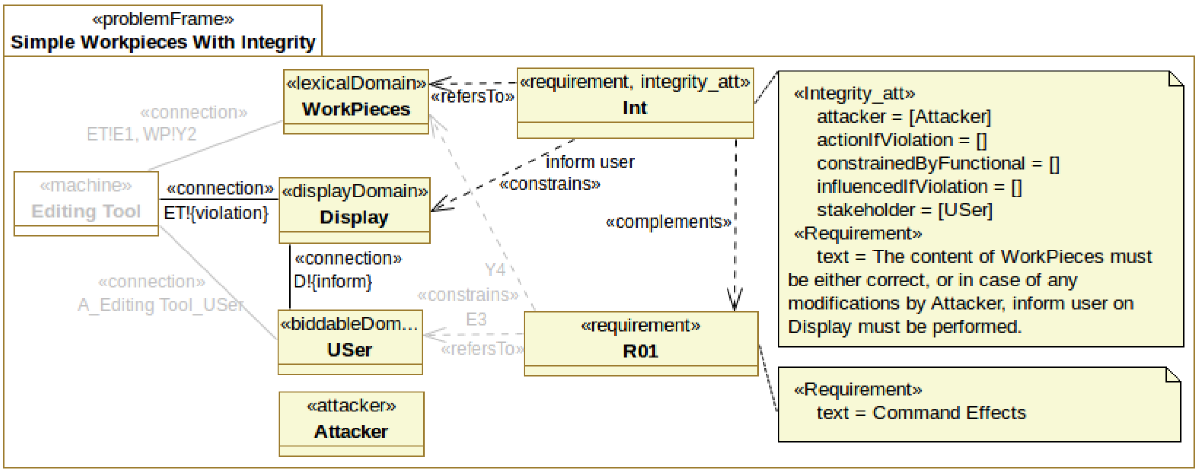


Figure 5.6.: *Simple Workpieces Problem Frame with Integrity Requirement*

This integrity requirement can be described with the predicate as follows:

$$\begin{aligned}
&\forall\, \mathit{constrainedByFunctional} : \mathit{ControlledDomain};\\
&\quad \mathit{influencedIfViolation} : \mathit{Display};\ \mathit{stakeholder} : \mathit{USer};\\
&\quad \mathit{attacker} : \mathit{Attacker} \bullet\\
&\mathit{int}_{\mathit{rnd}}(\{\mathit{constrainedByFunctional}\}, \{\text{'inform user'}\}, \{\mathit{influencedIfViolation}\},\\
&\quad \{\mathit{stakeholder}\}, \{\mathit{attacker}\})
\end{aligned}$$

5.1.3. Availability

Laprie (1995) defines that **Availability** is the readiness for service (up-time vs. down-time). The BS 4778 (British Standards Institution (BSI), 1998) defines availability as "the ability of an item (under combined aspects of its reliability, maintainability and maintenance support) to perform its required function at a stated instant of time or over a stated period of time". Since we want to have requirements that do not cover too many aspects and we do not consider maintenance, we use the definition of Laprie (1995). In this case, typical availability statements considering random faults are:

> The service (described in the functional statement) with influence on / of the domain constrained in the functional statement must be available (for users) with a probability of P_a.

When we talk about availability in the context of security, it is not possible to provide the service to everyone due to limited resources. Availability statements considering an attacker can be expressed as follows:

> The service (described in the functional statement) with influence on / of the domain constrained in the functional statement (constrained) must be available for users even in case of an attack by Attackers.

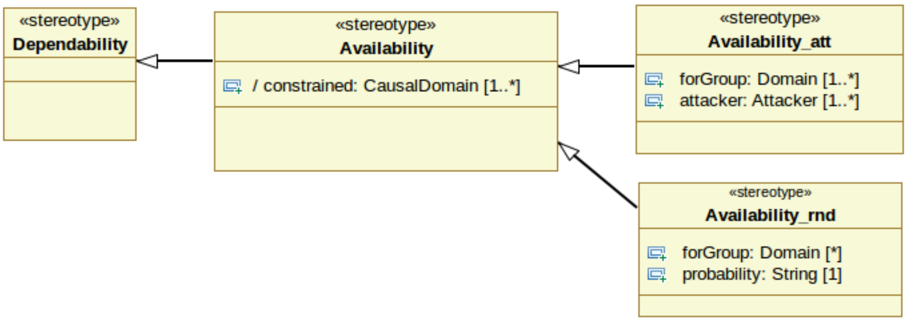


Figure 5.7.: *UML Dependability Problem Frames Profile - Availability*

Availability statements are modeled as classes with the stereotypes ≪Availability_att≫ or ≪Availability_rnd≫. In our profile, this stereotype is an indirect specialization of the stereotype ≪Dependability≫, shown in Fig. 5.7. The stereotype ≪Availability≫ should not be used directly in a model. Instead the derived stereotypes ≪Availability_att≫ or ≪Availability_rnd≫ should be used. The OCL expression in Appendix C, Listing C.69 checks that no class with the stereotype ≪Availability≫ exists.

Availability requirements also constrain the domains constrained by the complemented functional requirement. We only require this condition for requirements; for domain knowledge we do not want to force the developers to describe any functional aspect within the model. To validate that availability requirements constrains the domain constrained in the functional statement, we check for all classes with the stereotypes ≪Availability_rnd≫ or ≪Availability_add≫ and ≪Requirement≫ that the set of its constrained classes includes the set of all constrained classes of the complemented functional requirements (see Appendix C, Listing C.70).

The attribute constrained of the stereotype ≪Availability≫ is modeled as a derived attribute. It is derived from the dependencies with the stereotype ≪constrains≫. It can be set with an operation, defined by the precondition given in Appendix C, Listing C.71 and the postcondition is given in in Appendix C, Listing C.72.

For availability requirements considering random faults, we use the stereotype ≪Availability_rnd≫. In that case also the probability must be specified. This can be checked in the same way as for integrity, described in Section 5.1.2 (see Appendix C, Listing C.74). P_a is the probability that the service (i.e., the influence on the constrained domain) is accessible for defined users. A probaility P_a of $1 - 10^{-5}$ means that the service may be unavailable on average for 315 seconds in one year.

For availability requirements considering an attacker the stereotype attributes forGroup and attacker must be specified, as required by the OCL expression in Appendix C, Listing C.73.

It is possible to generate the text of the dependability statement from other model information. The domain constrained by the functional statement can be obtained from the stereotype attribute constrained. The user can be obtained from the stereotype attribute forGroup. It is not necessary, that we restrict the availability requirement considering random faults to a limited set of users. Therefore, the set forGroup can be empty in this case. If the set in forGroup is empty, the restriction for certain users can be omitted. The probability P_i can be obtained from the value of the stereotype attribute probability. The attacker can be obtained from the domain in the integrity stereotype attribute attacker. It is a class with the stereotype ≪attacker≫ or a subtype of ≪attacker≫. Additionally, the names of complemented functional requirement can be added.

The availability statement considering random faults can be expressed by the following availability predicate:

$$\begin{aligned}&avail_{rnd} : \mathbb{P}\, ClassWithCausalDomainStereotype \times\\ &\quad \mathbb{P}\, ClassWithDomainStereotype \times Probability \rightarrow Bool\end{aligned}$$

The predicate $avail_{rnd}(cd, fg, P_a)$ means that the service (described in the functional statement) of (or with influence on) the constrained domains in the set cd must be available for the users in the set fg with a probability of P_a.

The availability statement considering an attacker can be expressed by the following availability predicate:

$$\begin{aligned}&avail_{att} : \mathbb{P}\, \mathit{ClassWithCausalDomainStereotype} \times \mathbb{P}\, \mathit{ClassWithDomainStereotype} \times\\ &\quad \mathbb{P}\, \mathit{ClassWithAttackerStereotype} \rightarrow \mathit{Bool}\end{aligned}$$

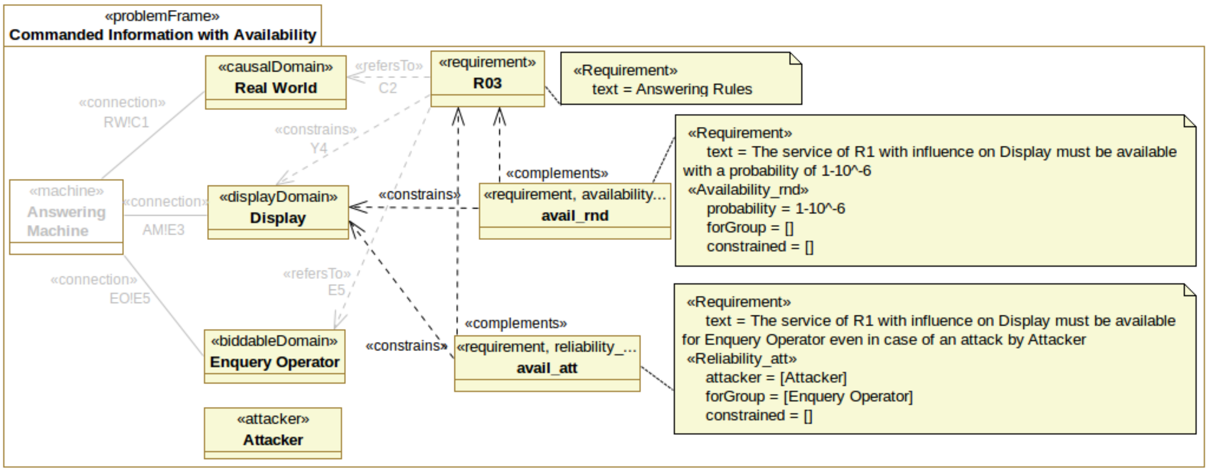
The predicate $avail_{att}(cd, fg, a)$ means that the service (described in the functional statement) of (or with influence on) the constrained domains in the set cd must be available for the users in the set fg even in case of an attack by the attackers in the set a.

The following patterns can be used to define the availability requirements for a given probability P_a:

$$\begin{aligned}&\forall\, \mathit{constrained} : \mathit{ClassWithCausalDomainStereotype};\\ &\quad \mathit{forGroup} : \mathit{ClassWithDomainStereotype} \bullet\\ &avail_{rnd}(\{\mathit{constrained}\}, \{\mathit{forGroup}\}, P_a)\\ &\forall\, \mathit{constrained} : \mathit{ClassWithCausalDomainStereotype} \bullet\\ &\quad avail_{rnd}(\mathit{constrained}, \emptyset, P_a)\\ &\forall\, \mathit{constrained} : \mathit{ClassWithCausalDomainStereotype};\\ &\quad \mathit{forGroup} : \mathit{ClassWithDomainStereotype};\\ &\quad \mathit{attacker} : \mathit{ClassWithAttackerStereotype} \bullet\\ &avail_{att}(\{\mathit{constrained}\}, \{\mathit{forGroup}\}, \{\mathit{attacker}\})\end{aligned}$$

Availability requirements are often used with functional requirements with a constrained causal domain. .

As an example, in Fig. 5.8, an availability requirements considering random faults and an availability requirements considering an attacker are applied to the commanded information problem frame.



«Requirement»
text = Answering Rules

«Requirement»
text = The service of R1 with influence on Display must be available with a probability of 1-10^-6
«Availability_rnd»
probability = 1-10^-6
forGroup = []
constrained = []

«Requirement»
text = The service of R1 with influence on Display must be available for Enquery Operator even in case of an attack by Attacker
«Reliability_att»
attacker = [Attacker]
forGroup = [Enquery Operator]
constrained = []

Figure 5.8.: *Commanded Information Problem Frame with Availability Requirements*

These availability requirements can be described with the predicates as follows:

$$\begin{aligned}&\forall\, \mathit{constrained} : \mathit{Display};\ \mathit{forGroup} : \mathit{User};\ \mathit{attacker} : \mathit{Attacker} \bullet\\ &\quad avail_{rnd}(\mathit{constrained}, \emptyset, 1 - 10^{-5}) \wedge\\ &\quad avail_{att}(\mathit{constrained}, \mathit{forGroup}, \mathit{attacker})\end{aligned}$$

5.1.4. Reliability

Reliability is a measure of continuous service accomplishment (Laprie, 1995). The ISO 8402 (International Organization for Standardization (ISO), 1994) defines reliability as "the ability of an item to perform a required function, under given environmental and operational conditions for a stated period of time." Reliability is defined in a similar way as availability (see Section 5.1.3). The same failure rates as for integrity (see Section 5.1.2) can be used. A typical reliability requirement considering random faults is that

> The service (described in the functional requirement) with influence on / of the domain constrained by the functional statement must be reliable (for users) with a probability of P_r.

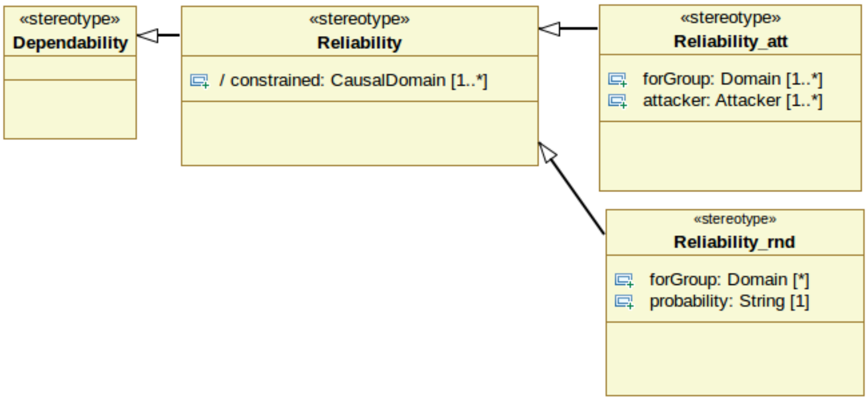


Figure 5.9.: *UML Dependability Problem Frames Profile - Reliability*

Reliability statements are modeled as classes with the stereotypes ≪Reliability_att≫ and ≪Reliability_rnd≫ being specializations of the ≪Reliability≫ that should not be used (see Appendix C, Listing C.75). The constraints are similar to the constraints for availability (see Appendix C, Listings C.76, C.77, C.78, and C.79). The probability is a constant determined by risk analysis. The same failure rates as for integrity (see Section 5.1.2) can be used (see Appendix C, Listing C.80).

Reliability considering random fault can be expressed with the following predicate:

$$rel_{rnd} : \mathbb{P}\, ClassWithCausalDomainStereotype \times \\ \mathbb{P}\, ClassWithDomainStereotype \times Probability \to Bool$$

The predicate $rel_{rnd}(cd, fg, P_r)$ means that the service (described in the functional statement) of (or with influence on) the constrained domains in the set cd must be reliable for the users in the set fg with a probability of P_r.

Reliability considering certain attackers can be expressed with the following predicate:

$$rel_{att} : \mathbb{P}\, ClassWithCausalDomainStereotype \times \\ \mathbb{P}\, ClassWithDomainStereotype \times \mathbb{P}\, ClassWithAttackerStereotype \to Bool$$

The predicate $rel_{att}(cd, fg, a)$ means that the service (described in the functional statement) of (or with influence on) the constrained domains in the set cd must be reliable for the users in the set fg even in case of an attack by the attackers in the set a.

It is possible to generate the statement text in the same way as for availability.

Reliability consists of two parts, the availability of the functionality and the integrity of the functionality. The availability only defines the probability that the service is available, and the integrity defines the probability that the service is correct or a fault is correctly handled. To have a reliable system, the service must be correct and available. Hence, with a probability of P_r:

$$\begin{array}{l} \forall\, d : \mathbb{P}\, \mathit{ClassWithCausalDomainStereotype} \bullet \\ \quad \mathit{rel}_{\mathit{rnd}}(\{d\}, \emptyset, P_r) \Leftrightarrow \\ \quad (\mathit{avail}_{\mathit{rnd}}(\{d\}, \emptyset, P_r) \wedge \mathit{int}_{\mathit{rnd}}(\{d\}, \{\text{"do not influence"}\}, \{d\}, P_r) \end{array}$$

$\mathit{int}_{\mathit{rnd}}(d, \text{"do not influence"}, d, P_r)$ requires that the service with influence on ConstrainedDomain must be either correct or it must be switched off (with a probability of P_r). But $\mathit{avail}_{\mathit{rnd}}(d, \emptyset, P_r)$ also requires that it should not be switched of or completely fail with the probability P_r.

It is also possible, that the integrity requirement states that

- with a probability of P_i, one of the following things should happen: the ConstrainedDomain must work correctly, or ConstrainedDomain must perform the same action as defined in the complemented statement.

In this case, the integrity requirement is equivalent to the reliability requirement:

$$\begin{array}{l} \forall\, d : \mathit{ClassWithCausalDomainStereotype} \bullet \\ \quad \mathit{rel}_{\mathit{rnd}}(\{d\}, \emptyset, P_r) \Leftrightarrow \\ \quad \mathit{int}_{\mathit{rnd}}(\{d\}, \{\text{"perform same action as defined in the complemented statement"}\}, \{d\}, P_r) \end{array}$$

The following pattern can be used to define the reliability requirements:

$$\begin{array}{l} \forall\, \mathit{constrained} : \mathit{ClassWithCausalDomainStereotype}; \\ \quad \mathit{forGroup} : \mathit{ClassWithDomainStereotype} \bullet \\ \quad \mathit{rel}_{\mathit{rnd}}(\{\mathit{constrained}\}, \{\mathit{forGroup}\}, P_a) \\ \forall\, \mathit{constrained} : \mathit{ClassWithCausalDomainStereotype} \bullet \\ \quad \mathit{rel}_{\mathit{rnd}}(\{\mathit{constrained}\}, \emptyset, P_a) \\ \forall\, \mathit{constrained} : \mathit{ClassWithCausalDomainStereotype}; \\ \quad \mathit{forGroup} : \mathit{ClassWithDomainStereotype}; \\ \quad \mathit{attacker} : \mathit{ClassWithAttackerStereotype} \bullet \\ \quad \mathit{rel}_{\mathit{rnd}}(\{\mathit{constrained}\}, \{\mathit{forGroup}\}, \{\mathit{attacker}\}) \end{array}$$

Reliability requirements are often used with functional requirements with a constrained causal domain. It can be annotated in the same way as availability requirements (see Fig. 5.8 for a commanded information problem frame with availability requirements). The reliability requirements can be described with the predicates as follows:

$$\begin{array}{l} \forall\, \mathit{constrained} : \mathit{Display};\ \mathit{forGroup} : \mathit{USer};\ \mathit{attacker} : \mathit{Attacker} \bullet \\ \quad \mathit{rel}_{\mathit{rnd}}(\{\mathit{constrained}\}, \emptyset, 1 - 10^{-5}) \wedge \\ \quad \mathit{rel}_{\mathit{att}}(\{\mathit{constrained}\}, \{\mathit{forGroup}\}, \{\mathit{attacker}\}) \end{array}$$

5.1.5. Authenticity

Authenticity describes a property that ensures that a communication partner is the partner it claims to be (*IT-Grundschutz-Katalog*, 2011). A typical authenticity statement is to

> Check authenticity of known domains to distinguish from unknown domains and modify influenced domain accordingly. Permit access for known domains and deny access for unknown domains on influenced domain.

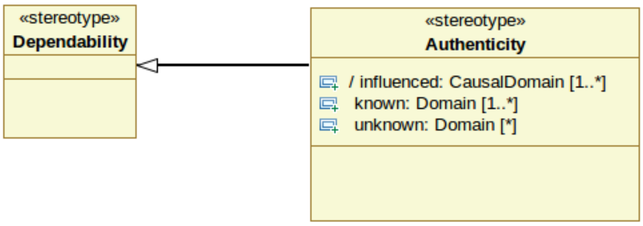


Figure 5.10.: *UML Dependability Problem Frames Profile - Authenticity*

Authenticity is only relevant for security. A statement about authenticity is modeled as a class with the stereotype ≪Authenticity≫ in our profile. This stereotype is a specialization of the stereotype ≪Dependability≫, as shown in Fig. 5.10. Three aspects have to be specified for an authenticity requirement:

1. The influenced domain is the domain that is modified according to an authentication process. It can be a display showing the authentication status, an internal representation of the authentication status that used, e.g., for access control, or a lexical domain that must not be changed by unknown users. The stereotype attribute influenced is modeled as a derived attribute. It is derived from the dependencies with the stereotype ≪constrains≫ (see Appendix C, Listing C.82 for the postcondition of the operation to derive the stereotype attribute influenced). Dependencies with the stereotype ≪constrains≫ shall only point to causal domains (see Appendix C, Listing C.81 for the precondition of the operation to derive the stereotype attribute influenced).

2. Known domains describe users or technical systems that are known by the machine and that prove their authenticity. The stereotype attribute known is modeled as a derived attribute. It is derived from the dependencies with the stereotype ≪refersTo≫ with the name known (see Appendix C, Listing C.83 for the postcondition of the operation to derive the stereotype attribute known). The dependencies with the stereotype ≪refersTo≫ shall only point to domains (see Appendix C, Listing C.24 for the precondition of the operation to derive the stereotype attribute known). At least one domain must be known as stated in the multiplicity (see also Appendix C, Listing C.84).

3. Unknown domains describe users or technical systems that are not known by the machine or cannot prove their authenticity. The stereotype attribute known is derived from the dependencies with the stereotype ≪refersTo≫ with the name unknown (see Appendix C, Listing C.85, postcondition of the operation to derive the stereotype attribute unknown). The dependencies with the stereotype ≪refersTo≫ shall only point to domains (see Appendix C, Listing C.24, precondition of the operation to derive the stereotype attribute unknown).

It is possible to generate the statement text from the stereotype attributes known, unknown, and influenced (constrained causal domains).

The authenticity requirement pattern can be expressed by the following dependability predicate:

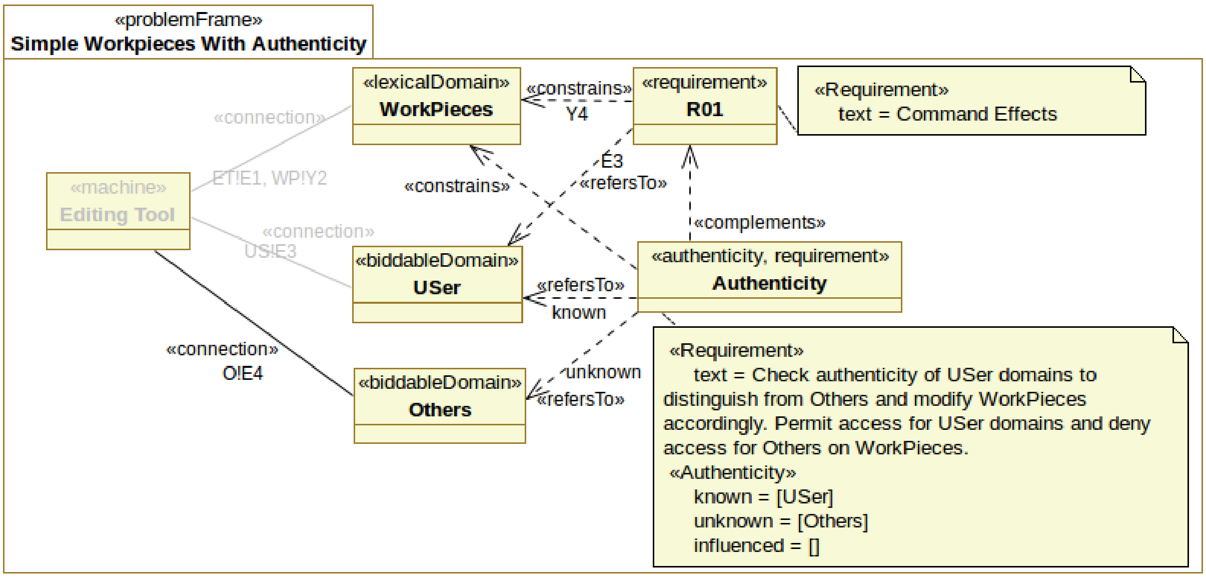
$$\begin{aligned} &auth_{att} : \mathbb{P}\ ClassWithCausalDomainStereotype \times \\ &\quad \mathbb{P}\ ClassWithDomainStereotype \times \mathbb{P}\ ClassWithDomainStereotype \to Bool \end{aligned}$$

The predicate $auth_{att}(id, k, u)$ means that the authenticity of the known domains in the set k has to be distinguished from the unknown domains in the set u and the influenced domains in the set id

The following pattern can be used to define the authentication requirement:

$$\begin{aligned} &\forall\ influenced : ClassWithCausalDomainStereotype; \\ &\quad known : ClassWithDomainStereotype; \\ &\quad unknown : ClassWithDomainStereotype \bullet \\ &auth_{att}(\{influenced\}, \{known\}, \{unknown\}) \end{aligned}$$

An authenticity requirement is often used together with functional requirements with lexical domains. As an example, in Fig. 5.11, the authenticity requirement is applied to the simple workpieces problem frame.



«Requirement»
text = Command Effects

«Requirement»
text = Check authenticity of USer domains to distinguish from Others and modify WorkPieces accordingly. Permit access for USer domains and deny access for Others on WorkPieces.
«Authenticity»
known = [USer]
unknown = [Others]
influenced = []

Figure 5.11.: *Simple Workpieces Problem Frame with Authenticity Requirement*

This authenticity requirement can be described with the predicate as follows:

$$\begin{aligned} &\forall\ influenced : WorkPieces;\ known : USer;\ unknown : Others \bullet \\ &auth_{att}(\{influenced\}, \{known\}, \{unknown\}) \end{aligned}$$

5.1.6. Security Management

Security management has several objectives (International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC), 2009a):

- management of security attributes (e.g., the Access Control Lists, and Capability Lists),

- management of security functions (e.g., selection of functions, and rules or conditions influencing the behavior of the security functions),
- management of data for security functions (e.g., security attribute expiration, revocation),
- definition of security roles,

A typical security management statement is that

> Valid clients should be able to manage, i.e. change security data (used for other security functions), but not attackers.

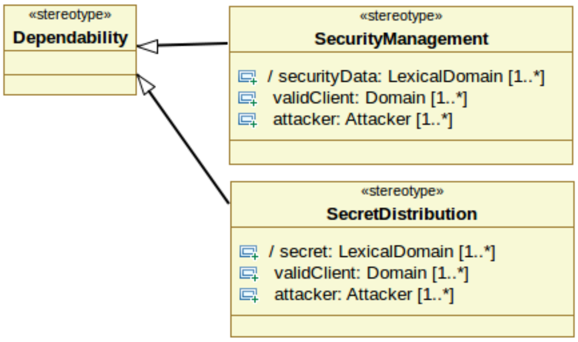

«stereotype»
Dependability
«stereotype»
SecurityManagement
/ securityData: LexicalDomain [1..*]
validClient: Domain [1..*]
attacker: Attacker [1..*]
«stereotype»
SecretDistribution
/ secret: LexicalDomain [1..*]
validClient: Domain [1..*]
attacker: Attacker [1..*]

Figure 5.12.: *UML Dependability Problem Frames Profile - Security Management / Secret Distribution*

A statement about security management is modeled as a class with the stereotype ≪SecurityManagement≫ in our profile. This stereotype is a specialization of the stereotype ≪Dependability≫. The stereotype with the attributes securityData, validClient, and attacker is shown in in Fig. 5.12. Three aspects have to be specified for a security management requirement:

1. The security data are data used for the implementation of a security functionality, e.g., the access rules for access control. The stereotype attribute securityData is modeled as a derived attribute. It is derived from the dependencies with the stereotype ≪constrains≫ (see Appendix C, Listing C.86 for the postcondition of the operation to derive the stereotype attribute securityData). Dependencies with the stereotype ≪constrains≫ shall only point to lexical domains. The corresponding OCL expression is similar to the expression for authentication (see Appendix C, Listing C.87 for the precondition of the operation to derive the stereotype attribute securityData).

2. The valid clients should be able to manage, i.e. change the security data. It is modeled with the stereotype attribute validClient. It must be assumed that the valid clients only performs correct changes on security data. This attribute need to be specified. This is required by the multiplicity of [1..*] (and can be validated by the OCL expression in Appendix C, Listing C.88.

3. The attackers should not be able to manage, i.e. change the security data. It is modeled with the stereotype attribute attacker. This is required by the multiplicity of [1..*] (and can also be validated by the OCL expression in Appendix C, Listing C.88.

The authenticity requirement is often a dependency from other security requirements. In this case it complements same same requirement as the requirement with an authentication dependency. Authenticity requirement are not necessary for protection against random faults.

It is possible to generate the statement text from the stereotype attributes validClient, attacker, and securityData (constrained lexical domains).

The security requirement pattern can be expressed by the following dependability predicate:

$$\begin{aligned}&man_{att} : \mathbb{P}\, \mathit{ClassWithLexicalDomainStereotype} \times \mathbb{P}\, \mathit{ClassWithDomainStereotype} \times \\ &\quad \mathbb{P}\, \mathit{ClassWithAttackerStereotype} \to \mathit{Bool}\end{aligned}$$

The predicate $man_{att}(sd, vc, a)$ means that valid clients in the set vc should be able to manage, i.e. change security data (used for other security functions) in the set sd, but not the attackers in the set a.

The following pattern can be used to define the authentication requirement:

$$\begin{aligned}&\forall\, \mathit{securityData} : \mathit{ClassWithLexicalDomainStereotype};\\ &\quad \mathit{validClient} : \mathit{ClassWithDomainStereotype};\\ &\quad \mathit{attacker} : \mathit{ClassWithAttackerStereotype} \bullet\\ &man_{att}(\{\mathit{securityData}\}, \{\mathit{validClient}\}, \{\mathit{attacker}\})\end{aligned}$$

For all data used to enforce security functions, the integrity must be ensured. Additionally, the ValidClient must be authenticated before. Therefore, we can state (considering that StoredData is a special InfluencedDomain):

$$\begin{aligned}&\forall\, \mathit{securityData} : \mathit{ClassWithLexicalDomainStereotype};\\ &\quad \mathit{validClient} : \mathit{ClassWithDomainStereotype};\\ &\quad \mathit{attacker} : \mathit{ClassWithAttackerStereotype} \bullet\\ &man_{att}(\{\mathit{securityData}\}, \{\mathit{validClient}\}, \{\mathit{attacker}\})\\ &\quad \Rightarrow (int_{att}(\{\mathit{securityData}\}, \{\text{"inform"}\}, \{\mathit{validClient}\}, \{\mathit{attacker}\})\\ &\quad \wedge\, auth_{att}(\{\mathit{securityData}\}, \{\mathit{validClient}\}, \{\mathit{attacker}\}))\end{aligned}$$

Since the security management includes a functional aspect, it can be used without extending a functional requirement. Nevertheless, it should complement the related functional requirement. Figure 5.13 shows a frame diagram for security management.

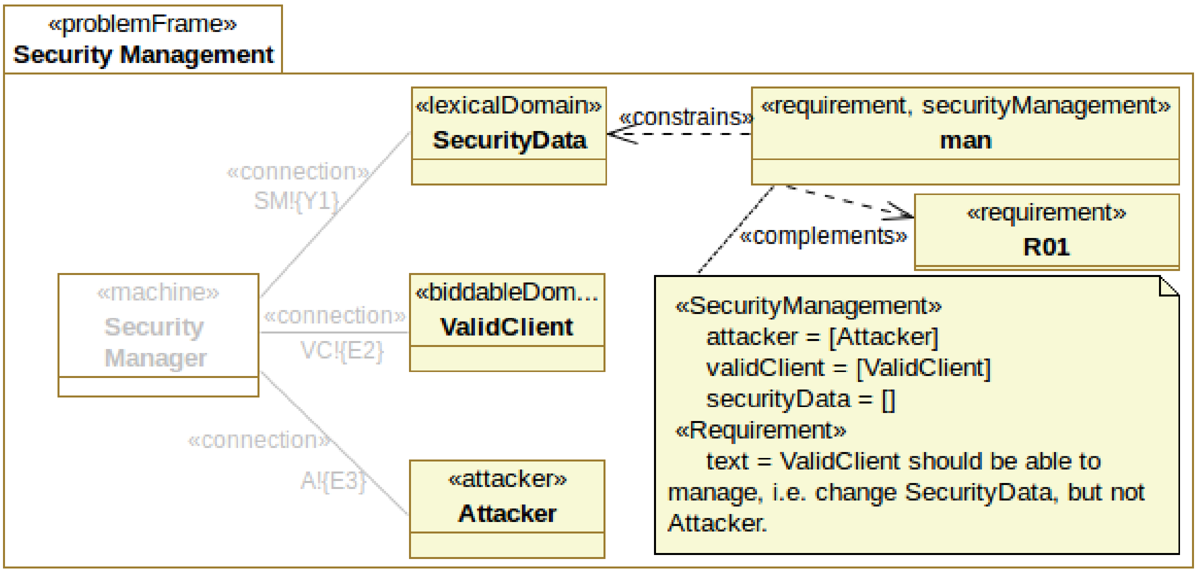


Figure 5.13.: *Security Management Requirement*

This security management requirement can be described with the predicate as follows:

$$\begin{aligned}&\forall\, \mathit{securityData} : \mathit{SecurityData};\ \mathit{validClient} : \mathit{ValidClient};\ \mathit{attacker} : \mathit{Attacker} \bullet\\ &man_{att}(\{\mathit{securityData}\}, \{\mathit{validClient}\}, \{\mathit{attacker}\})\end{aligned}$$

5.1.7. Secret Distribution

Secret distribution is a special security management activity. It additionally requires, that the secret, a special stored or transmitted data is kept confidential (e.g. for cryptographic key management in International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC) (2009a), confidentiality is necessary). A typical security secret distribution requirement is that

> Valid clients should be able to access secret (used for other security functions), but not attackers.

The statement about secret distribution is modeled in the same way as the security management stereotype with similar attributes (see Fig. 5.12). The OCK constraints are described in Appendix C, Listing C.89, C.90, and C.91.

It is possible to generate the statement text from the stereotype attributes validClient, attacker, and secret (constrained lexical domains).

The security requirement pattern can be expressed by the following dependability predicate:

$$\begin{aligned}&\mathit{dist}_{\mathit{att}} : \mathbb{P}\ \mathit{ClassWithLexicalDomainStereotype} \times \mathbb{P}\ \mathit{ClassWithDomainStereotype}\\&\quad \times \mathbb{P}\ \mathit{ClassWithAttackerStereotype} \rightarrow \mathit{Bool}\end{aligned}$$

The predicate $\mathit{dist}_{\mathit{att}}(s, vc, a)$ means that valid clients in the set vc should be able to access secrets (used for other security functions) in the set s, but not the attackers in the set a.

The following pattern can be used to define the distribution requirement:

$$\begin{aligned}&\forall\, \mathit{secret} : \mathit{ClassWithLexicalDomainStereotype};\\&\quad \mathit{validClient} : \mathit{ClassWithDomainStereotype};\\&\quad \mathit{attacker} : \mathit{ClassWithAttackerStereotype} \bullet\\&\mathit{dist}_{\mathit{att}}(\{\mathit{secret}\}, \{\mathit{validClient}\}, \{\mathit{attacker}\})\end{aligned}$$

Secret distribution can be traced back to security management and confidentiality:

$$\begin{aligned}&\forall\, \mathit{secret} : \mathit{ClassWithLexicalDomainStereotype};\\&\quad \mathit{validClient} : \mathit{ClassWithDomainStereotype};\\&\quad \mathit{attacker} : \mathit{ClassWithAttackerStereotype} \bullet\\&\mathit{dist}_{\mathit{att}}(\{\mathit{secret}\}, \{\mathit{validClient}\}, \{\mathit{attacker}\}) \Rightarrow\\&\quad (\mathit{man}_{\mathit{att}}(\{\mathit{secret}\}, \{\mathit{validClient}\}, \{\mathit{attacker}\}) \wedge\\&\quad \mathit{conf}_{\mathit{att}}(\{\mathit{secret}\}, \{\mathit{validClient}\}, \{\mathit{attacker}\}))\end{aligned}$$

It is possible to generate the statement text from the attributes validClient and attacker and the constrained causal domains.

The stereotype ≪SecretDistribution≫ can be used in the same way as the stereotype ≪SecurityManagement≫ (see Fig. 5.13). The corresponding secret distribution requirement can be described with the predicate as follows:

$$\begin{aligned}&\forall\, \mathit{secret} : \mathit{SecurityData};\ \mathit{validClient} : \mathit{ValidClient};\ \mathit{attacker} : \mathit{Attacker} \bullet\\&\quad \mathit{dist}_{\mathit{att}}(\{\mathit{secret}\}, \{\mathit{validClient}\}, \{\mathit{attacker}\})\end{aligned}$$

5.2. Procedure to Use the Dependability Extension

This section describes how to work with the UML profile for problem frames for dependable systems.

To use our profile and apply the dependability patterns, we assume that **hazards and threats are identified, and a risk analysis** has been performed. We also assume, that the functional requirements and the environment are described because dependability requirements can only be guaranteed for some specific intended environment. For example, a device may be dependable for personal use, but not for military use with more powerful attackers or a non-reliable power supply.

From hazards and threats an **initial set of dependability requirements can be identified**. These requirements supplement the previously described functional requirements.

For each dependability requirement, a pattern from Section 5.1 should be selected. After an appropriate pattern is determined, is must be connected with the concrete domains from the environment description. To prepare the analysis of depending and interacting requirements, the corresponding predicates should be instantiated. This instantiation is performed by replacing the stereotype names by classes with this stereotype or a derived stereotype.

The connected domains must be described using facts and assumptions. For an attacker, at least the attributes of the stereotype must be defined (objective, equipment, skill, time to attack, time to prepare). Via these assumptions, *threat models* are integrated into the development process using dependability patterns.[1]. The values for probabilities can be usually extracted from the risk analysis.

5.3. CACC Case Study

The approach is illustrated on the case study introduced in Section 4.3.

5.3.1. Identify Hazards and Threats, Perform Risk Analysis

The **hazard** to be avoided is an unintended acceleration or deceleration (that may lead to a rear-end collision). The considered **threat** is an attacker who sends wrong messages to the car in order to influence its speed.[2]

5.3.2. Describe Environment

Examples for domain knowledge of the CACC in the **described environment** are physical properties about acceleration, braking, and measurement of the distance (relevant for safety). Other examples are the assumed intention, knowledge and equipment of an attacker. The objective of the attacker could be to change the speed of the car, in order to produce a rear-end collision. We assume here that the attacker can only access the connection domain WiFi_WAVE interface. The context diagram for the CACC is shown in Section 4.3, Fig. 4.19.

5.3.3. Describe Dependability Requirements

The next step is to **identify an initial set of dependability requirements**. For the functional requirements R1 and R2, the following security requirement can be stated using the textual pattern from Section 5.1.2:

[1] To analyze and identify the threats, e.g., attack trees (Schneier, 1999) can be used

[2] The **risk analysis** is left out here, see Section 5.2.

> For Driver, the influence (as described in R1 and R2) on the Car (brake, accelerate) must be either correct or in case of a modification by CACCAttacker the Car (EngineActator_Brake) shall not brake/accelerate and the Car shall inform driver.

A problem diagram including this integrity requirement is depicted in Fig. 5.14. It complements the requirements R1_R2. It refers to an attacker (the CACCAttacker) within the stereotype attribute attacker and also refers to the domain constrained by R1_R2 (the Car). The Car is constrained because the EngineActuator_Brake as part of the car should not be influenced. Additionally, the Car is constrained because it acts as a display to warn the driver.

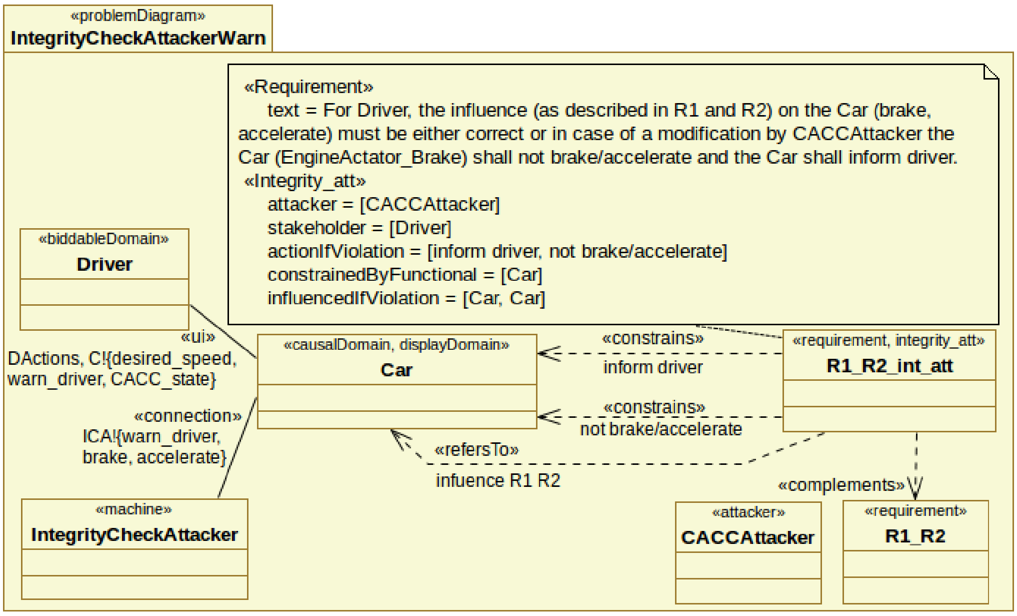

«problemDiagram»
IntegrityCheckAttackerWarn
«Requirement»
text = For Driver, the influence (as described in R1 and R2) on the Car (brake, accelerate) must be either correct or in case of a modification by CACCAttacker the Car (EngineActator_Brake) shall not brake/accelerate and the Car shall inform driver.
«Integrity_att»
attacker = [CACCAttacker]
stakeholder = [Driver]
actionIfViolation = [inform driver, not brake/accelerate]
constrainedByFunctional = [Car]
influencedIfViolation = [Car, Car]
«biddableDomain»
Driver
«ui»
DActions, C!{desired_speed, warn_driver, CACC_state}
«causalDomain, displayDomain»
Car
«constrains»
inform driver
«constrains»
not brake/accelerate
«requirement, integrity_att»
R1_R2_int_att
«connection»
ICA!{warn_driver, brake, accelerate}
«refersTo»
infuence R1 R2
«complements»
«machine»
IntegrityCheckAttacker
«attacker»
CACCAttacker
«requirement»
R1_R2

Figure 5.14.: *CACC Problem Diagram for Integrity Checks considering an Attacker*

All OCL constraints defined for the profile were checked. With checking these constraints, we detected several minor mistakes (e.g., wrong names), and we detected that the original version of our problem diagram did not refer to the domain constrained in the requirement.

These requirements can be expressed using the integrity predicates

$$\begin{aligned}\forall\, c :\ & Car, a : CACCAttacker, d : Driver \bullet \\ & int_{att}(\{c\}, \{\text{'not brake/accelerate', 'inform the driver'}\}, \{c, c\}, \{d\}, \{a\})\end{aligned} \tag{5.1}$$

The first occurrence of the variable *mab* in Equation 5.1 refers to the influenced domain as described in the functional requirement, and the second occurrence of *mab* expresses that this domain is not influenced in case of an attack.

A safety requirement is to keep a safe distance to the car ahead while being activated (see **R1** and **R2**). For each safety requirement the integrity or the reliability must be defined. For the CACC only integrity is required, because it is safe to switch off the functionality and inform the driver in case of a failure. The risk analysis performed in the first step showed that a probability of at most 10^{-7} untreated random errors per hour (that may lead to an accident) can be accepted. Hence, for **R1** and **R2** can be stated that

> With a probability of $1 - 10^{-7}$ per hour, one of the following things should happen: service (as described in R1 and R2) with influence on the EngineActuator_Brake must be correct, or the Car (EngineActator_Brake) shall not brake/accelerate and the Car shall inform driver.

A problem diagram including this dependability requirement is depicted in Fig. 5.15. This integrity requirement also complements the requirements R1 and R2. It defines a probability and also refers to the domain constrained by R1 and R2 (the Car). Additionally, the Car is constrained because it acts as a display to inform the driver, and it is constrained because the EngineActuator_Brake as part of the car should not be influenced (no brake, no accelerate).

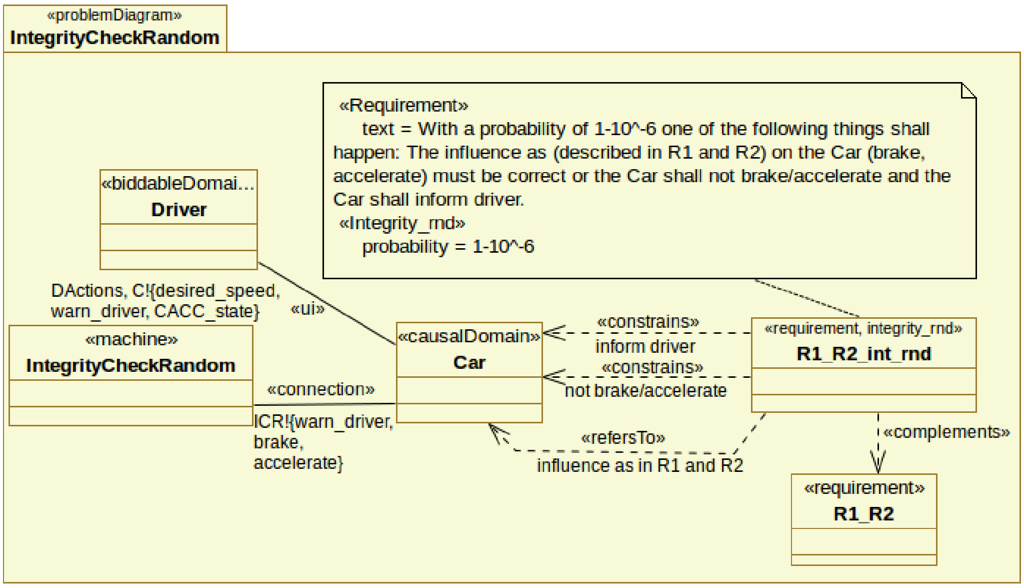

«problemDiagram»
IntegrityCheckRandom
«Requirement»
text = With a probability of 1-10^-6 one of the following things shall happen: The influence as (described in R1 and R2) on the Car (brake, accelerate) must be correct or the Car shall not brake/accelerate and the Car shall inform driver.
«Integrity_rnd»
probability = 1-10^-6
«biddableDomai...
Driver
DActions, C!{desired_speed, warn_driver, CACC_state}
«ui»
«machine»
IntegrityCheckRandom
«connection»
ICR!{warn_driver, brake, accelerate}
«causalDomain»
Car
«constrains»
inform driver
«constrains»
not brake/accelerate
«requirement, integrity_rnd»
R1_R2_int_rnd
«refersTo»
influence as in R1 and R2
«complements»
«requirement»
R1_R2

Figure 5.15.: *CACC Problem Diagram for Integrity Check Random*

The corresponding predicate is:

$$\forall\, c : \quad Car \bullet \\ int_{rnd}(\{c\}, \{\text{'not brake/accelerate', 'inform the driver'}\}, \{c, c\}, 1 - 10^{-7}) \qquad (5.2)$$

Additionally, to satisfy the drivers buying the CACC:

> The service (described in R1 and R2) with influence on the Car (EngineActuator_Brake) must be available with a probability of $1 - 10^{-7}$.

The corresponding problem diagram is depicted in Fig. 5.16. Because the reliability requirement has the same properties, it is also included in this diagram.

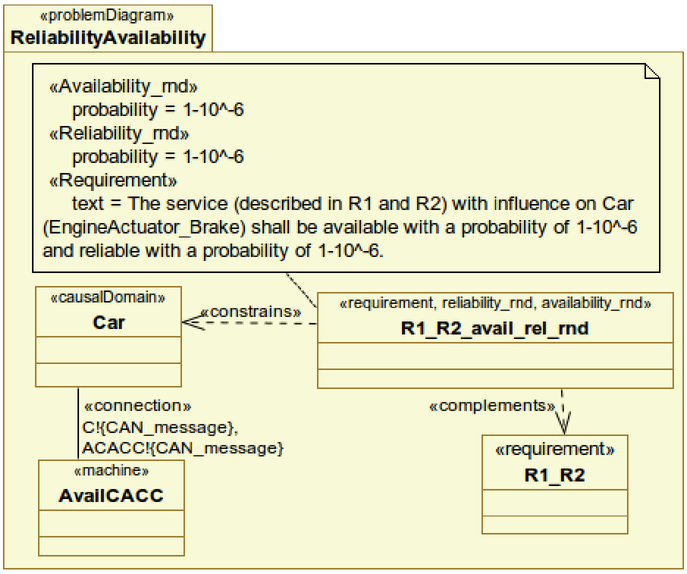


Figure 5.16.: *CACC Problem Diagram for Availability and Reliability*

This requirement can be expressed with the predicate

$$\forall\, c : Car \bullet avail_{rnd}(\{c\}, \emptyset, 1 - 10^{-6}) \tag{5.3}$$

For availability, we only consider random faults, because for the corresponding security requirement we have to limit the group of users (the service is provided for) as described in Section 5.1.3, and this is not possible in the described environment. Since both integrity and availability for keeping the distance are required, the following reliability requirement can be stated:

$$\forall\, c : Car \bullet reli_{rnd}(\{c\}, \emptyset, 1 - 10^{-6}) \tag{5.4}$$

5.4. Relation to Security Problem Frames

In this section, we demonstrate, how to use the profile to express already published security problem frames. In contrast to the security problem frames (SPF), the patterns presented in this thesis separates functional requirements from dependability requirements.

The following frames have been already published:

- SPF confidential data transmission / Secure Data Transmission Frames (Schmidt, Hatebur, & Heisel, 2011; Schmidt, 2010a; Hatebur et al., 2007a; Hatebur & Heisel, 2005a)
- SPF confidential data storage (Schmidt et al., 2011; Schmidt, 2010a, Section 4.2.2)
- SPF integrity-preserving data transmission (Schmidt, 2010a, Section 4.2.3)
- SPF integrity-preserving data storage (Schmidt, 2010a, Section 4.2.4)
- SPF authentication / Accept Authentication Frame / Submit Authentication Frame (Schmidt, 2010a; Hatebur et al., 2007a, 2006; Hatebur & Heisel, 2005a)
- SPF distributing secrets / Distribute Security Information Frame (Schmidt, 2010a; Hatebur et al., 2007a; Hatebur & Heisel, 2005a)

- SPF anonymity (Hatebur, Heisel, & Schmidt, 2007b)
- SPF accountability (Hatebur, Heisel, & Schmidt, 2008a)

All of these frames can be expressed using our Problem Frames Profile for UML. As an example, the SPF confidential data transmission (depicted in Fig. 5.17) is expressed using the profile of Section 5.1 as shown in Fig. 5.18.

Name SPF confidential data transmission

Intent Conceal data (e.g., files, raw data, E-Mails, etc.) transmitted from a sender to a recipient over some communication medium (e.g., LAN, Wifi, Internet, etc.).

Frame Diagram Figure 5.17 shows the frame diagram of the SPF confidential data transmission.

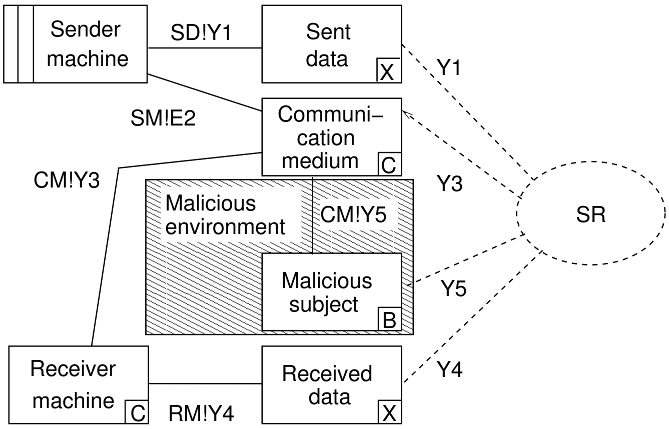


Figure 5.17.: *Security Problem Frame "Confidential Data Transmission"*

Predefined Interfaces The interfaces of the SPF confidential data transmission are defined as follows:

$$
\begin{aligned}
\mathsf{Y1} &= \{\mathsf{ContentOfSentData}\}\\
\mathsf{E2} &= \{\mathsf{TransmitContentOfSentData}\}\\
\mathsf{Y3} &= \{\mathsf{TransmittedContentOfSentData}\}\\
\mathsf{Y4} &= \{\mathsf{ContentOfReceivedData}\}\\
\mathsf{Y5} &= \{\mathsf{ObservationsMS}\}
\end{aligned}
$$

Informal Description The domain Sent data denotes the data that is sent by a sender, represented by the machine domain Sender machine. The data (ContentOfSentData) is transferred from the domain Sent data to the machine via the interface SD!Y1 (between Sender machine and Sent data). Analogously, the domain Received data denotes the data that is received by the domain Receiver machine. The data (ContentOfReceivedData) is transferred from the domain Receiver machine to the domain Received data via the interface RM!Y4 (between Receiver machine and Received data).

The data is sent (TransmitContentOfSentData) over some network, which is represented by the domain Communication medium using the interface SM!E2 (between machine and Communication medium). Then, the domain Communication medium forwards the data (TransmittedContentOfSentData) to the domain Receiver machine using the interface CM!Y3 (between Communication medium and Receiver machine). Informally speaking, the sender machine generates the data to be transmitted from the sent data, and the receiver machine generates the received data from the data sent over the communication medium.

The malicious environment is represented by the domain Malicious subject. It is graphically emphasized by the hatched area in Fig. 5.17. The domain Sent data represents the data to be protected against the malicious environment. The Malicious subject domain uses the interface CM!Y5 (between Malicious subject and Communication medium) to eavesdrop on the Communication medium domain. Some observations (ObservationsMS), e.g., meta-information about Sent data such as its length or type, can be received from the Communication medium domain using the interface CM!Y5.

Security Requirement Template The security requirement template (SR) is described as follows:

> Preserve confidentiality of *sent data* for *sender machine* and prevent disclosure via *communication medium* to *malicious environment.*

SPF expressed using the Problem Frames profile for UML This frame corresponds to Jackson's transformation frame with the connection domains Communication medium and Receiver machine complemented with a confidentiality requirement. The problem frame diagram is depicted in Fig. 5.18. This diagram additionally contains the supplemented functional requirement R (required by the constraint described in Listing 5.1) and the stakeholder DataOwner (required by the constraint described in Appendix, Listing C.60).

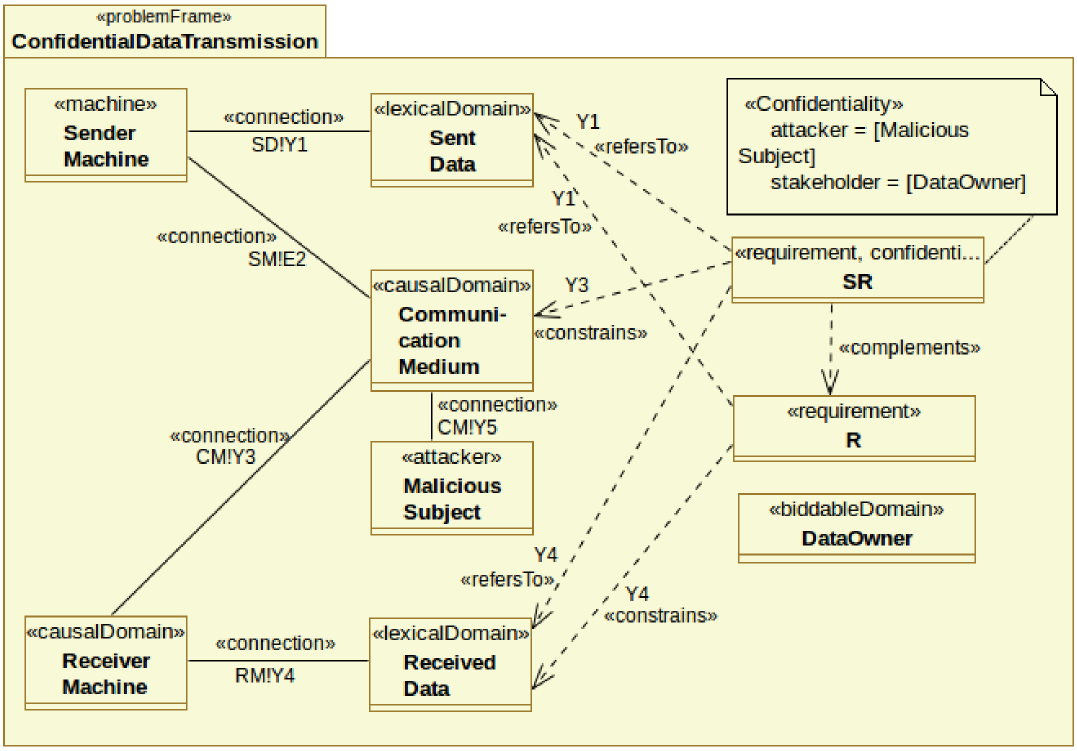


Figure 5.18.: *Security Problem Frame "Confidential Data Transmission" with UML4PF*

Concretized Security Problem Frames (CSPFs) (see (Schmidt et al., 2011)) are not explicitly created by the procedure presented in this thesis since too many frames are necessary to cover the problems of real projects. Therefore, we replace the frames by rule that describe the changes on the model when a generic mechanism is selected. Details are discussed in Chapter 6.

5.5. Related Work

Fabian, Gürses, Heisel, Santen, and Schmidt (2010) give an overview on other methods for security requirements engineering. It compares MSRA/CREE, SQUARE, Misuse cases, SecureUML, UMLsec, KAOS, Secure Tropos, Secure i*, GBRAM, Abuse frames, SEPP, SREF, CORAS, Model-based ISSRM, CC, and SREP. Our approach presented in this chapter inherits the properties from SEPP. The notions of assumptions and facts (domain knowledge) and specification have the same meaning as the notions in the conceptual framework described in Fabian et al. (2010). Vulnerabilities are not considered since they cannot be an element of the requirements engineering. Security goal are seen as more general requirements. Threat and risk analysis are also not covered by the requirements engineering steps described here, but they are seen as a necessary input. Additionally, to the requirements in SEPP (Hatebur et al., 2008a), the stakeholder are referenced in the requirements, the functional requirements and the security requirements are separated.

Haley, Laney, Moffett, and Nuseibeh (2008) present an approach for security requirements elicitation and analysis that is similar to our approach. This approach is limited to security requirements and within the paper no security requirement patterns are presented.

MARTE (UML Revision Task Force, 2011) is a well-known UML profile that allows the annotation of embedded and real-time systems with performance attributes. Bernardi, Merseguer, Cortellessa, and Berardinelli (2009) also present a UML profile to express the non-functional properties reliability, availability and performance. Their stereotype can be used to annotate use case diagrams, deployment diagrams, and sequence diagrams. It is an extension of the MARTE profile for performance, but does not cover security aspects. The stereotype attributes are similar to the stereotype attributes in our profile. The profile is not designed to extend the problem frames approach and does not systematically support the engineer in describing the necessary properties.

Rodríguez, Merseguer, and Bernardi (2010) present an extension for security. The profile focuses on security faults, vulnerabilities and attacks. These aspects can be used to analyze given systems, but are are not appropriate to describe security requirements.

Jürjens (2005) describes a profile for UML 1.4 to extend UML specifications with security aspects. This profile cannot be used for requirements engineering. Therefore, Mouratidis and Jürjens (2010a) use Secure Tropos (Mouratidis, 2004a) to transform security requirements to design.

The Common Criteria (International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC), 2009a), Part 2 define a large set of so-called *Security Functional Requirements (SFRs)* as patterns for requirements. But some of these SFRs directly anticipate a solution, e.g. the SFR in the class ***f**unctional requirements for **c**ryptographic **s**upport* with the name ***c**ryptographic **op**eration* (FCS_COP) specifies the cryptographic algorithm, key sizes, and the assigned standard to be used. Together with this SFR, the ***c**ryptographic **k**ey **m**anagement* (FCS_CKM) has to be specified. To require confidentiality, alternatively, an ***a**ccess **c**ontrol **p**olicy* (FDP_ACC, in class ***f**unctional requirements for **d**ata **p**rotection*)) and an ***a**cess **c**ontrol **f**unction* (FDP_ACF) can be be specified. The SFRs in the Common Criteria are limited to security issues. Table 5.1 shows a fragment of our mapping from dependability patterns to SFRs.

$conf_{att}$	Cryptographic operation (FCS_COP) and Cryptographic key management (FCS_CKM)
	Access control policy (FDP_ACC) and Access control functions (FDP_ACF)
	Inter-TSF user data confidentiality transfer protection (FDP_UCT)
	...
int_{att}	Cryptographic operation (FCS_COP) and Cryptographic key management (FCS_CKM) and Inter-TSF user data integrity transfer protection (FDP_UIT)
	Cryptographic operation (FCS_COP) and Cryptographic key management (FCS_CKM) and Stored data integrity (FDP_SDI)
	Access control policy (FDP_ACC) and Access control functions (FDP_ACF)
	Information flow control policy (FDP_IFC) and Information flow control functions (FDP_IFF)
	...
$avail_{att}$	Inter-TSF availability (FPT_ITA)
$auth_{att}$	Authentication failures (FIA_AFL) User attribute definition (FIA_ATD) User identification and authentication (FIA_UAU, FIA_UID) ...
man_{att} $dist_{att}$	Management of functions in TSF (FMT_MOF)
	Management of security attributes (FMT_MSA)
	Management of TSF data (FMT_MTD)

Table 5.1.: *Mapping: Dependability Patterns vs. SFRs*

5.6. Conclusions and Future Work

In this chapter, we have presented patterns that can be used to describe security requirements. The approach extends problem frames from Chapter 4 to describe dependability.

We have defined a set of patterns that can be used to describe and analyze dependability requirements. These patterns are represented by

- a textual pattern with references to relevant domains,
- stereotypes that can be used to extend a problem diagram, and
- a corresponding predicate.

To provide tool support, we have defined a Unified Modeling Language (UML) profile (UML Revision Task Force, 2010c) that allows us to represent problem frames in UML. This UML profile is then augmented with stereotypes that support the expression of dependability requirements. The stereotypes are complemented by constraints expressed in the Object Constraint Language (OCL). They can be checked, as described in Section 4.2

These constraints express important integrity conditions, for example, that security requirements must explicitly address a potential attacker. By checking the different OCL constraints, we can substantially aid system and software engineers in analyzing dependability requirements.

We have set up 35 OCL constraints for requirements engineering concerning dependability. These constraints show how functional requirements can be complemented by dependability requirements.

This chapter also describes the process to work with our UML profile for problem frames for dependable systems and shows the to SEPP with security problem frames.

The patterns for dependability requirements may not be complete. Of curse, patterns for maintainability requirements are missing and we have not specified any patterns for privacy requirements and accountability requirements.

The advantages of our general approach are preserved when dependability requirements are added as suggested in this chapter:

- Artifacts from the analysis development phase that are part of a model created with our profile can be re-used in later phases in the software development process.
- The notation is based on UML. UML is commonly used in software engineering, and many developers are able to read our models.
- The concept is not tool-specific. It can be easily adapted to other UML2 tools that allow to specify new stereotypes.

In summary, the advantages of our patterns for dependability requirements are:

- The dependability statements are re-usable for different projects.
- A manageable number of statement types can be used for a wide range of problems, because they are separated from the functional requirements.
- Statements expressed using our profile refer to the environment description and are independent from solutions. Hence, they can be easily re-used for new product versions.
- A generic textual description of the requirement or the domain knowledge can be generated from other model elements.
- Statements expressed using our profile help to structure and classify the dependability requirements. For example, integrity statements can be easily distinguished from availability statements. It is also possible to trace all dependability statements that refer to one domain.

In the future, we plan to consider also privacy and accountability requirements. We also plan to check the requirement patterns for completeness by associating more design patterns to dependability requirements.

Chapter 6

Analysis of Dependability Requirements

Dependability requirements should be described and analyzed. When the requirements are described as suggested in Chapter 5, missing and conflicting requirements can be identified.

This chapter presents a construction system and shows how to work with our construction system built up on the predicates defined in Section 5.1. It can be used to find possible interactions with other dependability requirements and helps to complete the dependability requirements by a set of defined necessary conditions for each mechanism that can be used to solve dependability problems.

This chapter is based on Hatebur and Heisel (2009b), in which we have presented a foundation for requirements analysis of dependable systems, based on problem frames (Jackson, 2001). The dependencies for security requirements are based on Hatebur et al. (2008a).

In Section 6.1, the construction system is presented. Section 6.2 describes how to integrate the use of the dependability patterns into a system development process. The case study in Section 6.3 applies that process to a cooperative adaptive cruise control system. Section 6.4 discusses related work, and the chapter closes with a summary and perspectives in Section 6.5.

6.1. Construction System

For each dependability requirement pattern from Chapter 5, we provide a table describing mechanisms that can be used to address the requirement. For each mechanism in Column 1, we provide a set of

- requirements that may interact and lead to conflicts in Column 2,
- domains that need to be introduced or considered in Column 3,
- necessary conditions that must be either assumed or established in Column 4, and
- related requirement that may be also relevant for the problem in Column 5.

In the following paragraphs, we explain these columns:

When a mechanism that addresses a requirement is chosen, this may have an negative impact to another requirements. In case of such a conflict, either another mechanism has to be chosen, or one of the requirements have to be changed (Column 2).

A mechanism usually needs some domains (e.g., keys) to realize the requirement to be addressed. These domains maybe already described in the context diagram. If it is not possible to consider an already described domain, we have to introduce this domain (Column 3).

For these introduced or considered domain, it is necessary to fulfill some conditions. If these conditions are not fulfilled, the selected generic mechanism will not address the initial requirement, e.g., if a key used for encryption is not kept confidential, the initial confidentiality requirement cannot be fulfilled (Column 4). Therefore, these necessary conditions lead to new requirements or the necessary condition have to be established by the environment. For new requirements, again a mechanism has to be selected and the whole procedure has to be repeated for this requirement. To avoid an endless cycle at some point, the necessary conditions have to established by the environment (expressed as domain knowledge, i.e., facts or assumptions).

If an initial requirement has to be addressed by the machine with a mechanism, there are often related requirements, e.g. if same data has the be kept confidential, often it is important the its integrity is preserved. The developer has to decide if the related requirements are necessary in the context of the system (Column 5).

These tables have been created from project experience and patterns, describing the generic mechanisms (Hamner (2007), Schumacher, Fernandez-Buglioni, Hybertson, Buschmann, and Sommerlad (2006), Schumacher (2003)). Neither the set of dependability requirements nor the set of generic mechanisms is complete, but can be easily extended. For each necessary condition in the tables, a corresponding requirement is provided, and for each requirement at least one generic mechanism is suggested.

The tables are constructed using the predicates introduced in Chapter 5. The dependability requirement predicates in the table refer to

- all instances d of a causal domain to be protected (e.g., transmitted or stored data),
- all instances of the causal domains constrained by the functional requirement c_1 and c_2,
- all instances of a biddable domain (e.g., users or stakeholders) u, to all instances of the Attacker domains a,
- all instances of the Machine domain including all relevant connection domains m, and
- a set of specific actions A as described in Section 5.1.2.

6.1.1. Confidentiality

To achieve confidentiality ($conf_{att}$) (as introduced in Section 5.1.1), e.g., *symmetric encryption*, *asymmetric encryption*, or *access control* can be used as mechanisms. All mechanisms may lead to reduced availability - intentionally for the attacker, but if a key is lost, maybe also for the stakeholder. A necessary authentication may also lead to reduced availability (e.g., due to login time or lost credentials). Depending on the mechanism, additional domains have to be introduced or considered. For all mechanisms, the Machine needs to be protected from modification and its data from modification since it processes data that is not protected.

For symmetric encryption, a Key1 for sender, a Key2 for receiver, and the plain text including related data (PlainData) have to be introduced or considered. Note that Key1 and Key2 need to be the same but are stored at different places. We have to ensure that the internal state of the machine, the plain text including related data, the key of the sender, and the key of the receiver are protected from disclosure. The integrity of both keys has to be ensured, because otherwise an attacker can change the value to a known value in order to decrypt the protected data. For symmetric encryption, the keys have to be distributed before in a secure manner (confidential and integrity-preserving).

For asymmetric encryption, a SenderKey, a ReceiverKey, and the plain text including related data (PlainData) have to be introduced or considered. We have to ensure the same conditions as for symmetric encryption, except that the key used to encrypt the message (SenderKey) need not to be protected against disclosure. For asymmetric encryption, it is sufficient to ensure the integrity of the SenderKey when distributing the keys – confidentiality is only necessary for the ReceiverKey.

For access control, instead of keys, AccessRules have to be considered. These access rules are used to allow or prevent the access of users to the data at the user interface. But access control does not solve the confidentiality problem itself: the data is unprotected if the computer is stolen. Therefore, protecting the confidentiality is still necessary. Additionally, the data processed by the machine and the plain text including related data (PlainData) have to be protected from disclosure and its integrity have to be preserved. Also the integrity of the access rules have to be preserved. Modification of the access rule shall be only possible for trusted Administrators. If confidentiality is required, often integrity is also important.

For the confidentiality predicate $conf_{att}(\{d\}, \{u\}, \{a\})$, Table 6.1 shows a set of mechanisms with possible interaction, introduced / considered domains, necessary conditions and related requirements.

Generic mechanism	**Possible interaction**	**Introduced / considered domains**	**Necessary conditions**	**Related**
symmetric encryption	$avail_*(\{d\}, \{a\}, *)$ $avail_*(\{d\}, \{u\}, *)$[1]	$k_{Snd} : Key_1$ $k_{Rcv} : Key_2$ $pr : PlainData$ $m : Machine$	$conf_{att}(\{m\}, \{u\}, \{a\})$ $int_{att}(\{m\}, *, *, \{u\}, \{a\})$ $conf_{att}(\{pr\}, \{u\}, \{a\})$ $conf_{att}(\{k_{Snd}\}, \{u\}, \{a\})$ $int_{att}(\{k_{Snd}\}, *, *, \{u\}, \{a\})$ $conf_{att}(\{k_{Rcv}\}, \{u\}, \{a\})$ $int_{att}(\{k_{Rcv}\}, *, *, \{u\}, \{a\})$ $dist_{att}(\{k_{Snd}\}, \{u\}, \{a\})$ $dist_{att}(\{k_{Rcv}\}, \{u\}, \{a\})$	*int*
asymmetric encryption	$avail_*(\{d\}, \{a\}, *)$ $avail_*(\{d\}, \{u\}, *)$ [1]	$k_{Snd} : SenderKey$ $k_{Rcv} : ReceiverKey$ $pr : PlainData$ $m : Machine$	$conf_{att}(\{m\}, \{u\}, \{a\})$ $int_{att}(\{m\}, *, *, \{u\}, \{a\})$ $conf_{att}(\{pr\}, \{u\}, \{a\})$ $conf_{att}(\{k_{Rcv}\}, \{u\}, \{a\})$ $int_{att}(\{k_{Snd}\}, *, *, \{u\}, \{a\})$ $int_{att}(\{k_{Rcv}\}, *, *, \{u\}, \{a\})$ $man_{att}(\{k_{Snd}\}, \{u\}, \{a\})$ $dist_{att}(\{k_{Rcv}\}, \{u\}, \{a\})$	*int*
access control	$avail_*(\{d\}, \{a\}, *)$ $avail_*(\{d\}, \{u\}, *)$[2]	$ar : AccessRules$ $pr : PlainData$ $m : Machine$ $ad : Administrator$	$conf_{add}(\{d\}, \{u\}, \{a\})$ [3] $conf_{att}(\{m\}, \{u\}, \{a\})$ $int_{att}(\{m\}, *, *, \{u\}, \{a\})$ $conf_{att}(\{pr\}, \{u\}, \{a\})$ $int_{att}(\{ar\}, *, *, \{ad\}, \{u, a\})$ $man_{rnd}(\{ar\}, \{ad\}, \{u, a\})$ $auth_{att}(\{d\}, \{u\}, \{a\})$	*int*
...	...	...	...	...

1. Lost secrets may decrease availability.
2. Necessary authentication may decrease availability for User u.
3. The stored data d must be still protected. This is often just assumed.

Table 6.1.: *Confidentiality – Dependencies*

6.1.2. Integrity

To achieve integrity of the domain constrained by the functional requirement (c_1) (as introduced in Section 5.1.2), considering random faults (int_{rnd}), e.g., *checksums* or *plausibility checks* can be used. Other mechanisms to preserve integrity are the detection pattern in (Hamner, 2007), e.g., *Fault Correlation*, *Realistic Threshold*, *Complete Parameter Checking*, or *Voting*. Further mechanisms (e.g., *Watchdog* or *test patterns*) are described in the standard ISO/IEC 61508, Part 2, Tables A.1 to A.15 (International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC), 2000). The possible interactions, introduced or considered domains, necessary conditions, and related requirements for these mechanisms are similar to those described for checksums and plausibility checks. In case of a detected fault, an action A has to be performed on another domain (c_2). For all mechanisms, the availability would be decreased if the performed action is to switch off the machine or delete corrupted data in case of a detected fault.

For all mechanisms the Machine has to be considered. A necessary condition is, that the integrity of the machine performing the integrity checks is preserved and the domain used in case of a detected fault (c_2) is reliable. A related requirement for all mechanisms maybe the integrity considering an attacker.

For the checksum mechanism, no additional necessary conditions exist. For redundancy, diverse input information (Input and DiverseInput) are used. They have to be reliable enough to achieve the required probability of the initial integrity requirement.

To achieve integrity of the domain constrained by the functional requirement (c_1) (as introduced in Section 5.1.2), considering an attacker (int_{att}), e.g., *message authentication codes (MACs)*, *cryptographic signatures*, or *access control* can be used as mechanisms.

For MAC protection, a Key1 for sender, a Key2 for receiver, and the Machine have to be introduced or considered. We have to ensure that the internal state of the machine and the key for sender and the key of the receiver are protected from disclosure because the attackers can create a valid MAC if they know the key. The integrity of the receiver keys have to be ensured, because otherwise an attacker can change the value to a known value in order to accept invalid MACs. The domain constrained in case of a detected modification (C_2, e.g., the display) has to be reliable for the stakeholder u. For MAC protection, additionally, the keys have to be distributed before in a secure manner (confidential and integrity-preserving).

For cryptographic signatures, a SenderKey, a ReceiverKey, and the Machine have to be introduced or considered. We have to ensure the same conditions as for MAC protection, except that key used to verify the signature (ReceiverKey) need not to be protected against disclosure. For cryptographic signatures, it is sufficient to ensure the integrity of the ReceiverKey when distributing the keys - confidentiality is only necessary for the SenderKey.

For access control, instead of keys, access rules (AccessRules) have to be considered. But access control does not solve the integrity preserve problem itself: the data is unprotected if the physical access is possible. Therefore, the required integrity of the data is also a necessary condition. Additionally, the data processed by the machine must be protected from disclosure and modification. The domain constrained in case of a detected modification (c_2, e.g., the display) has to be reliable for the stakeholder u. Also the integrity of the access rules have to be preserved for the Administrators.

If integrity is required, confidentiality often is also important.

For the integrity predicate $int_{rnd}(\{c_1\}, A, \{c_2\}, P_i)$, Table 6.2 shows a set of mechanisms with possible interaction, introduced / considered domains, necessary conditions and related requirements.

For the integrity predicate $int_{att}(\{c_1\}, A, \{c_2\}, \{u\}, \{a\})$, Table 6.3 shows a set of mechanisms with possible interaction, introduced / considered domains, necessary conditions and related

Generic mechanism	Possible interaction	Introduced / considered domains	Necessary conditions	Related
checksums	$avail_*(\{c_1\}, *)$ [1]	$m : Machine$	$int_{rnd}(\{m\}, A, \{c_2\}, P_i)$ $rel_{rnd}(\{c_2\}, \emptyset, P_i)$	int_{att}
plausibility checks	$avail_*(\{c_1\}, *)$ [1]	$i : Input$ $d : DiverseInput$ $m : Machine$	$int_{rnd}(\{m\}, A, \{c_2\}, P_i)$ $rel_{rnd}(\{c_2\}, \emptyset, P_i)$ $rel_{rnd}(\{d\}, \emptyset, P_i')$ $rel_{rnd}(\{i\}, \emptyset, P_i'')$ [2]	int_{att}
...	...	...	...	...

1. Availability may be decreased if modified data is just deleted.
2. It must be verified that the mechanism with P_i' and P_i'' can achieve P_i.

Table 6.2.: *Integrity Considering Random Faults – Dependencies*

requirements.

6.1.3. Availability

To achieve a certain availability of a domain (c_1) (as introduced in Section 5.1.3), considering random faults $(avail_{rnd})$, e.g., *redundancy* or *reliable hardware and software* can be used. Other mechanisms to achieve availability are described by patterns in (Hamner, 2007), e.g., *Minimize Human Intervention*, *Maximize Human Participation*, *Fallover*, or *Leaky Bucket Counter*.

For the redundancy mechanism, confidentiality maybe decreased because of more targets can be attacked. Especially, diverse machines used to achieve redundancy may have different vulnerabilities. In addition to the Machine, a redundant machine (RedundantMachine) is necessary. It is possible, that only the critical parts of the software are executed on the redundant machine.

The availability of all machines may be lower than the availability to achieve, but together the target availability has to be achieved. If availability is required, often reliability is also important.

If reliable hardware and software is used, the required probability has to be achieved by the used hardware and software of the Machine. In case of reliable hardware and software, reliability considering an attacker maybe also important.

To achieve availability of a domain (c_1) (as introduced in Section 5.1.3), considering attackers $(avail_{att})$, e.g., service degradation can be used. A Firewall that blocks requests from attackers can be used for service degradation. This firewall has to be reliable and the users who should be able to use the service have to show their authenticity. Such a firewall cannot increase the availability, but it can keep the given availability even in case of malicious requests from an identified attacker. If requests are bocked, or event the wrong requests are blocked, the availability can be decreased by a firewall.

If availability is required, reliability can also be important.

For the availability predicate $avail_{rnd}(\{c_1\}, \{u\}, P_a)$, Table 6.4 shows a set of mechanisms with possible interaction, introduced / considered domains, necessary conditions and related requirements.

For the availability predicate $avail_{att}(\{c_1\}, \{u\}, P_a)$, Table 6.5 shows a set of mechanisms with possible interaction, introduced / considered domains, necessary conditions and related requirements.

Generic mechanism	**Possible interaction**	**Introduced / considered domains**	**Necessary conditions**	**Related**
MAC	$avail_*(\{c_1\},*,*)$ [1]	$k_{Snd} : Key1$ $k_{Rcv} : Key2$ $m : Machine$	$conf_{att}(\{m\},\{u\},\{a\})$ $int_{att}(\{m\},A,\{c_2\},\{u\},\{a\})$ $rel_{att}(\{c_2\},\{u\},\{a\})$ $conf_{att}(\{k_{Snd}\},\{u\},\{a\})$ $conf_{att}(\{k_{Rcv}\},\{u\},\{a\})$ $int_{att}(\{k_{Rcv}\},*,*,\{u\},\{a\})$ $dist_{att}(\{k_{Snd}\},\{u\},\{a\})$ $dist_{att}(\{k_{Rcv}\},\{u\},\{a\})$	$conf_{att}$
cryptographic signature	$avail_*(\{c_1\},*,*)$ [1]	$k_{Snd} : SenderKey$ $k_{Rcv} : ReceiverKey$ $m : Machine$	$conf_{att}(\{m\},\{u\},\{a\})$ $int_{att}(\{m\},A,\{c_2\},\{u\},\{a\})$ $rel_{att}(\{c_2\},\{u\},\{a\})$ $conf_{att}(\{k_{Rcv}\},\{u\},\{a\})$ $int_{att}(\{k_{Rcv}\},*,*,\{c_2\},\{a\})$ $dist_{att}(\{k_{Snd}\},\{u\},\{a\})$ $dist_{att}(\{k_{Rcv}\},\{u\},\{a\})$	$conf_{att}$
access control	$avail_*(\{c_1\},*,*)$ [2]	$ar : AccessRules$ $ad : Administrator$ $m : Machine$	$int_{att}(\{c_1\},A,\{c_2\},$ _ $\{u\},\{a\})$ [3] $int_{att}(\{m\},A,\{c_2\},$ _ $\{u\},\{a\})$ $int_{att}(\{ar\},A,\{c_2\},\{u\},\{a\})$ $man_{rnd}(\{ar\},\{ad\},\{a\})$ $auth_{att}(\{d\},\{u\},\{a\})$	$conf_{att}$
...	...	...	...	...

1. Availability may be decreased if modified data is just deleted.
2. Necessary authentication may decrease availability for User u.
3. The stored data d must be still protected. This is often just assumed.

Table 6.3.: *Integrity Considering an Attacker – Dependencies*

6.1.4. Reliability

To achieve a certain reliability of a domain (c_1) (as introduced in Section 5.1.4), considering random faults and attackers (rel_{rnd} and rel_{att}) the same mechanisms as for availability with the same possible interactions and necessary conditions can be used. In case of a redundant implementation, reliability requirements instead of availability have to be considered by each of the redundant machines.

Other mechanisms to achieve reliability are described by patterns in (Hamner, 2007), e.g., *Error Correcting Audits*, *Remote Storage*, or *Fallover*. Note that some mechanisms can be used for both availability and reliability. If reliability of a certain domain is required, the behavior of the domains for input and output must also be correct. Therefore, we require the same integrity for these domains. If the integrity cannot be achieved, an appropiate action should be performed.

If reliability considering random faults is required, reliability considering attackers can be also important. If reliability considering attackers is required, reliability considering random faults can be also important.

For the reliability predicate $rel_{rnd}(\{c_1\},\{u\},P_r)$, Table 6.6 shows a set of mechanisms with possible interaction, introduced / considered domains, necessary conditions and related require-

Generic mechanism	**Possible interaction**	**Introduced / considered domains**	**Necessary conditions**	**Related**
redundancy	*conf* [1]	rm : *Redundant-Machine* m : *Machine*	$avail_{rnd}(\{m\}, \{u\}, P_{a1})$ $avail_{rnd}(\{rm\}, \{u\}, P_{a2})$ $P_{a1} \leq P_a P_{a2} \leq P_a$ [2]	*rel*
reliable hardware and software	-	m : *Machine*	$rel_{rnd}(\{m\}, \{u\}, P_a)$	rel_{att}
...	...	...	...	...

1. More targets that can be attacked.
2. But the availability of both machines together must be $\geq P_a$, even if common cause errors are considered.

Table 6.4.: *Availability Considering Random Faults – Dependencies*

Generic mechanism	**Possible interaction**	**Introduced / considered domains**	**Necessary conditions**	**Related**
service degradation, e.g., block requests from a	$avail_*(\{c_1\}, \{a\}, *)$	fw : *Firewall*	$rel_{att}(\{fw\}, \{u\}, \{a\})$ $auth_{att}(\{c_1\}, \{u\}, \{a\})$	*rel*
...	...	...	...	...

Table 6.5.: *Availability Considering an Attacker – Dependencies*

ments.

For the reliability predicate $rel_{att}(\{d\}, \{u\}, \{a\})$, Table 6.7 shows a set of mechanisms with possible interaction, introduced / considered domains, necessary conditions and related requirements.

6.1.5. Authenticity

To achieve authenticity of a domain ($auth(\{d\}, \{u\}, \{a\})$) (as introduced in Section 5.1.5), e.g., *dynamic authentication using random numbers (symmetric)*, *dynamic authentication using random numbers (asymmetric)*, *static authentication*, or *authentication using unforgeable credentials* can be used as mechanisms. All mechanisms may lead to reduced availability - intentionally for the attacker, but due to login time or if the key or the credential is lost, maybe also for the stakeholder (u). Depending on the mechanism, additional domains have to be introduced or considered. For all mechanisms, the Machine needs to be protected from modification.

For dynamic authentication using random numbers with symmetric encryption, a Key1 k_{Mchn} for the machine and a Key2 k_{Ext} for the external system (e.g., the authentic user) have to be introduced or considered. Both keys have to be protected from disclosure and modification. Additionally, the keys have to be distributed before in a secure manner (confidential and integrity-preserving). Note that the random number generator is considered to be part of the mechanism and therefore no necessary conditions about key freshness of the random number

Generic mechanism	Possible interaction	Introduced / considered domains	Necessary conditions	Related
redundancy	$conf$ [1]	$rm : RedundantMachine$ $m : Machine$ $I : \mathbb{P}\,Input$ $O : \mathbb{P}\,Outout$	$rel_{rnd}(\{m\},\{u\},P_{r1})$ $rel_{rnd}(\{rm\},\{u\},P_{r2})$ $P_{r1} \leq P_r P_{r2} \leq P_r$ $int_{rnd}(I,*,*,P_r)$ $int_{rnd}(O,*,*,P_r)$	rel_{att}
reliable hardware and software	-	$m : Machine$ $I : \mathbb{P}\,Input$ $O : \mathbb{P}\,Outout$	$rel_{rnd}(\{m\},\{u\},P_r)$ $int_{rnd}(I,*,*,P_r)$ $int_{rnd}(O,*,*,P_r)$	rel_{att}
...	...	...	...	...

1. More targets that can be attacked.

Table 6.6.: *Reliability Considering Random Faults – Dependencies*

Generic mechanism	Possible interaction	Introduced / considered domains	Necessary conditions	Related
degredated service, e.g., block requests from a	$avail_*(\{c_1\},\{a\},*)$	$fw : Firewall$	$rel_{att}(\{fw\},\{u\},\{a\})$ $auth_{att}(\{d\},\{u\},\{a\})$	rel_{rnd}
...	...	...	...	...

Table 6.7.: *Reliability Considering an Attacker – Dependencies*

generator output are given.

For dynamic authentication using random numbers with asymmetric encryption, the similar domains have to be considered. The keys are for asymmetric encryption instead of symmetric encryption. Therefore, only the integrity of the machine's key and the confidentiality of the key owned by the external system have to be preserved. For adding a machine key, confidentiality need not to be preserved. Therefore, security management is sufficient (instead of key distribution).

Instead of dynamic authentication mechanisms using random numbers, other mechanism based on key freshness can be used, e.g. authentication mechanisms using time stamps. Possible interaction, necessary conditions, introduced / considered domains, and related requirements are the same as for dynamic authentication using random numbers.

For static authentication, additionally a ConnectionDomain has to be considered. As for dynamic authentication using random numbers with symmetric encryption, all keys and the machine have to be protected from disclosure and modification. Since always the same key is transmitted to the machine (using the connection domain) it could be reused by the attacker if it is not protected from disclosure and modification.

If unforgeable credentials are used for authentication (e.g., idealized biometric mechanisms, idealized identity card), an unforgeable reference data in the machine Credential has to be considered. Its integrity has to be preserved to prevent that an attacker replaces the unforgeable reference data with its own values. If this mechanism is used, confidentiality can be relevant to protect the privacy of users.

For all mechanisms to achieve authenticity, integrity of other data may be relevant.

For the authenticity predicate $auth_{att}(\{d\},\{u\},\{a\})$, Table 6.8 shows a set of mechanisms with possible interaction, introduced / considered domains, necessary conditions and related requirements.

Generic mechanism	Possible interaction	Introduced / considered domains	Necessary conditions	Related
dynamic authentication using random numbers (symmetric)	$avail_*(\{d\},*,*)$	$m : Machine$ $k_{Mchn} : Key1$ $k_{Ext} : Key2$	$conf_{att}(\{m\},\{u\},\{a\})$ $int_{att}(\{m\},*,*,\{u\},\{a\})$ $conf_{att}(\{k_{Mchn}\},\{u\},\{a\})$ $int_{att}(\{k_{Mchn}\},*,*,\{u\},\{a\})$ $conf_{att}(\{k_{Ext}\},\{u\},\{a\})$ $int_{att}(\{k_{Ext}\},*,*,\{u\},\{a\})$ $dist_{att}(\{k_{Mchn}\},\{u\},\{a\})$ $dist_{att}(\{k_{Ext}\},\{u\},\{a\})$	int_{att}
dynamic authentication using random numbers (asymmetric)	$avail_*(\{d\},*,*)$	$m : Machine$ $k_{Mchn} : VerifyKey$ $k_{Ext} : SignKey$	$int_{att}(\{m\},*,*,\{u\},\{a\})$ $int_{att}(\{k_{Mchn}\},*,*,\{u\},\{a\})$ $conf_{att}(\{k_{Ext}\},\{u\},\{a\})$ $man_{att}(\{k_{Mchn}\},\{u\},\{a\})$ $dist_{att}(\{k_{Ext}\},\{u\},\{a\})$	int_{att}
static authentication	$avail_*(\{d\},*,*)$	$m : Machine$ $k_{Mchn} : Key1$ $k_{Ext} : Key2$ $con : ConnectionDomain$	$conf_{att}(\{m\},\{u\},\{a\})$ $int_{att}(\{m\},*,*,\{u\},\{a\})$ $conf_{att}(\{k_{Mchn}\},\{u\},\{a\})$ $int_{att}(\{k_{Mchn}\},*,*,\{u\},\{a\})$ $conf_{att}(\{k_{Ext}\},\{u\},\{a\})$ $int_{att}(\{k_{Ext}\},*,*,\{u\},\{a\})$ $conf_{att}(\{con\},\{u\},\{a\})$ $int_{att}(\{con\},*,*,\{u\},\{a\})$ $dist_{att}(\{k_{Mchn}\},\{u\},\{a\})$ $dist_{att}(\{k_{Ext}\},\{u\},\{a\})$	int_{att}
authentication using unforgeable credentials	$avail_*(\{d\},*,*)$	$m : Machine$ $cred : Credential$ $con : ConnectionDomain$	$int_{att}(\{m\},\{u\},\{a\})$ $int_{att}(\{cred\},\{u\},\{a\})$ $int_{att}(\{con\},\{u\},\{a\})$ $dist_{att}(\{cred\},\{u\},\{a\})$	int_{att} $conf$
...	...	...	...	...

Table 6.8.: *Authenticity – Dependencies*

6.1.6. Security Management

If security management (*man*) (as introduced in Section 5.1.6) is required, functional requirements about the security data to be managed have to be stated. Since the functionality should only be available for a limited group of persons, the authenticity of these persons is required and the integrity of the security data to be managed have to be preserved. Hence, for security management the same mechanisms as for authentication and integrity can be applied.

6.1.7. Secret Distribution

If secret distribution (*dist*) (as introduced in Section 5.1.7) is required, functional requirements about the secrets to be distributed have to be stated. Since the functionality should only be available for a limited group of persons and the attacker should not be able to retrieve the secret, the authenticity of these persons is required, the integrity of the secrets have to be preserved, and the confidentiality of the secrets have to be preserved. Hence, for secret distribution the same mechanisms as for authentication, integrity and confidentiality can be applied.

6.2. Working with Dependability Requirement Patterns

This section describes a process that allows to identify missing and conflicting requirements. To identify missing and conflicting requirements, the following steps can be performed:

1. select a generic mechanism
2. check possible interactions
3. consider or introduce additional domains
4. inspect the necessary conditions
    - assume necessary conditions, or
    - establish necessary conditions (continue with Step 1)
5. consider all relevant domains and interfaces (for newly identified dependability requirements, continue with Step 1)
6. extend the environment description
7. check for related requirements (for newly identified dependability requirements, continue with Step 1)
8. derive a specification (see next chapter)

For each dependability requirement stated as a predicate (see Chapter 5), we **select one generic mechanism** that solves the problem; for example, to achieve integrity (int_{att}) message authentication codes (MACs) can be used. Depending on the selected mechanism, different additional requirements have to be considered. Tables 6.1–6.8 in Section 6.1 list the first columns a set of possible mechanisms for each dependability requirement pattern. Within this pattern system we do not consider the selection of a combination of mechanisms since a combination may lead to completely different introduced domains and necessary conditions.

The tables in Section 6.1 support the analysis of conflicts between the dependability patterns. For some of the mechanisms, **possible interactions** with other dependability requirements are given (see second columns). These possible conflicts must be analyzed, and it must be determined if they are relevant for the application domain. In case they are relevant, conflicts can be resolved by modifying or prioritizing the requirements. Prioritizing requirements is not subject of this chapter. For example, if the MAC protection mechanism is applied and the specific action is to delete modified data, we may have a contradiction with the availability of that data.

For many mechanisms, additional **domains must be introduced or considered** (see 3rd columns). These domains are classes with the stereotype ≪Domain≫ or a subtype, e.g. ≪LexicalDomain≫, ≪Attacker≫, or ≪CausalDomain≫. The introduced domains should be described in the context of the problem, e.g., the reliability should be specified. We support this description with the dependability stereotypes. For example, MAC protection requires a domain Key1 k_{Snd} used to calculate the MAC and another domain Key2 k_{Rcv} used to verify the MAC.

The next step is to **inspect the necessary conditions** given in the 4th columns. The generic mechanisms usually have a set of *necessary conditions* to be fulfilled. These necessary conditions describe conditions that are necessary to establish the dependability requirement when a certain mechanism is selected. For example, the introduced keys for the MAC protection must be kept confidential, and their integrity must be preserved. Before the mechanism is applied, some other activities are necessary, e.g., a key must be distributed before it can be used for MAC calculation. The necessary condition ensures that the the key is kept confidential for the time period the requirement should the fulfilled.

Two alternatives are possible to guarantee that the necessary conditions hold: either, they can be *assumed* to hold, or they have to be *established* by instantiating a further dependability requirement pattern, that matches the necessary condition. The set of reasonable assumptions depends on the hazards to be avoided and the threats the system should be protected against. Assumptions cannot be avoided completely, because otherwise it may be impossible to achieve a dependability requirement. For example, we must assume that the user sees a warning messages on a display or keeps a password confidential. Assumptions are appropriate if the environment provides a solution, e.g., a physical protection or a certain reliability of a domain. Only in the case that necessary conditions *cannot* be assumed to hold, one must instantiate further appropriate dependability patterns, and the procedure is repeated until all necessary conditions of all applied mechanisms can be proved or assumed to hold. To avoid cyclic dependencies, the selected mechanisms should reduce the completely of the problems to be solved. In the case that necessary conditions are assumed to hold, domain knowledge have to be stated. The dependencies expressed as necessary condition are used to develop a consolidated set of dependability requirements and solution approaches that additionally cover all dependent requirements and corresponding solution approaches, some of which may not have been known initially.

We have to **consider all relevant domains and interface**. Especially, connection domains between or to domains considered in the necessary conditions have to be considered and either domain knowledge or requirements have to be specified. For these dependability requirements, we have to continue with the first step. All interfaces to the introduced or referred domains must be considered. If they cannot be assumed to fulfill the stated dependability requirement or dependency, a connection domain must be introduced. For example, the interface to the introduced key s_1 must also be kept confidential as the key itself. Since the dependencies for interfaces are always the same as for corresponding connection domains, the tables in Section 6.1 does not consider interfaces.

Additionally, we **extend the environment description** with the new domains being not part of the initial context diagram. This extended environment description is necessary to see the whole problem context.

To find additional requirements, we check the **related** column (see 5th columns). There, dependability requirements that are commonly used in combination with the described dependability pattern are mentioned. This information helps to find missing dependability requirements right at the beginning of the requirements engineering process. We add missing dependability requirements and for these requirements, we continue with the first step.

The next step in the software development process is to **derive a specification**, which describes the machine and is the starting point for its development. To specify the machine, concrete mechanisms are chosen. For example, for encryption, a software developer must select the algorithms and the key lengths according to the assumptions about the attacker. This step and all design and test steps are beyond the scope of this chapter.

6.3. CACC Case Study

The approach is illustrated on the case study introduced in Section 3.2 with the initial dependability requirements defined in Section 5.3.

6.3.1. Select Appropriate Generic Mechanisms

To establish Equation 5.1 on Page 73 (requiring integrity considering attackers), Messages Authentication Codes (MACs) can be used as a generic mechanism to check integrity and authenticity of the messages (position and speed data) from other cars with trusted CACCs.

To establish Equation 5.2 on Page 74 (requiring integrity considering random faults), mechanisms described in the standard ISO/IEC 61508, Part 2, Tables A.1 to A.15 (International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC), 2000) can be used. Here, we only consider as one example the checksum mechanism from Table 6.2 on Page 85. Therefore, our software has to check the hardware and initiate the required actions.

To establish Equation 5.3 on Page 75 (requiring availability considering random faults) and Equation 5.4 on Page 75 (requiring reliability considering random faults), reliable hardware and software can be used, because $rel_{rnd}(\{c\}, \emptyset, P) \Rightarrow avail_{rnd}(\{c\}, \emptyset, P)$.

6.3.2. Inspect Possible Interactions

A possible interaction when using MACs or checksums and deleting modified messages, is a decreased availability of the messages (position, acceleration and speed data). We decided that integrity of this data is more important than availability since the radar fall-back exists. For using reliable hardware and software no possible interactions are given in Table 6.6 on Page 88.

6.3.3. Introduce or Consider Additional Domains

Keys for sender and receiver are necessary to calculate and verify the MAC. We decide to use session keys for Sender (SessionKeySnd or SessionKeySndOC for the key in other cars) and Receiver (SessionKeyRcv). A session key has the advantage that it has a short life-time: even if the attacker is able to obtain this key, it can only be used for a short time period. For the checksum mechanism (and most of the other mechanism from International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC) (2000)), we have to regard the machine CACC as consisting of two parts: the software (CACC_SW) and the hardware (CACC_HW). For the checksum mechanism, the integrity of the Machine has to be ensured and the mechanisms to ensure that the EngineActuator_Brake is not influenced have to be reliable. If no signal to brake or accelerate is received by the EngineActuator_Brake, it continues its normal operation. Therefore, only the CACC has to be switched off (using SwitchOffHW) to ensure that the EngineActuator_Brake is not influenced. The Car is used to warn the driver acoustically and visually. This driver information is considered not to be relevant for safety, since the driver also perceives that the EngineActuator_Brake is not influenced by CACC.

For the hardware we use, a reliability of only $1 - 10^{-6}$ is guaranteed. For our software we assume (and try to achieve using several quality assurance activities, see ISO/IEC 61508 (International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC), 2000, Part 3)) a reliability of $1 - 10^{-7}$. These reliabilities can be stated with the following predicates:

$$\begin{aligned} \forall\, cacc_{HW} :\ & CACC\_HW, cacc_{SW} : CACC\_SW \bullet \\ & rel_{rnd}(\{cacc_{HW}\}, \emptyset, 1 - 10^{-6}) \wedge && (6.1) \\ & rel_{rnd}(\{cacc_{SW}\}, \emptyset, 1 - 10^{-7}) && (6.2) \end{aligned}$$

For using reliable hardware and software, the CACC has to be considered as a relevant domain.

6.3.4. Inspect Necessary Conditions

To use Messages Authentication Codes (MACs), according to Table 6.3 on Page 86, a “necessary conditions” is that confidentiality and integrity of SessionKeySnd, SessionKeySndOC, SessionKeyRcv, and the data of CACC have to be preserved. Any access by the CACCAttacker on the CACC with its session keys (SessionKeySnd, SessionKeyRcv) has to be denied and the the session keys of other cars (SessionKeySndOC) have to be kept confidential (Equations 6.3, 6.4, 6.6,and 6.5). Note that for SessionKeySnd and SessionKeySndOC confidentiality has to be ensured since other cars and the car itself can act as a sender. Another necessary condition is that the keys have to be distributed beforehand (Equation 6.7). The CACC and the OtherCarsWithCACC should be allowed to access the session keys and the CACCAttacker not. The conditions can be stated with the following predicates:

$$
\begin{aligned}
\forall\, cacc :\ & CACC, otwc : OtherCarsWithCACC, \\
c :\ & Car, eab : EngineActuator\_Brake, \\
sk_1 :\ & SessionKeySnd, sk_2 : SessionKeyRcv, \\
sk_{oc} :\ & SessionKeySndOC, \\
m :\ & Manufacturer, a : CACCAttacker \bullet \\
& conf_{att}(\{cacc\},\{m\},\{a\}) \wedge int_{att\_d}(\{cacc\},\{m\},\{a\}) \wedge && (6.3) \\
& \wedge\ conf_{att}(\{sk_1\},\{m\},\{a\}) \wedge && (6.4) \\
& int_{att}(\{sk_2\},\{\text{'inform driver'},\text{'not brake/accelerate'}\},\{c, eab\},\{m\},\{a\}) \\
& \wedge\ conf_{att}(\{sk_2\},\{m\},\{a\}) \wedge && (6.5) \\
& \wedge\ conf_{att}(\{sk_{oc}\},\{m\},\{a\}) \wedge && (6.6) \\
& dist_{att}(\{sk_1, sk_2\},\{cacc, otwc\},\{a\}) && (6.7)
\end{aligned}
$$

To use the checksum mechanism, according to Table 6.6 on Page 88, a “necessary conditions” is a sufficient integrity of the machine and a sufficient reliability of the domain used in case of a violation. In this case study, the machine is the IntegrityChecksumRandom part of the overall machine and the domain used in case of a violation is the SwitchOffHW. The conditions can be stated with the following predicates:

$$
\begin{aligned}
\forall\, cso :\ & SwitchOffHW, sw : IntegrityChecksumRandom, \\
& int_{rnd}(\{sw\},\{\text{'not brake/accelerate'}\},\{cso\},1-10^{-7}) && (6.8) \\
& rel_{rnd}(\{cso\},\emptyset,1-10^{-7}) && (6.9)
\end{aligned}
$$

We decided to fulfill all necessary conditions about confidentiality (see Equations 6.3, 6.4, 6.6 and 6.5) by physical protection. The physical protection ensures that the key values cannot be exploited by any measurement (voltage, electromagnetic radiation) and in case of an opened CACC the keys are destroyed. Therefore, we state domain knowledge about these domains. The conditions can also be modeled with the stereotype ≪DomainKnowledge≫ of our profile, as shown in the domain knowledge diagram in Fig. 6.1 on the next page.

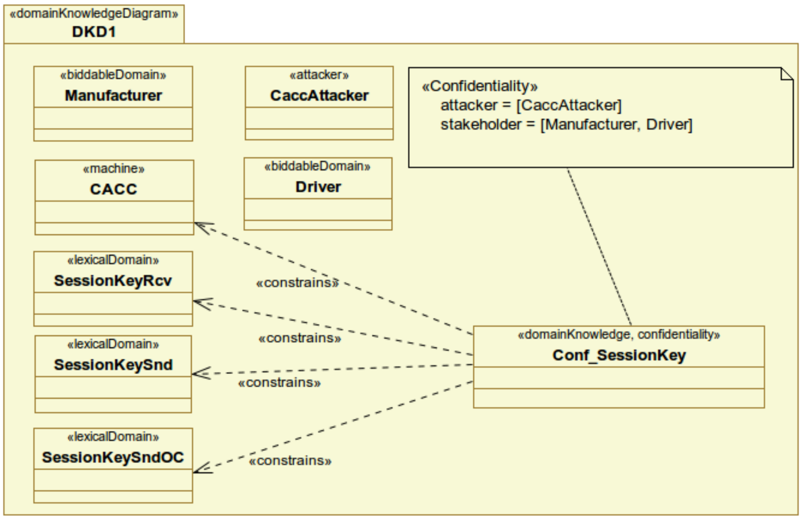


Figure 6.1.: *CACC Domain Knowledge Diagram for Confidentiality*

We decided to fulfill all necessary conditions about integrity (see Equations 6.3 on the preceding page and 6.5 on the previous page) also by physical protection. The physical protection ensures that the key values cannot be modified, e.g., by external voltage or electromagnetic radiation. Therefore, we state domain knowledge about these domains. The conditions can also be modeled with the stereotype ≪DomainKnowledge≫ of our profile, as shown in the domain knowledge diagram in Fig. 6.2.

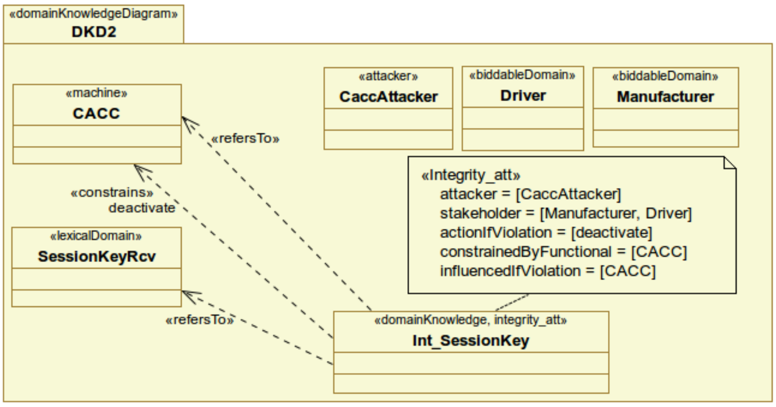
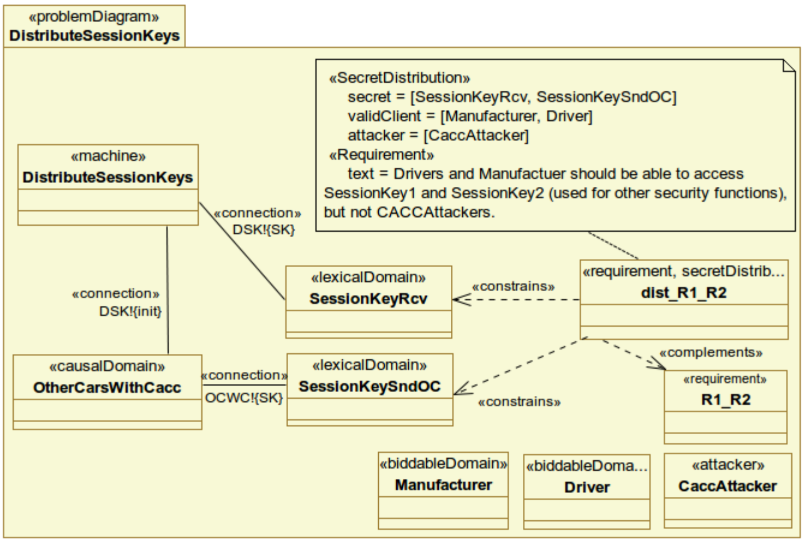


Figure 6.2.: *CACC Domain Knowledge Diagram for Integrity*

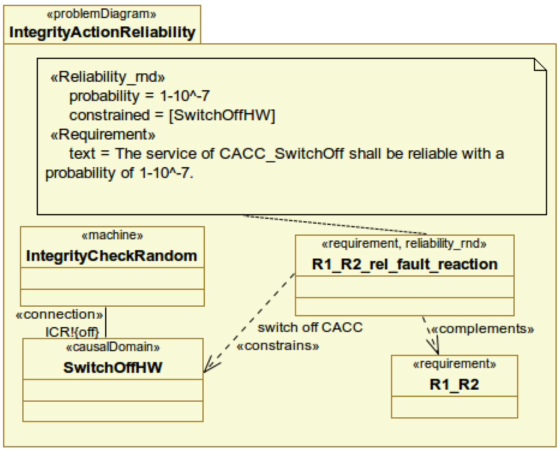
To fulfill the necessary condition express in Equation 6.7 on the previous page, a requirement for secret distribution is stated. A problem diagram describing this requirement is depicted in Fig. 6.3 on the facing page.



«SecretDistribution»
secret = [SessionKeyRcv, SessionKeySndOC]
validClient = [Manufacturer, Driver]
attacker = [CaccAttacker]
«Requirement»
text = Drivers and Manufactuer should be able to access SessionKey1 and SessionKey2 (used for other security functions), but not CACCAttackers.

Figure 6.3.: *CACC Problem Diagram for Secret Distribution*

To detect hardware faults with the the checksum mechanism and inform the driver in case of a detected fault (Equation 5.2 on Page 74) the integrity of the CACC software and the reliability of the EngineActuator_Brake and the Car (to inform the driver) has to be fulfilled. The integrity of the CACC software (see Equation 6.8 on Page 93) is fulfilled by the reliability statement expressed in Equation 6.2 on Page 92 (see Fig. 5.16 on Page 75). The reliability of the CACC_SwitchOff as a part of the CACC (see Equation 6.9 on Page 93) has to be fulfilled. The problem diagram describing the reliability requirement is depicted in Fig. 6.4.



«Reliability_rnd»
probability = 1-10^-7
constrained = [SwitchOffHW]
«Requirement»
text = The service of CACC_SwitchOff shall be reliable with a probability of 1-10^-7.

Figure 6.4.: *CACC Problem Diagram for Reliability of Domains Necessary for Fault Reaction*

6.3.5. Consider all Relevant Domains

All relevant additional domains have to be considered. The OtherCarsWithCACC acts as a connection domain between the CACC and the SessionKeySndOC. Therefore, the same dependability requirements as for the machine (Equation 6.3 on Page 93) have to also hold for the OtherCarsWithCACC:

$$\begin{aligned} \forall\, ac :\quad & OtherCarsWithCACC, m : Manufacturer, a : CACCAttacker \bullet \\ & conf_{att}(\{ac\},\{m\},\{a\}) \wedge int_{att\_d}(\{ac\},\{m\},\{a\}) \end{aligned} \tag{6.10}$$

The integrity and confidentiality of the CACC with its data, in particular the SessionKeyRcv ss_1 (required by Equations 6.3 on Page 93 and 6.4 on Page 93), can be established by some physical protection. To avoid that an attacker uses the CACC to send authentic messages to another CACC, it have to be checked that the CACC is part of a real car. This can be done by checking the signature provided by the ignition lock, and the immobilizer of the car. The corresponding dependability requirements can also be stated using the predicates of Chapter 5 on Page 53.[1] The SessionKeySndOC ss_2 is stored in the OtherCarsWithCACC. Its confidentiality (Equation 6.5 on Page 93) and the interface to this key are also established by physical protection. For the interface to the other cars WIFI_WAFE integrity is required. It is achieved in the same way as for Equation 5.1 on Page 73 with MAC protection.

Additionally, the OtherCarsWithCACC has to calculate the signature using the session key to sign its position and speed. The same calculation as to be performed by the CACC to provide speed and position to the following car. Therefore, the subproblem with the machine SendPosSpeed is complemented by the subproblem SendPosSpeedWithSignature depicted in Fig. 6.5.

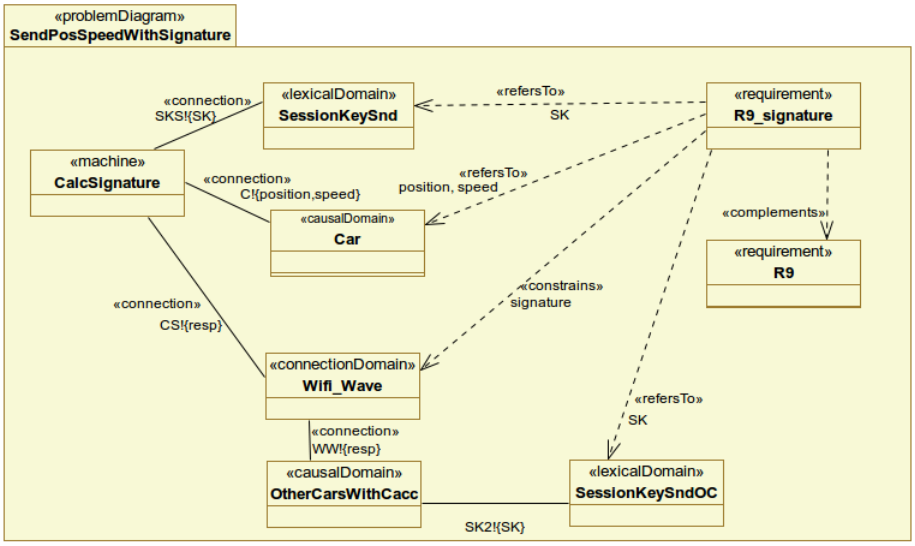


Figure 6.5.: *CACC Problem Diagram for Signature Generation*

6.3.6. Select Appropriate Generic Mechanisms (to Fulfill Conditions)

To establish Equation 6.7 on Page 93, a dynamic authentication mechanism with random numbers can be used. With this authentication mechanism additionally a session key can be generated. Since replay attacks cannot be avoided in the described context, random numbers are used for authentication (cf. *CSPF Dynamic Authentication* in (Hatebur et al., 2006)).

To fulfill the reliability requirement for the SwitchOffHW, reliable hardware can be used.

6.3.7. Introduce or Consider Additional Domains (to Fulfill Conditions)

Table 6.3 on Page 86 shows that two keys are necessary for the MAC mechanism (Keys s_{Mchn} and s_{Ext}). For our concrete problem the keys are of type AuthKey1 or AuthKey2. The mechanism is applied as follows: The random number is sent to the car ahead. The CACC of this car encrypts this random number with its AuthKey1 ak_1 and sends it back. Our CACC checks if the returned number is valid using its AuthKey2 ak_2. Only if the returned number is valid, the

[1] Left out to avoid repetitions.

speed and acceleration from the car ahead is used to calculate the acceleration or deceleration. The returned number is used as session keys sk_1 and sk_2.

6.3.8. Inspect Necessary Conditions (to Fulfill Conditions)

Table 6.8 on Page 89 shows for the dynamic authentication mechanism that integrity and confidentiality of the Machine (see Equations 6.12 and 6.11), as well as of AuthKey1 and AuthKey2 (see Equations 6.13 - 6.16) have to be preserved. Additionally, AuthKey1 and AuthKey2 have to be distributed beforehand (see Equation 6.17).

$$
\begin{aligned}
\forall\, cacc :\quad & CACC, ac : ak_1 : AuthKey1, ak_2 : AuthKey2,\\
m :\quad & Manufacturer, a : CACCAttacker \bullet\\
& conf_{att}(\{cacc\}, \{m\}, \{a\}) \wedge && (6.11)\\
& int_{att\_d}(\{cacc\}, \{m\}, \{a\}) \wedge && (6.12)\\
& conf_{att}(\{ak_1\}, \{m\}, \{a\}) \wedge && (6.13)\\
& int_{att\_d}(\{ak_1\}, \{m\}, \{a\}) \wedge && (6.14)\\
& conf_{att}(\{ak_2\}, \{m\}, \{a\}) \wedge && (6.15)\\
& int_{att\_d}(\{ak_2\}, \{cacc\}, \{a\}) \wedge && (6.16)\\
& dist_{att\_d}(\{ak_1\}, \{cacc\}, \{a\}) \wedge dist_{att}(\{ak_2\}, \{cacc\}, \{a\}) && (6.17)
\end{aligned}
$$

AuthKey1 is stored in the CACC and AuthKey2 in OtherCarsWithCACC. The integrity and the confidentiality of the CACC and its data (required by Equations 6.11 - 6.16) can be established in the same way as described above. Secure distribution of the authentication keys (Equation 6.17) is assumed to be done in the production environment of the CACC.

6.3.9. Consider all Relevant Domains (to Fulfill Conditions)

All relevant additional domains have to be considered. OtherCarsWithCACC acts as a connection domain between the CACC and SessionKeySndOC. Therefore, the same dependability requirements as for the machine (Equation 6.3 on Page 93) have to hold (see Equation 6.10 on Page 95).

6.3.10. Extended Environment Description

The new context diagram for the CACC resulting from applying dependability requirements patterns is shown in Fig. 6.6 on the following page. New domains were **added to the description of the environment**.

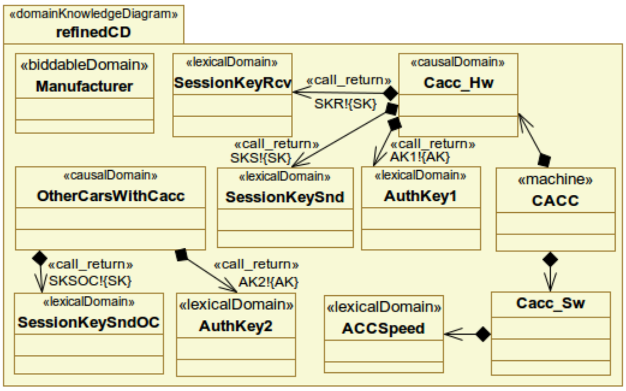


Figure 6.6.: *Refined CACC Context Diagram (Domain Knowledge Diagram)*

Additionally, the machine CACC is split into CACC_HW and CACC_SW. Since some dependability requirements state that the Driver have to be informed, the additional phenomenon warn_driver is introduced.

6.4. Related Work

We are not aware of any similar approach for modeling a wide range of dependability requirements. However, the Common Criteria (International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC), 2009a), Part 2 defines a large set of so-called *Security Functional Requirements (SFRs)* with explicitly given dependencies between these SFRs. For example, the SFR ***c**ryptographic **op**eration* in the class ***f**unctional requirements for **c**ryptographic **s**upport* (FCS_COP) has a dependency to ***c**ryptographic **k**ey **m**anagement* in the class ***f**unctional requirements for **c**ryptographic **s**upport* (FCS_CKM). The SFRs in the Common Criteria are limited to security issues. The dependencies given in the Common Criteria are re-used for our pattern system. Our dependability requirements can be regarded on the level of Security Objectives that have to be stated according to Common Criteria, Part 3, before suitable SFRs are selected.

Røstad et al. (2006) present an initial set of possible conflicts between safety and security requirements. The interactions described in this chapter are based on these initial possible conflicts.

6.5. Conclusions and Future Work

In this chapter, we have described a pattern system that can be used to identify missing requirements in a systematic way. The pattern system is based on the predicates used to express the requirements. The parameters of the predicates refer to domains of the environment descriptions and are used to describe the dependencies precisely. The pattern system may also show possible conflicts between dependability requirements in an early requirements engineering phase.

The pattern system is not complete, but can be easily extended with additional requirements and mechanisms. Nevertheless, a huge set of real-live problems can be expressed and analyzed using this pattern system.

In summary, our pattern system has the following advantages:

- The patterns help to structure and classify the dependability requirements. For example, requirements considering integrity can be easily distinguished from availability requirements. It is also possible to trace all dependability requirements that refer to one domain.

- The predicates are the basis of a modular construction system used to identify dependencies and possible interactions with other dependability requirements.

In the future, we plan to systematically search for additional dependability requirements and dependencies using existing specifications (e.g., public Security Targets). It is interesting to develop a method for prioritizing the requirements in order to resolve conflicts. Additionally, our tool can be extended to support interactive identification of missing and interacting requirements.

Chapter 7

Development of Specifications for Dependable Systems

The specification of the machine describes the behavior at its external interfaces, while the requirements refer to relevant domains in the environment (cf. Jackson (2001)). The specification can be derived from the requirements using domain knowledge. Specifications are implementable requirements. Requirements that are not implementable are transformed into specifications using domain knowledge and assumptions. For an example, see (Jackson & Zave, 1995). It must be shown that, when the machine fulfills S, then the requirements are satisfied. For that proof, domain knowledge and assumptions can be used in the validation condition $D \wedge A \wedge S \implies R$. $D \wedge A \wedge S$ must be consistent, otherwise everything can be deduced. Several means of expressing the specification are used in ADIT and are described in the appendix:

- Sequence Diagrams (Step A3, see Appendix A.3)
- Description of pre- and postconditions (Step A5, see Appendix A.5)
- Lifecycle expressions (A6, see Appendix A.6)

This chapter describes on one hand a method for deriving the specification from the functional requirements more systematically and on the other hand a specialized approach for deriving specifications from dependability requirements that can be transformed into functional requirements, especially security requirements.

The specifications are developed systematically based on given problem diagrams. When the specifications have been derived systematically from functional requirements, the specifications directly correspond to the problem diagrams, e.g., relevant domains are represented as lifelines and the phenomena correspond to the messages. These conditions can be expressed with OCL and validated automatically. This conditions are based on joined work with Côté et al. (2008).

The approach to derive a specification for dependability requirements expressed using problem diagrams is based on Hatebur, Heisel, Jürjens, and Schmidt (2011) and Hatebur et al. (2006).

When building *dependable systems*, it is instrumental to take *dependability requirements* into account right from the beginning of the development process to reach the best possible match between the expressed requirements and the developed software product, and to eliminate any source of error as early as possible. Knowing that building dependable systems is a highly sensitive process, it is important to accomplish the transition from dependability requirements to specifications *correctly*, i.e., without introducing vulnerabilities.

For dependability requirements considering random faults, usually the interface specifications can be derived from the functional requirements. For systems with requirements considering random faults, there are usually no dedicated protocols at the external interface of the machine. For these systems, dedicated protocols are used within the machine to be developed, e.g., protocols for handling faults with redundant machines or protocols handling a watchdog.

For dependability requirements considering an attacker, the some dependability requirements (i.e., security requirements) themselves have to be transformed into functional specifications – especially if external systems have to exchange data with the machine. To express specifications for systems with security requirements, UMLsec (Jürjens, 2005) can be used. Another alternative notation not chosen for this chapter is described by Lodderstedt, Basin, and Doser (2002). This notation can be used for specifying access control policies for actions on protected resources.

The different security requirements analysis (see Fabian et al. (2010) for an overview) and secure specification methods are mostly not integrated with each other. In particular, for existing approaches bridging the gap between dependability requirements analysis and specifications only provide informal guidelines for the transition from dependability requirements to specifications. Carrying out the transition manually according to these guidelines is highly non-trivial and error-prone, which leaves the risk of inadvertently introducing vulnerabilities. Ultimately, this would lead to the dependability requirements not being enforced in the specifications (and later its implementation).

We present a method to systematically develop structural and behavioral specifications based on security requirements. We use the security requirements analysis method presented in Chapters 4, 5, and 6. to capture, structure, and analyze security requirements. We extend this approach by a detailed procedure for developing *UMLsec* (Jürjens, 2005) design models from previously captured and analyzed security requirements. Our method is supported by *model generation rules* expressed as pre- and postconditions using the formal specification language *OCL* (Object Constraint Language) (UML Revision Task Force, 2010a). Since our rules are specified in a formal and evaluatable way, the implementation of this tool can be checked automatically for correctness with respect to the model generation rules. Consequently, applying our method to generate UMLsec specifications supported by our tool and based on previously captured and analyzed security requirements becomes systematic, less error-prone, and a more routine engineering activity. We illustrate our method by the example of the already introduced CACC case study.

The chapter is organized as follows: Section 7.1 describes how to derive and verify specifications for functional requirements. We show in Section 7.2 how sequence diagrams can be used to express specifications and functional requirements. Section 7.3 presents an approach to systematically develop UMLsec specifications based on previously captured and analyzed security requirements. We consider related work in Section 7.4. In Section 7.5, we give a summary and directions for future research.

7.1. Derive and Verify Specification for Functional Requirements

Specifications are implementable requirements. They are requirements expressed using phenomena of the machine interface. Requirements are not implementable, if they

- constrain phenomena that are controlled by the environment, e.g., the lift is not to be overloaded.
- refer to phenomena that are not observable by the machine, e.g., the lift should go to a floor where people are waiting.
- make constraints for the future, e.g., as soon as a user has dialed the last digit, he receives the dial tone, the busy signal, or the announcement "number not assigned".

To derive the specification in most cases, references in the requirements to domains or phenomena that are not controlled or observed by the machine are replaced by phenomena that are controlled or observed by the machine according to given or obtained domain knowledge. In the CACC case study, the requirement

R1 The CACC should accelerate the car if the desired speed is higher than the current speed, the CACC is activated and the measured distance and the calculated distance to the car(s) ahead is safe.

constrain the phenomenon accelerate that is controlled by the EngineActuator and it refers to the phenomenon distance are not observable by the CACC. The requirement can be transformed into a specification using the following domain knowledge (facts or assumptions):

F1 The car accelerates when the accelerate value provided to the EngineActuator_Brake increase and the brake value is zero (or quite small).

F2 The distance is either measured with the radar sensor integrated in the CACC, or calculated from the positions of the car itself (via GPS receiver integrated in the car) and the car ahead (transmitted via Wifi_Wave).

The derived specification is the following:[1]

S1 The CACC should increase the accelerate value for the EngineActuator_Brake if the desired speed is higher than the current speed, the CACC is activated and the measured distance to the car ahead is safe and the positions provided by the car itself and provided via Wifi_Wave show that the distance is safe.

The other specifications derived in this way are:

S2 The CACC should increase the brake value for the EngineActuator_Brake if the desired speed is much (30 km/h) lower as the current speed provided via CAN, the CACC is activated and measured distance to the car ahead is decreasing towards the safe limit or the positions provided by the car itself and provided via Wifi_Wave show that the distance to the car ahead is decreasing towards the safe limit.

S3 When the CAN bus indicates that the brake pedal or the deactivate CACC button is pressed, the CACC is deactivated and last_speed is set to desired_speed.

S4 When the CAN bus indicates that resume is pressed (and resume_speed exists), the CACC is activated and desired_speed is set to last_speed.

S5 When the CAN bus indicates that increase_speed is pressed and CACC is activated, the desired_speed is increased by 5 km/h (max.: 200 km/h).

S6 When the CAN bus indicates that decrease_speed is pressed and CACC is activated, desired_speed is decreased by 5 km/h (min: 30 km/h).

S7 When the CAN bus indicates that set_speed is pressed, desired_speed is set to current_speed.

S8 At that point of time when the CACC is deactivated, a warning message and a new CACC status is sent on the CAN bus in order to inform the driver the he has to take control over accelerating and braking. The desired_speed and the CACC_state (CACC is activated or not) should be send to Car when the CACC is powered in order to display this information.

S9 The position and speed received via CAN should be sent via Wifi_Wave to OtherCarsWithCACC.

[1] To simplify the case study, the value changes for accelerating and braking are not specified. Compared to (Hatebur & Heisel, 2009b), requirements are detailed.

7.2. Express Specifications and Functional Requirements

Requirement and Specifications can be expressed using sequence diagrams. Whereas problem diagrams are used to describe static aspects of the machine we are going to build, sequence diagrams cover the dynamic aspects of the machine. In this section we show how sequence diagrams can be checked to be consistent with the problem diagrams we created in a previous step. We developed the following checks:

- Problem diagram domains vs. lifelines in sequence diagrams
- Lifelines in sequence diagrams vs. problem Diagram domains
- Sent messages in sequence diagrams vs. operations in controlled interfaces in problem diagrams
- Received messages in sequence diagrams vs. operations in observed interfaces in problem diagrams

For each problem diagram, we create a set of sequence diagrams (normal cases and exceptional behavior). In each sequence diagram, we draw a lifeline for the machine and all domains in the corresponding problem diagram. The consistency between problem diagram domains and lifelines in the sequence diagram can be checked on an EMF model with the following OCL expression (see Listing 7.1): for all sequence diagrams (Interaction in EMF) *sd* (line 1), check in the set of all packages with the stereotype ≪ProblemDiagram≫ (line 2) if there exists a package where the names of the sequence diagram lifelines (line 8) are included in the set of the names of the package elements with the stereotype ≪Domain≫ or a subtype of ≪Domain≫ (lines 3-7). Additional sequence diagrams for later phases may be included into the EMF model. These sequence diagrams have no corresponding problem diagram. To avoid errors caused by these sequence diagrams, interactions between classes that compose another class are allowed (lines 10-29).

```
1  Interaction.allInstances() ->forAll( sd |
2    Package.allInstances()
         ->select(getAppliedStereotypes().name->includes(’ProblemDiagram’))
3    ->exists(clientDependency.target ->select(oclIsTypeOf(Class))
4      ->select(getAppliedStereotypes().name ->includes(’Domain’) or
5        getAppliedStereotypes().general.name ->includes(’Domain’) or
6        getAppliedStereotypes().general.general.name ->includes(’Domain’) or
7        getAppliedStereotypes().general.general.general.name ->includes(’Domain’))
8        .oclAsType(Class).name ->includesAll(sd.oclAsType(Interaction).lifeline.name)
9    )
10   or
11   ( sd.oclAsType(Interaction).lifeline.name->forAll(ln|
12         Class.allInstances()->exists(c |
13           let names_of_included_classes:Set(String) =
14             c.member->select(oclIsTypeOf(Property)).oclAsType(Property).type
                   ->select(oclIsTypeOf(Class)).oclAsType(Class).name->asSet()
15           in
16           let ln_ss:Sequence(String) = Sequence{1..ln.size()}
                 ->collect(i|ln.substring(i,i))
17           in
18           let class_name: String =
19             if ln_ss->indexOf(’:’) = null
20             then ln
21             else Sequence{(ln_ss->indexOf(’:’) + 1)..ln_ss->size()} ->iterate(i;
                   res:String=’’| res.concat(ln_ss->at(i)))
22             endif
23           in
```

```
            names_of_included_classes->includes(class_name)
        )
        or ln = 'ENVIRONMENT'
    )
  )
)
```

Listing 7.1: *Lifelines in sequence diagrams vs. problem Diagram domains*

We have to describe the behavior for each problem diagram with at least one sequence diagram. The OCL expression in Listing 7.2 checks this condition: when the first interaction has be created (line 1), for all packages with the stereotype ≪ProblemDiagram≫ *pd* (line 2), check in the set of all sequence diagrams (Interaction in EMF) (line 3) if there exists a sequence diagram where the names of the sequence diagram lifelines (line 9) are included in the set of the names of the package elements with the stereotype domain or a subtype of domain (lines 4-8). Additionally, all domains in the package connected with the machine (lines 11-13) have to be included as lifelines in the sequence diagram (line 14).

```
1  Interaction.allInstances()->size()>0 implies
2  Package.allInstances()->select(getAppliedStereotypes().name
       ->includes('ProblemDiagram')) ->forAll( pd |
3    Interaction.allInstances()
4    ->exists(pd.oclAsType(Package).clientDependency.target ->select(oclIsTypeOf(Class))
5          ->select(getAppliedStereotypes().name ->includes('Domain') or
6                  getAppliedStereotypes().general.name ->includes('Domain') or
7                  getAppliedStereotypes().general.general.name ->includes('Domain') or
8                  getAppliedStereotypes().general.general.general.name
                       ->includes('Domain'))
9          .oclAsType(Class).name->includesAll(lifeline.name )
10         and
11         pd.oclAsType(Package).clientDependency.target ->select(oclIsTypeOf(Class))
12         ->select(d | clientDependency.target
               ->select(oclIsTypeOf(Association)).oclAsType(Association)
13         ->exists(endType->includes(d.oclAsType(Type)) and
               endType.getAppliedStereotypes().name->includes('Machine')))
14         ->forAll(d | lifeline.name ->includes(d.oclAsType(Class).name))
15   ) )
```

Listing 7.2: *Problem diagram domains vs. lifelines in sequence diagrams*

The messages in the diagram directly correspond to the operations in the interfaces. All operations in observed interfaces of problem diagram domains must occur as messages in a sequence diagram. These messages must point to the lifeline that corresponds to the observing domain. This condition is expressed in Listing 7.3. In this expression, mchns is set to be the set of machines being in a problem diagram (lines 1-7), doms is the set of machines mchns together with all connected classes with the stereotype ≪DisplayDomain≫, ≪ConnectionDomain≫, or a subtype (lines 8-23), lexsmsgs is set to be the set of all messages controlled by lexical domains (lines 45-51), and obphen is set to be the set of all operations in interfaces observed by doms (line 54) and being part of a problem diagram (lines 55-57). The expression checks that all relevant controlled phenomena in the problem diagrams at the machine interface are sent messages in the sequence diagrams (lines 24-42). Relevant are all phenomena (interface operations) of interfaces controlled by doms (line 25), being part of a problem diagram (lines 26-28), not controlled by a lexical domain (lines 29-36). The expression also checks that the relevant sent messages in the sequence diagrams are controlled phenomena in the problem diagram (lines 62-70). Relevant are all messages sent by lifelines that correspond to an element of doms (lines 62-66) and are not in the set of messages controlled by lexical domains lexsmsgs.

```
let mchns: Set(Class) =
  Package.allInstances()->select(getAppliedStereotypes().name
      ->includes(’ProblemDiagram’))
  .clientDependency ->select(getAppliedStereotypes().name ->includes(’isPart’))
  .target ->select(getAppliedStereotypes().name ->includes(’Machine’) or
                  getAppliedStereotypes().general.name ->includes(’Machine’))
  .oclAsType(Class)->asSet()
in let doms: Set(Class) =
  Package.allInstances()->select(getAppliedStereotypes().name
      ->includes(’ProblemDiagram’))
  .clientDependency ->select(getAppliedStereotypes().name ->includes(’isPart’))
  .target ->select(getAppliedStereotypes().name ->includes(’DisplayDomain’) or
                  getAppliedStereotypes().general.name ->includes(’DisplayDomain’) or
                  getAppliedStereotypes().name ->includes(’ConnectionDomain’) or
                  getAppliedStereotypes().general.name ->includes(’ConnectionDomain’))
  ->select(cddd |
    Association.allInstances()
    ->exists(as |
        as.oclAsType(Association).endType ->includes(cddd.oclAsType(Class)) and
        as.oclAsType(Association).endType ->exists(et | mchns
            ->includes(et.oclAsType(Class)))
    )
  )
  ->union(mchns)
  .oclAsType(Class)->asSet()
in
  doms.clientDependency ->select(getAppliedStereotypes().name
      ->includes(’controls’)).target
        ->intersection(Package.allInstances()->select(getAppliedStereotypes().name
            ->includes(’ProblemDiagram’))
                .clientDependency ->select(getAppliedStereotypes().name
                    ->includes(’isPart’))
                .target ->select(oclIsTypeOf(Interface)))
        .ownedElement
        ->select(oclIsTypeOf(Operation)).oclAsType(Operation).name ->asSet()
        ->reject(op |
          Class.allInstances()->select(getAppliedStereotypes().name
              ->includes(’LexicalDomain’))
          .clientDependency ->select(getAppliedStereotypes().name
              ->includes(’controls’)).target.oclAsType(Interface).ownedElement
          ->select(oclIsTypeOf(Operation)).oclAsType(Operation).name
          ->includes( op )
        )
        ->forAll( phen |
                Lifeline.allInstances()
                .coveredBy
                ->select(oclIsTypeOf(MessageOccurrenceSpecification) and
                    name.substring(1,1)=’S’)
                .oclAsType(MessageOccurrenceSpecification).message->select(m | m <>
                    null).name
                ->includes(phen)
        )

and
let lexrmsgs: Set(String) =
  Lifeline.allInstances()
  ->select(ln | Class.allInstances()->select(getAppliedStereotypes().name
      ->includes(’LexicalDomain’))->exists(name=ln.name))
  .coveredBy
  ->select(oclIsTypeOf(MessageOccurrenceSpecification) and name.substring(1,1)=’R’)
  .oclAsType(MessageOccurrenceSpecification).message.name
  .oclAsType(String)->asSet()
in
let contrphen: Set(String) =
   doms.clientDependency ->select(getAppliedStereotypes().name
       ->includes(’controls’)).target
             ->intersection(Package.allInstances()->select(getAppliedStereotypes().name
                 ->includes(’ProblemDiagram’))

            .clientDependency ->select(getAppliedStereotypes().name
                ->includes(’isPart’))
        .target ->select(oclIsTypeOf(Interface))).ownedElement
        ->select(oclIsTypeOf(Operation)).oclAsType(Operation).name
       .oclAsType(String)->asSet()
in
  Lifeline.allInstances()
  ->reject(ln | Class.allInstances()->select(getAppliedStereotypes().name
      ->includes('LexicalDomain'))->exists(name=ln.name))
  ->select(ln | doms ->exists(name=ln.name))
  .coveredBy
  ->select(oclIsTypeOf(MessageOccurrenceSpecification) and name.substring(1,1)='S')
  .oclAsType(MessageOccurrenceSpecification).message.name
  ->reject(n | lexrmsgs ->includes(n))
  ->select(n | n<>'')
  ->forAll(msg | contrphen ->includes(msg)  )
```

Listing 7.3: *Sent messages in sequence diagrams vs. operations in controlled interfaces in problem diagrams*

The above-mentioned condition must also be valid for the opposite direction: all operations in controlled interfaces of problem diagram domains must occur in a sequence diagram. They must come from the lifeline that corresponds to the using or controlling domain. A similar OCL expression can be used. The adaptations to be made are to exchange observes and controls, as well as 'R' (receives) and 'S' (sends) (see Appendix C, Listing C.49).

Requirements R1 and R2 can be expressed by the sequence diagram in Fig. 7.1 on the next page. In a loop, the machine receives the position and speed of the car itself, the position of other cars with CACC, the desired speed and activation state of the CACC, and the measured distance to the car ahead. Depending on these information the commands accelerate or brake may be sent to the car. To express Specifications S1 and S2 exactly, a lifeline for the car has to be replaced by a lifeline for the engine actuator and a lifeline for the brake.

Requirements R3 – R7 can be expressed by the sequence diagram in Fig. 7.2 on Page 109. The Specifications S3 – S7 can be expressed in the same way. In a loop, the machine receives the desired speed, the current speed and may be also the last desired speed. The driver can perform different actions, expressed by alternatives. Depending on the requested action of the driver, the specified activities are performed. Some of these activities are only performed if the given condition is true.

Requirement R8 and also Specification S8 can be expressed by the sequence diagram in Fig. 7.3 on Page 110. This diagram shows that the desired speed and the state of the CACC is displayed continuously. If the CACC is deactivated, a warning message is sent to the driver. To display the desired speed and the state, the CACC has to sent CAN messages to the car.

Specification S9 can be expressed by the sequence diagram in Fig. 7.4 on Page 110. It shows that position and speed are forwarded to other cars via Wifi_Wafe. The order of position and speed is not relevant. Checking the constraints reveals that all expressions checking the consistency between problem diagrams and sequence diagrams (Listings 7.2, 7.1, and 7.3, as well as the expression in Appendix C, Listing C.49) are satisfied.

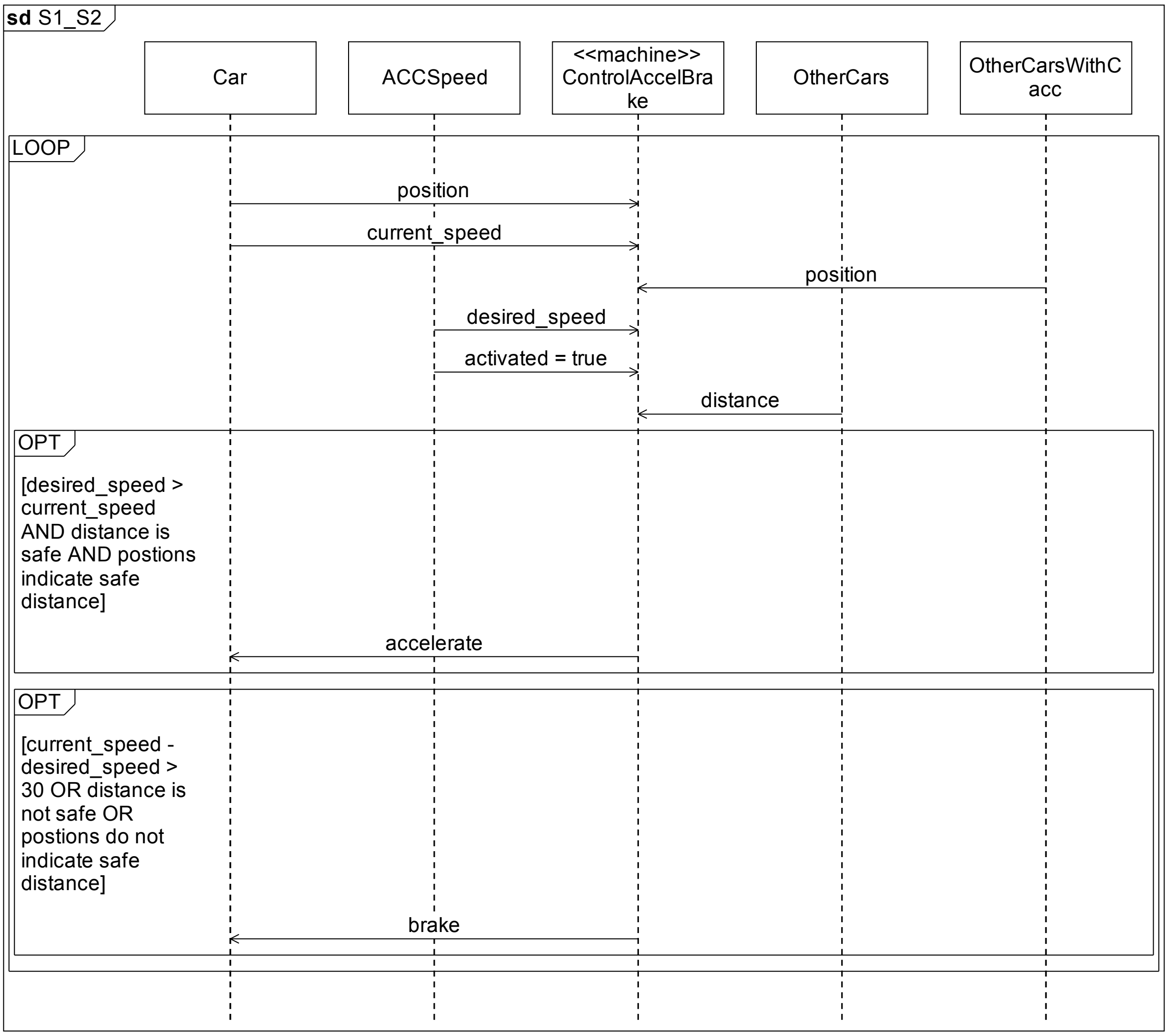

sd S1_S2
Car
ACCSpeed
<<machine>> ControlAccelBrake
OtherCars
OtherCarsWithCacc
LOOP
position
current_speed
position
desired_speed
activated = true
distance
OPT
[desired_speed > current_speed AND distance is safe AND postions indicate safe distance]
accelerate
OPT
[current_speed - desired_speed > 30 OR distance is not safe OR postions do not indicate safe distance]
brake

Figure 7.1.: *Sequence Diagram: CACC Specifications S1 and S2*

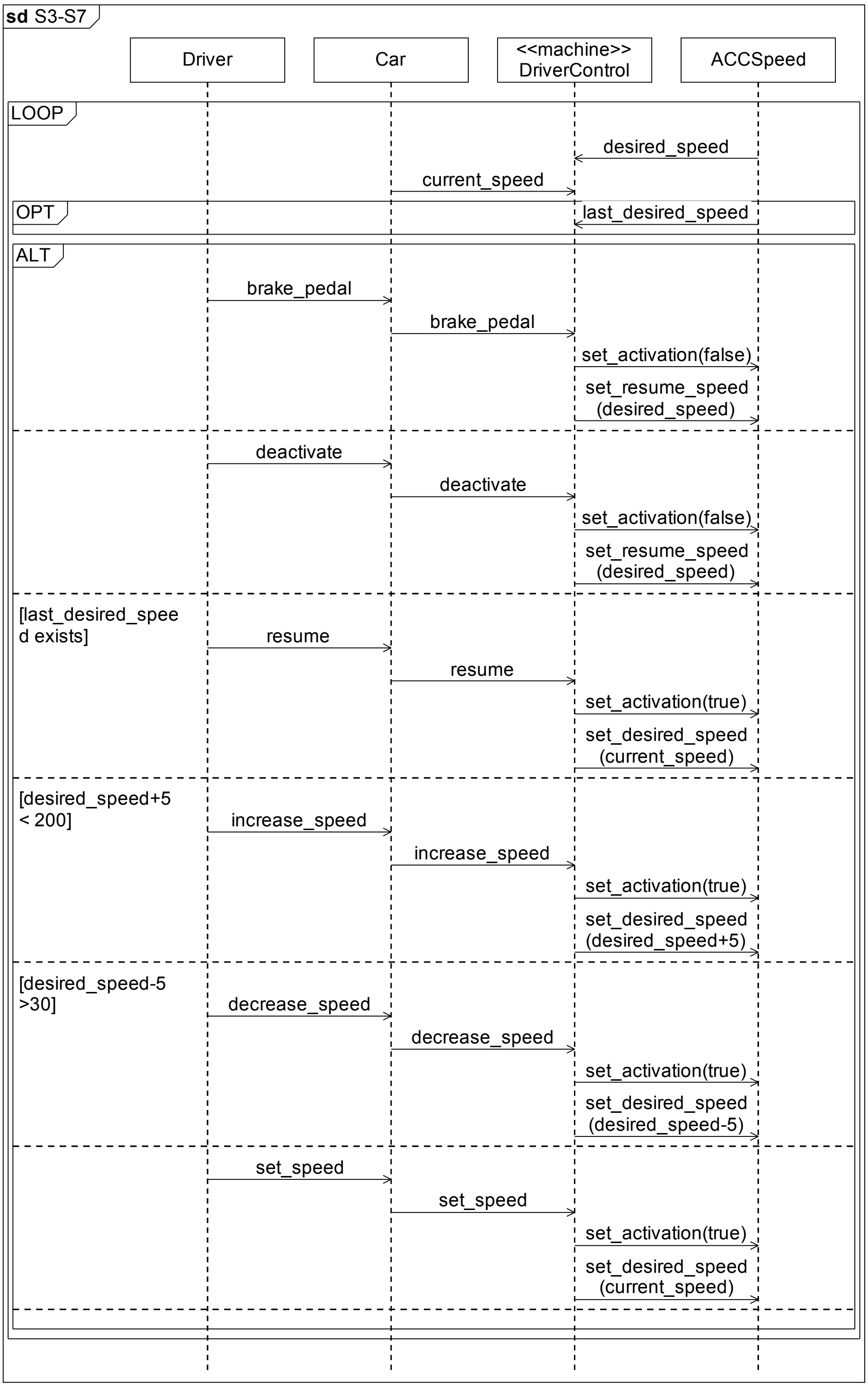


Figure 7.2.: *Sequence Diagram: CACC Specifications S3 – S7*

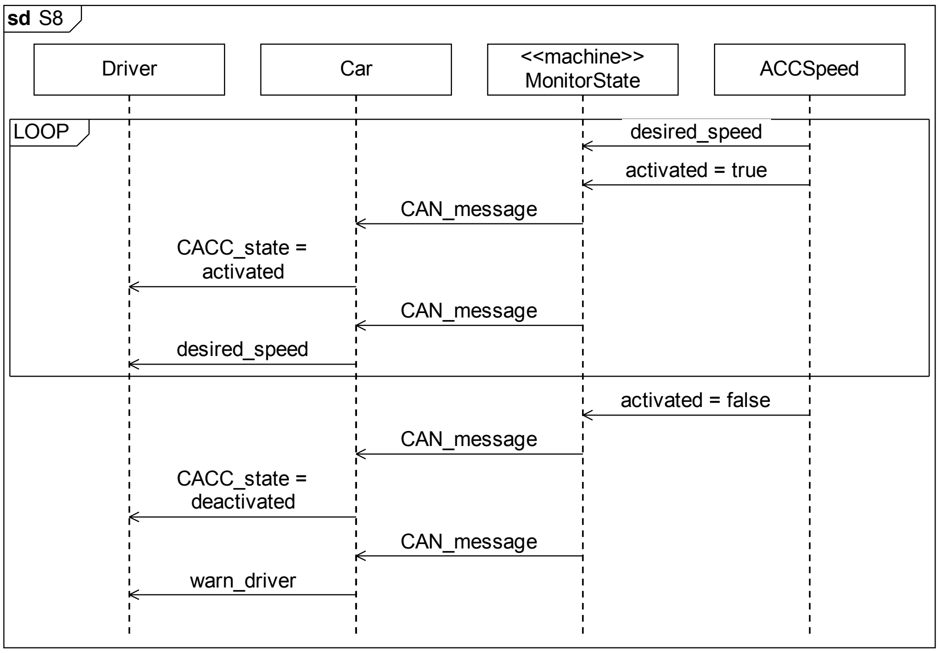


Figure 7.3.: *Sequence Diagram: CACC Specification S8*

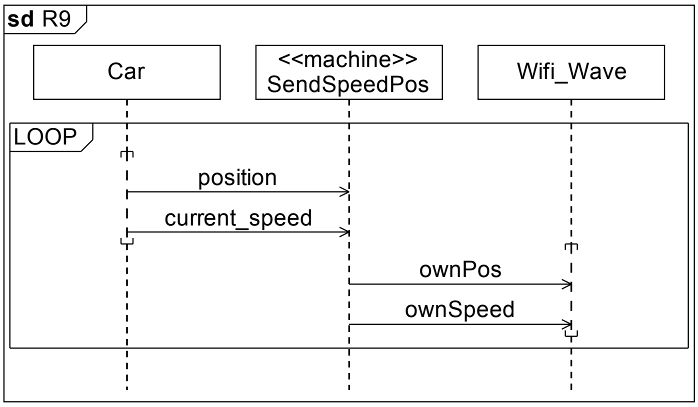


Figure 7.4.: *Sequence Diagram: CACC Specification S9*

7.3. From Security Requirements to UMLsec Specifications

In Sections 7.1 and 7.2, we showed how to handle functional requirements. In this section, we connect the security requirements engineering approach presented in Chapters 4 – 6 with secure specification based on UMLsec (cf. (Hatebur et al., 2011)). We first present a procedure to generate UMLsec diagrams describing the environment in Section 7.3.1. Second, we introduce a procedure to generate UMLsec diagrams describing security mechanisms in Section 7.3.2. We finally present in Section 7.3.3 work in progress on the construction of a tool that realizes the aforementioned procedures to develop UMLsec specification models based on security requirements. In contrast to the tool described in Sections 4.2 that can also be used for the validation of the conditions in Sections 7.2, 5.1, and 8.2, this tool is used to generate specifications.

7.3.1. UMLsec Deployment Diagrams for Environment Descriptions

According to our security requirements engineering approach as illustrated in Chapters 4 – 6, describing the operational environment of a secure software system is of great importance. In fact, the environment description is also necessary for secure specification: security-critical design decisions should lead to the fulfillment of the security requirements in the given environment. However, in a different environment, the same design decisions might lead to an insecure system.

In the following, we present a procedure to develop deployment diagrams enriched with UMLsec elements from context diagrams and security requirements. A deployment diagram can be used by the UMLsec tool to perform certain checks. For each step, an operation name with parameters is provided. These operations represent *model generation rules*.

1. Create a UML package named adequately that contains a deployment diagram *(it is required that such a diagram does not yet exist and that exactly one context diagram exists)*.
   `createDeploymentDiagram(diagramName: String)`

2. Add the ≪secure links≫ stereotype to the package and assign a certain type of attacker (e.g., *default* or *insider* as described in (Jürjens, 2005, Chapter 4.1)) to the adversary tag. Decide which attacker type is appropriate based on threats modeled in the context diagram and domain knowledge collected during security requirements engineering. For example, *default* attackers cannot execute attacks in a LAN environment, but *insider* attacker can. Hence, if the context diagram describes an attack in a LAN environment, the attacker is of type *insider*.
   `addSecureLinksStereotype(inDiagram: String, adv: String)`

3. Each domain contained in the context diagram *(it is required that exactly one context diagram exists and that the deployment diagram exists)* that is not a connection or biddable domain (and not an attacker) is represented as a node in the deployment diagram.
   `createNodes(inDiagram: String)`

4. Moreover, each domain that is part of another domain in the context diagram is represented either as a nested node or a nested class.
   `createNestedNodes(domainNames: String[])` or `createNestedClasses(domainNames: String[])`

5. Each connection between the aforementioned domains is represented as a communication path and a dependency:
   a) We create a communication path stereotyped according to the communication type as described in Tab. 7.1. Note that only one of the UMLsec stereotypes is allowed for

Context Diagram	*UMLsec Deployment Diagram*
≪physical≫	≪wire≫ (physical protection against default adversary is assumed)
≪ui≫	not considered since biddable domains are not part of deployment diagrams
≪remote_call≫	see ≪network_connection≫
≪network_connection≫	≪Internet≫, ≪LAN≫, ≪encrypted≫ depending on the domain knowledge collected during security requirements engineering

Table 7.1.: *From Context Diagrams to UMLsec Deployment Diagrams*

each communication path. Moreover, the defined mapping for context diagram stereotypes also applies to sub-stereotypes. For example, ≪wireless≫ is a sub-stereotype of ≪network_connection≫, and therefore, ≪wireless≫ can be mapped to ≪Internet≫, ≪LAN≫, and ≪encrypted≫, too.

We create communication paths for all associations between the aforementioned domains of type ≪physical≫, and we also associate a communication type. For these associations no decision is necessary (`createCommunicationPaths (inDiagram: String)`). For all network connections (retrievable with `getNetworkConnections(): String[]`), the developer has to choose between ≪Internet≫, ≪LAN≫, or ≪Encrypted≫ (`setCommunicationPathType(inDiagram: String, assName: String, type: String)`).

b) We create a dependency stereotyped according to the control direction of the interfaces in the problem diagram with a security requirement and according to the following rules:

- the domain controlling the interface is translated into the target of the dependency.
- if more than one observing domains exist, the same number of dependencies must be introduced.
- if a confidentiality requirement constrains the connection domain in the problem diagram exists that corresponds to the connection in the deployment diagram, then the dependency is stereotyped ≪secrecy≫.
- if an integrity requirement refers to the connection domain in the problem diagram exists that corresponds to the connection in the deployment diagram, then the dependency is stereotyped ≪integrity≫.

`createDependencies(inDiagram: String)`

The result of applying this method to the context diagram of the CACC shown in Fig. 4.19 on Page 44 is presented in Fig. 7.5. Please note that due to UMLsec limitations we have used the stereotype ≪encrypted≫ although only integrity and not confidentiality is required. This UMLsec deployment diagram can be created following the command sequence depicted in Listing 7.4.

```
createDeploymentDiagram('CACC_deployment');
addSecureLinksStereotype('CACC_deployment','default');
createNodes('CACC_deployment');
createNestedNodes({'CACC, EngineActuator_Break'});
createNestedClasses({'ACCSpeed'});
getNetworkConnections(); -- returns {'WW!{postion, speed}'}
createCommunicationPaths('CACC_deployment');
setCommunicationPathType('CACC_deployment','WW!{postion, speed}','encrypted');
createDependencies('CACC_deployment');
```

Listing 7.4: *Generating a UMLsec Deployment Diagram for CACC*

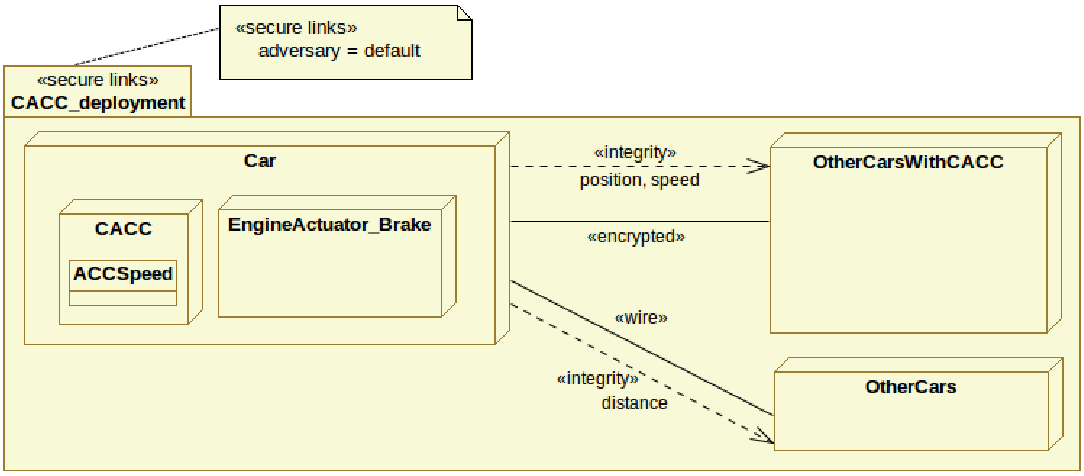


Figure 7.5.: *UMLsec Deployment Diagram Representing the Target State of CACC*

We now present the OCL specification of the model generation rule for step 5. We present the complete set of specifications of the aforementioned model generation rules in Appendix D. We express model generation rules using OCL pre- and postconditions.

Listing 7.5 contains the specification for step 5, generating the communication paths and stereotypes for those associations that can be derived directly.

```
createCommunicationPaths(inDiagram: String)
PRE  Package.allInstances() ->select(name=diagramName)
         ->size()=0 and
     Package.allInstances() ->select(getAppliedStereotypes()
       .name ->includes(’ContextDiagram’)) ->size()=1 and
     Package.allInstances() ->select(getAppliedStereotypes()
      .name ->includes(’ContextDiagram’)) .clientDependency
      .target ->select(oclIsTypeOf(Association)) .oclAsType(Association)
       ->select(not endType.getAppliedStereotypes().name
         ->includes(’BiddableDomain’)
      ).getAppliedStereotypes() ->forAll(rel_ass_st|
          not rel_ass_st.name ->includes(’ui’) and
          not rel_ass_st.general.name ->includes(’ui’) and
          -- similar for ’event’, ’call_return’, ’stream’, ’shared_memory’
      )
POST Package.allInstances() ->select(name=inDiagram).ownedElement
      ->select(oclIsTypeOf(CommunicationPath)) .oclAsType(CommunicationPath
     .endType.name =
     Package.allInstances() ->select(getAppliedStereotypes()
     .name ->includes(’ContextDiagram’)) .clientDependency
     .target ->select(oclIsTypeOf(Association)) .oclAsType(Association)
      ->select(not endType.getAppliedStereotypes().name
      ->includes(’BiddableDomain’)).endType.name and
     Package.allInstances() ->select(getAppliedStereotypes()
     .name ->includes(’ContextDiagram’)) .clientDependency
     .target ->select(oclIsTypeOf(Association)) .oclAsType(Association)
      ->select(not endType.getAppliedStereotypes().name
      ->includes(’BiddableDomain’)) ->forAll(rel_ass|
       Package.allInstances() ->select(name=inDiagram).ownedElement
        ->select(oclIsTypeOf(CommunicationPath)) .oclAsType(CommunicationPath)
        ->exists(cp |
         cp.name = rel_ass.name and
         cp.endType.name = rel_ass.endType.name and
         ( cp.getAppliedStereotypes().name ->includes(’physical’) implies
           rel_ass.getAppliedStereotypes().name ->includes(’wire’)) and
         ( cp.getAppliedStereotypes().general.name ->includes(’physical’) implies
           rel_ass.getAppliedStereotypes().name ->includes(’wire’))
       )
     )
```

Listing 7.5: *createCommunicationPaths(inDiagram: String)*

The first two formulas of the precondition of the model generation rule `createCommunicationPaths(inDiagram: String)` state that there does not exist a package named equal to the parameter `diagramName` (lines 2-3 in Listing 7.5), and that there exists exactly one package that contains a diagram stereotyped ≪ContextDiagram≫ (lines 4-5). The third formula of the precondition expresses that associations between transformed domains do not contain any of the ≪ui≫, ≪event≫, ≪call_return≫, ≪stream≫, ≪shared_memory≫, stereotypes and subtypes (lines 6-21). If these conditions are fulfilled, then the postcondition can be guaranteed, i.e., names of nodes connected by each communication path are the same as the names of domains connected by an association in the context diagram (lines 22-29), and there exists for each relevant association contained in the context diagram a corresponding and equally named communication path in the deployment diagram that connects nodes with names equal to the names of the domains connected by the association. These communication paths are stereotyped ≪wire≫ if the corresponding associations are stereotyped ≪physical≫ or a subtype (lines 30-45).

7.3.2. UMLsec Class and Sequence Diagrams for Security Mechanism Descriptions

In the following, we show how to specify security mechanisms by developing UMLsec diagrams based on security requirements. For each communication path contained in the UMLsec deployment diagram developed as shown in Section 7.3.1 that is not stereotyped ≪wire≫, we select an appropriate security mechanism according to the results of the problem analysis, e.g., MAC for integrity, symmetric encryption for security, and a protocol for key exchange). A security mechanism specification commonly consists of a structural and a behavioral description, which we specify based on the attributes of the UMLsec ≪data security≫ stereotype. To create security mechanism specifications, we developed the following model generation rules:

- Securing data transmissions using MAC: `createMACSecuredTransmission (senderNodeName: String, receiverNodeName: String, newPackage: String)`
- Symmetrically encrypted data transmissions: `createSymmetricallyEncryptedTransmission (senderNodeName: String, receiverNodeName: String, newPackage: String)`
- Key exchange protocol: `createKeyExchangeProtocol (initiatorNodeName: String, responderNodeName: String, newPackage: String)`

Model generation rules can be regarded as *patterns* for security mechanism specifications. Each of the aforementioned model generation rules describes the construction of a package stereotyped ≪data security≫ containing structural and behavioral descriptions of the mechanism expressed as class and sequence diagrams. Moreover, the package contains a UMLsec deployment diagram developed as shown in Section 7.3.1.

The model generation rule `createKeyExchangeProtocol(initiatorNodeName: String, responderNodeName: String, newPackage: String` is presented in Hatebur et al. (2011) and shown in Appendix D, Listing D.10. A detailed description of this protocol pattern is given in (Jürjens, 2005, Chapter 5.2). As an additional example, we present in this chapter the model generation rule `createMACSecuredTransmission (senderNodeName: String, receiverNodeName: String, newPackage: String)` shown in Listing 7.6 in more detail.

```
createMACSecuredTransmission(senderNodeName: String, receiverNodeName: String,
    newPackage: String)
PRE    Node.allInstances() ->select(name=senderNodeName) ->size()=1 and
       Node.allInstances() ->select(name=receiverNodeName) ->size()=1 and
       let cp_types: Bag(String) =
         CommunicationPath.allInstances()->select( cp |
           cp.endType->includes(Node.allInstances()
               ->select(name=senderNodeName)->asSequence()->first() ) and
           cp.endType->includes(Node.allInstances()
               ->select(name=receiverNodeName)->asSequence()->first() )
```

```
8          ).getAppliedStereotypes().name
9        in
10         cp_types->includes(’encrypted’) or cp_types->includes(’Internet’) or
               cp_types->includes(’LAN’) and
11       Package.allInstances() ->select(name=newPackage) ->size()=0
12
13 POST  Package.allInstances() ->select(name=newPackage) ->size()=1 and
14       -- ... Stereotype with attributes exists
15       Class.allInstances() ->select(name=senderNodeName) ->select(oclIsTypeOf(Class))
             ->size()=1 and
16       Class.allInstances() ->select(name=receiverNodeName)
             ->select(oclIsTypeOf(Class)) ->size()=1 and
17       -- ... dependencies with integrity between initiator and responder (both
             direction) created ...
18       Class.allInstances() ->select(name=senderNodeName)
             ->select(oclIsTypeOf(Class)).ownedAttribute
19         ->select(name=’inv(AuthKey)’).type ->select(name = ’Keys’) -> size() = 1 and
20       -- ... other attributes exist...
21       Class.allInstances() ->select(name=receiverNodeName)
             ->select(oclIsTypeOf(Class)).ownedOperation
22       ->select(name=’resp’)
23       ->select( member->forAll(oclIsTypeOf(Parameter))) .member ->forAll( par  |
24         par->select( name->includes(’encrData’)) ->one(
               oclAsType(Parameter).type.name->includes(’Data’))
25       ) and
26       -- ... other operations exist
27       -- ... stereotype and tags for initiator and responder class exist
28       let intera : Bag(Interaction) =
29         Package.allInstances() ->select(name=newPackage) .ownedElement
               ->select(oclIsTypeOf(Collaboration))
30         .ownedElement ->select(oclIsTypeOf(Interaction)) .oclAsType(Interaction)
31       in
32         intera.ownedElement ->select(oclIsTypeOf(Lifeline)) .oclAsType(Lifeline).name
               ->includes(senderNodeName) and
33         intera.ownedElement ->select(oclIsTypeOf(Lifeline)) .oclAsType(Lifeline).name
               ->includes(receiverNodeName) and
34         intera.ownedElement ->select(oclIsTypeOf(Message)) .oclAsType(Message).name
               ->includes(’init(Encr(inv(AuthKey),SessionKey))’) and
35         intera.ownedElement ->select(oclIsTypeOf(Message)) .oclAsType(Message).name
               ->includes(’resp(Sign(snd(Dec(inv(AuthKey)),data))’)  and
36       -- ... conditions in sequence diagram exist
```

Listing 7.6: *createMACSecuredTransmission(senderNodeName: String, receiverNodeName: String, newPackage: String)*

We use this protocol to realize the security requirement "For Driver, the influence (as described in R1 and R2) on the Car (brake, accelerate) must be either correct or in case of a modification by CACCAttacker the Car (EngineActator_Brake) shall not brake/accelerate and the Car shall inform driver." shown in Fig. 5.14 on Page 73.

The precondition of the model generation rule for key exchange protocols states that nodes named `senderNodeName` and `receiverNodeName` exist (lines 2-3 in Listing 7.6). The communication path between these nodes (line 8) should have the stereotype ≪encrypted≫, ≪Internet≫, or ≪LAN≫ (lines 4-10). Additionally, a package named `newPackage` must not exist (line 11). If these conditions are fulfilled, then the postcondition can be guaranteed. The first part of the postcondition describes the construction of a class diagram, and the second part specifies the construction of a sequence diagram. Both are created according to the pattern The following class diagram elements are created as shown in the example in Fig. 7.6 on the next page:

- exactly one package named `newPackage` (line 13)
- stereotype ≪data security≫ and tags (adversary) for this package
- classes for sender and receiver named `senderNodeName` and `receiverNodeName` (lines 15-16)

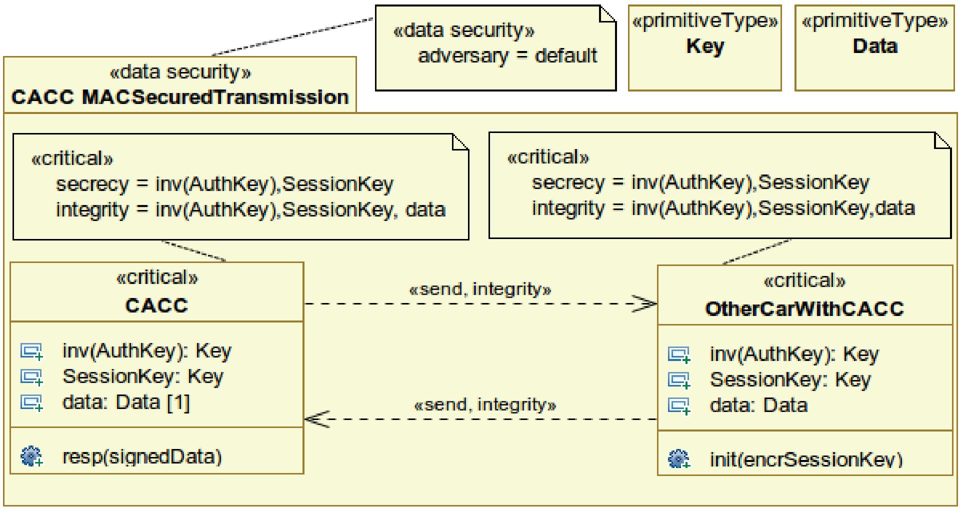


Figure 7.6.: *Class Diagram of Securing Data Transmissions using MAC for CACC*

- dependencies with ≪integrity≫ between sender and receiver (both directions)
- attributes for sender and receiver classes (lines 18-20)
- methods with parameters for sender and receiver classes (lines 21-26)
- stereotype ≪critical≫ and corresponding tags (e.g., integrity) for sender and receiver classes

The following sequence diagram elements are created as shown in the example in Fig. 7.7 on the facing page:

- lifelines for initiator and for responder in an interaction being part of a collaboration that is part of the created package (lines 28-33)
- messages in sequence diagram (lines 34-35)
- the condition in the sequence diagram as a guard for the following messages[2]

Figure 7.6 shows the class diagram and Fig. 7.7 the sequence diagram developed for the CACC according to this model generation rule. They are created with `createMACSecuredTransmission('CACC', 'OtherCarWithCACC', 'CACC MACSecuredTransmission')`.

In the created model, the tag secrecy of the ≪critical≫ class CACC contains the secret SessionKey, which represents a the randomly chosen secret to be exchanged by this protocol. It also contains the key inv(AuthKey) used to decrypt the secret SessionKey, Next to these assets, the integrity tag additionally contains the data to be transmitted.

The tag secrecy of the ≪critical≫ class OtherCarWithCACC contains the session keys SessionKey and the authentication key inv(AuthKey) of the OtherCarWithCACC. The integrity tag consists of assets similar to the ones of the same tag of the CACC.

These tag values are reasonable because the security domain knowledge in Section 6.3 states that

- confidentiality of SessionKey1, SessionKey2, and the data of CACC have to be preserved (see Fig. 6.1 on Page 94),

[2] Since UMLsec is based on UML 1.3 and UML 1.3 does not support combined fragments, in UMLsec conditions are annotated as line comments.

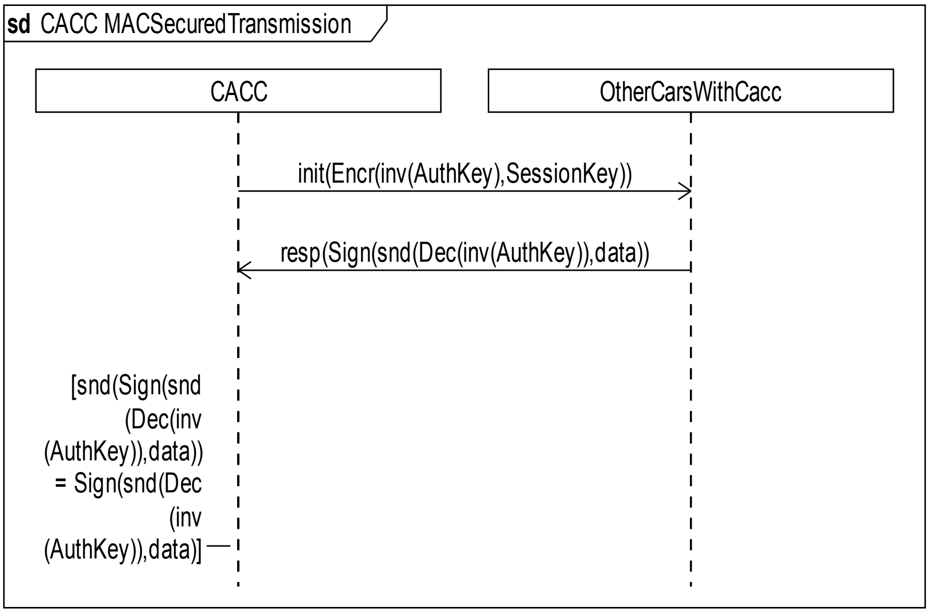

sd CACC MACSecuredTransmission
CACC
OtherCarsWithCacc
init(Encr(inv(AuthKey),SessionKey))
resp(Sign(snd(Dec(inv(AuthKey)),data))
[snd(Sign(snd (Dec(inv (AuthKey)),data)) = Sign(snd(Dec (inv (AuthKey)),data)]

Figure 7.7.: *Sequence Diagram of Securing Data Transmissions using MAC for CACC*

- integrity of SessionKey1, SessionKey2, and the data of CACC have to be preserved (see Fig. 6.2 on Page 94),
- confidentiality of AuthKey1 and AuthKey2 have to be preserved (see Equations 6.13 and 6.15 on Page 97), and
- integrity of AuthKey1 and AuthKey2 have to be preserved (see Equations 6.14 and 6.16 on Page 97).

The sequence diagram in Fig. 7.7 specifies two messages and one guard. The first message from CACC initiates the communication by sending a generated SessionKey, encrypted with the authentication key (AuthKey) to an OtherCarWithCACC. This car responds by sending its data (position and speed) together with a MAC calculated with the SessionKey (using the operation *Sign*). Only if the guard at the lifeline of the Car is true, i.e., the calculated MAC is correct, the data is used for further processing within CACC.

7.3.3. Tool Design

We are currently constructing a graphical wizard-based tool that supports a software engineer in interactively generating UMLsec specification models. The tool will implement the model generation rules presented in the previous subsections to generate UMLsec deployment, class, and sequence diagrams. A graphical user interface allows users to choose the parameters, and it ensures that these parameters fulfill the precondition. For example, users can choose the value of the second parameter of the model generation rule `setCommunicationPathType(inDiagram: String, assName: String, type: String)` based on the return values of the rule `getNetworkConnections()`. Our tool will automatically construct the corresponding parts of the UMLsec model as described in the postcondition. Since our model generation rules are specified with OCL in a formal and evaluatable way, our tool implementation can be checked automatically for correctness with respect to our specification based on an appropriate

API such as the Eclipse implementation for EMF-based models (*Eclipse Modeling Framework Project (EMF)*, 2009).

In summary, we presented in this section a novel integrated and formal approach connecting security requirements analysis and secure specification.

7.4. Related Work

The method for handling functional requirements and specifications presented in Section 7.1 is directly based on the method developed by Jackson (2001).

The UML integration by using sequence diagrams (see Section 7.2) is also addressed by Lavazza and Bianco (2004), Bianco and Lavazza (2006), Konrad and Cheng (2002) and Choppy and Reggio (2005). We extended the presented methods by formal constraints about consistency between problem diagrams and sequence diagrams. We are not aware of any other work on expressing such constraints.

The method for handling security requirements presented in Section 7.3 can be compared on the one hand-side to other work bridging the gap between security requirements engineering secure specification, and on the other hand-side to work on transforming UML models based on rules expressed in OCL.

Relatively little work has been done on the first category of related work, i.e., bridging the gap between security requirements analysis and design. Recently, an approach to connect the security requirements analysis method *Secure Tropos* by Mouratidis et al. (Giorgini & Mouratidis, 2007) and UMLsec (Jürjens, 2005) was published by Mouratidis and Jürjens (2010). A further approach from Houmb, Islam, Knauss, Jürjens, and Schneider (2010) connects UMLsec with security requirements analysis based on heuristics. In contrast to our work, these approaches only provide informal guidelines for the transition from security requirements to design. Consequently, they do not allow one to verify the correctness of this transition step.

The second category of related work considers the transformation of UML models based on *OCL transformation contracts* (Cariou, Marvie, Seinturier, & Duchien, 2004; Millan, Sabatier, Le Thi, Bazex, & Percebois, 2009). We basically use parts of this work, e.g., the specification of transformation operations using OCL pre- and postconditions. Additionally, our model generation rules can be seen as patterns, because they describe the generation of completely new model elements according to generic security mechanisms, e.g., cryptographic keys.

7.5. Conclusions and Future Work

In this chapter, we have presented a method for transforming functional requirements into specifications.

We have also defined constraints describing the consistency between problem diagrams and the specifications expressed with sequence diagrams. These constraints are specified with OCL, which allows one to check them with out tool UML4PF.

We have also presented in this chapter a novel method to bridge the gap between security requirements analysis and secure specification. We complemented our method by *formal model generation rules* expressed in OCL. Thus, the construction of UMLsec specification models based on results from security requirements engineering becomes

- more feasible,
- systematic,
- less error-prone, and

- and a more routine engineering activity.

We validated the presented approach on the CACC case study.

Currently, our method is limited to functional requirements and security requirements to be transformed into sequence diagrams. In the same way, it is possible to extend the method to generate other notations expressing behavior. It can also be extended to other dependability requirements that can be transformed into functional specifications.

Chapter 8

Systematic Architectural Design based on Problem Patterns

In this chapter, we present a systematic method to derive system and software architectures from problem descriptions. We give detailed guidance by elaborating concrete steps that are equipped with validation conditions. The method works for different types of systems, e.g., for embedded systems, web-applications, and distributed systems as well as standalone ones.

In previous work, we (Choppy et al., 2005) proposed architectural patterns for each of the basic problem frameproposed by Jackson (Jackson, 2001). In a follow-up paper (Choppy, Hatebur, & Heisel, 2006), we showed how to merge the different sub-architectures obtained according to the patterns presented in (Choppy et al., 2005), based on the relationship between the subproblems. In (Hatebur & Heisel, 2009a), we showed how interface descriptions for layered architectures can be derived from problem descriptions. A more flexible approach that keeps the advantages is presented in this chapter. It is based on joint work and is published in Choppy et al. (2011). The published version focuses on software architecture and applies the method to an Automatic Teller Machine (ATM) case study. In this chapter, we additionally consider the system architecture, and we apply the procedure to the CACC case study.

The method is based on different kinds of patterns. On the one hand, it makes use of *problem frames* (Jackson, 2001) (see Chapter 4). On the other hand, it builds on architectural and design patterns.

The starting point of the method is a set of diagrams that are set up during requirements analysis, in particular, the technical context diagram. Furthermore, the overall development problem must be decomposed into simple subproblems, which are represented by problem diagrams. The different subproblems should be instances of problem frames.

From these pattern-based problem descriptions, we derive an architecture that is suitable to solve the development problem described by the problem descriptions. The problem descriptions as well as the architectures are represented as UML diagrams, extended by stereotypes. The stereotypes are defined in profiles that extends the UML metamodel (UML Revision Task Force, 2010c).

The method to derive architectures from problem descriptions consists of three steps. In the first step, an initial architecture is set up. It contains one component for each submachine in the problem diagrams. The overall machine component has the same interface as described in the technical context diagram (e.g., ≪call_and_return≫, ≪shared_memory≫, ≪event≫, ≪ui≫).

In the second step, we apply different architectural and design patterns. We introduce coordinator and facade components and specify them. A facade component is necessary if several internal components are connected to one external interface. A coordinator component must be added if the interactions of the machine with its environment must be performed in a certain order. For different problem frames, specific architectural patterns are applied.

In the final step, the components of the implementable architecture are re-arranged to form a layered architecture, and interface and driver components are added. This process is driven by the stereotypes introduced in the first step. For example, a connection stereotype ≪ui≫ motivates to introduce a user interface component. Of course, a layered architecture is not the only possible way to structure the software, but a very convenient one. We have chosen it because a layered architecture makes it possible to divide platform-dependent from platform-independent parts, because different layered systems can be combined in a systematic way, and because other architectural styles can be incorporated in such an architecture. Furthermore, layered architectures have proven useful in practice.

Our method exploits the subproblem structure and the classification of subproblems by problem frames. Additionally, most interfaces can be derived from the problem descriptions (Hatebur & Heisel, 2009a). Stereotypes guide the introduction of new components. They also can be used to generate adapter components automatically. The re-use of components is supported, as well.

The method is tool-supported, as described in Section 4.2 on Page 38. To support the architectural design, we added a UML profile that allows us to annotate composite structure diagrams with information on components and connectors. In order to automatically validate the semantic integrity and coherence of the different models, we provide a number of validation conditions. These conditions concern the following:

- coherence of problem descriptions and architectural descriptions
- internal coherence of single architectural descriptions
- coherence of different architectural descriptions

In Section 8.1, we introduce the UML profile for architectural descriptions that we have developed and which provides the notational elements for the architectures we derive. In Section 8.2, we describe our method in detail. Not only do we give guidance on how to perform the three steps, but we also give detailed validation conditions that help to detect errors as early as possible. As a running example, we apply our method to derive the architecture for the CACC. Section 8.3 discusses related work, and in Section 8.4, we give a summary of our achievements and point out directions for future work.

We illustrate the method by deriving an architecture for the CACC (see 4.3 on Page 43).

8.1. Architectural Descriptions

For each machine in the context diagram, we design an architecture that is described using composite structure diagrams (UML Revision Task Force, 2010c). In such a diagram, the components with their ports and the connectors between the ports are given. The components are another representation of UML classes. The ports are typed by a class that uses and realizes interfaces. An example is depicted in Figure 8.5 on Page 126. The ports (with this class as their type) provide the implemented interfaces (depicted as lollipops) and require the used interfaces (depicted as sockets), see Fig. 8.4 on Page 125.

In our UML profile, we introduce stereotypes to indicate which classes are components. The stereotype ≪Component≫ extends the UML meta-class Class. For re-used components we use the stereotype ≪ReusedComponent≫, which is a specialization of the stereotype ≪Component≫. Reused components may also be used in other projects. This fact must be recorded in case such a component is changed. Fig. 8.1 on the next page depicts this part of our profile.

A machine domain may represent completely different things. It can either be a distributed system (e.g., a network consisting of several computers), a local system (e.g., a single computer), a process running on a certain platform, or just a single task within a process (e.g., a clock as part of a graphical user interface). The kind of the machine can be annotated with the stereotypes

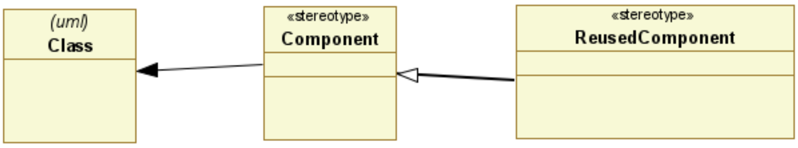


Figure 8.1.: *Components*

≪distributed≫, ≪local≫, ≪process≫, or ≪task≫. They all extend the UML meta-class Class. If design decisions on the operating system, the processor architecture, or the provided memory and speed have been made, the attributes of these stereotypes can be used to document the design decisions. This documentation is helpful for further design decisions, e.g. Windows Message Queues cannot be used to connect components on a Linux operating system. Fig. 8.2 depicts this part of our profile.

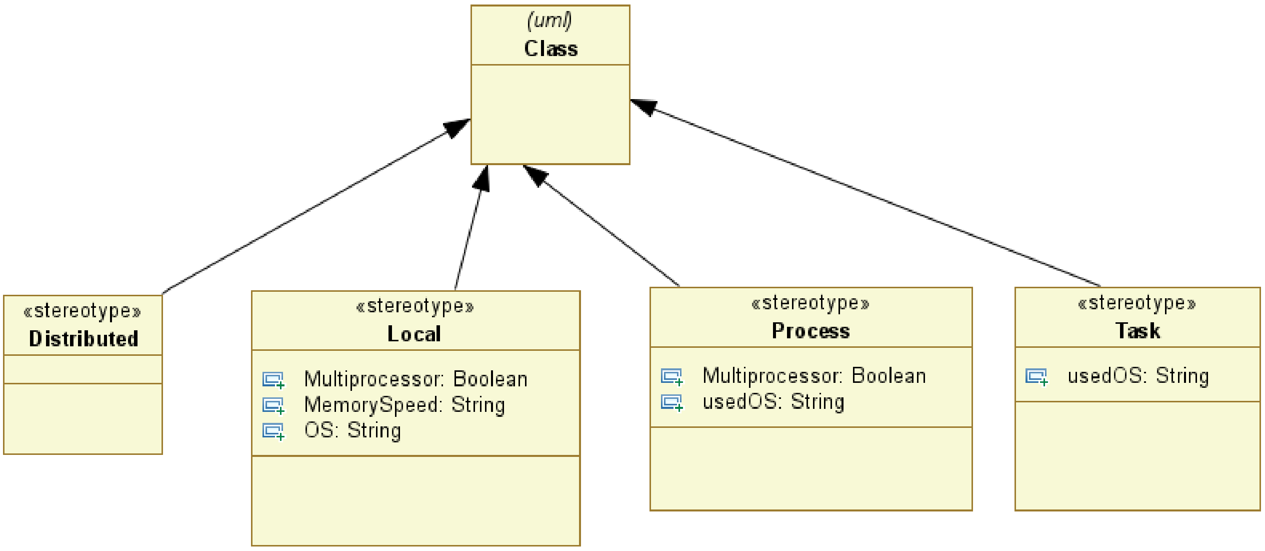


Figure 8.2.: *Machine Types*

For the architectural connectors, we allow the same stereotypes as for associations, e.g. ≪ui≫ or ≪tcp≫, described in Section 4.1.2 on Page 25. However, these stereotypes extend the UML meta-class Connector (instead of the meta-class Association).

8.2. Deriving Architectures from Problem Descriptions

We now present our method to derive software architectures from problem descriptions in detail. For each of its three steps, we specify the input that is needed, the output that is produced, and a procedure that can be followed to produce the output from the input. Of course, these procedures are not automatic, and a number of decisions have to be taken by the developer. Such developer decisions introduce the possibility to make errors. To detect such errors as early as possible, each step of the method is equipped with validation conditions. These validation conditions must be fulfilled if the developed documents are semantically coherent. For example, a passive component cannot contain an active component. The validation conditions cannot be complete in a formal sense. Instead, they constitute necessary but not sufficient conditions for the different documents to be semantically valid. New conditions can be defined and integrated in our tool as appropriate.

8.2.1. Starting Point

To derive the architecture, we have to describe how the machine is embedded in its environment. The technical context diagram contains the necessary information. For the CACC case study, the technical context diagram given is in Fig. 8.3. In this diagram, the machine CACC is split into software (Cacc_Sw) and hardware (Cacc_Hw). It only contains the domains directly connected to the machine in the context diagram.

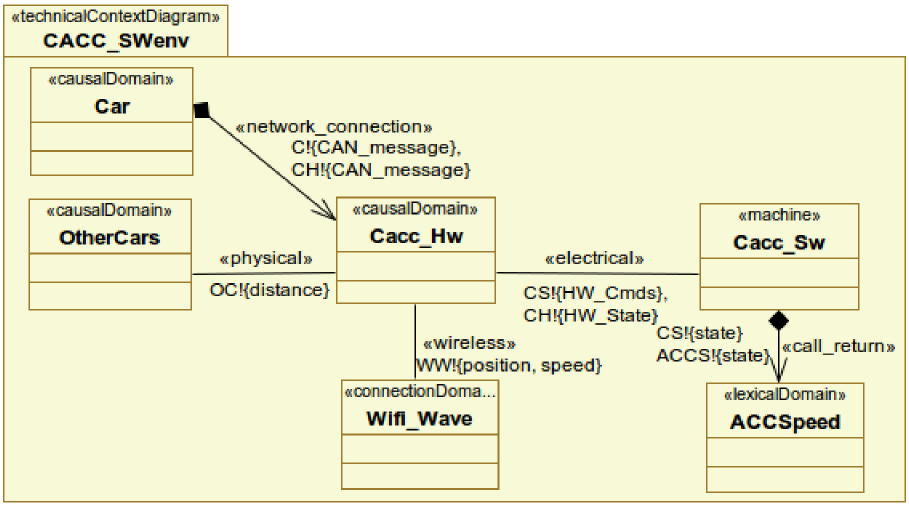


Figure 8.3.: *Technical Context Diagram of CACC*

Furthermore, the overall software development problem must be decomposed into simple subproblems, which are represented by problem diagrams, as shown in Section 4.3. The subproblems can be found in Figs. 4.21 on Page 46, 4.22 on Page 47, and 4.23 on Page 48. Finally, the relations between the different subproblem (life-cycle model, e.g. sequential or alternative) must be known. In CACC case study, all subproblem have to work in parallel.

8.2.2. Initial Architecture

The purpose of this first step is to collect the necessary information for the architectural design from the requirements analysis phase, to determine which component has to be connected to which external port, to make coordination problems explicit (e.g. several components are connected to the same external domain), and to decide on the machine type, verifying that it is appropriate (considering the connections). At this stage, the submachine components are not yet coordinated.
The *input* for this step are the technical context diagram and the problem diagrams. The *output* is an initial architecture, represented by a composite structure diagram. It is set up as follows. There is one *component* for the overall machine with stereotype ≪machine≫, and it is equipped with ports corresponding to the interfaces of the machine in the technical context diagram, see Fig. 8.4.

Inside this component, there is one component for each submachine identified in the problem diagrams, equipped with ports corresponding to the interfaces in the problem diagrams, and typed with a class. This class has required and provided *interfaces*. A controlled interface in a problem diagram becomes a required interface of the corresponding component in the architecture. Usually, an observed interface of the machine in the problem diagram will become a provided interface of the corresponding component in the architecture. However, if the interface connects a lexical domain, it will be a required interface containing operations with return values (see Côté et al. (2008, Section 3.1)). The ports of the components should be connected to the ports of the machine, and stereotypes describing the technical realization of these connectors are added. A stereotype describing the type of the machine (local, distributed, process, task) is added, as well as stereotypes ≪ReusedComponent≫ or ≪Component≫ to all components. If appropriate, stereotypes describing the type of the components (local, distributed, process, task) are also added.

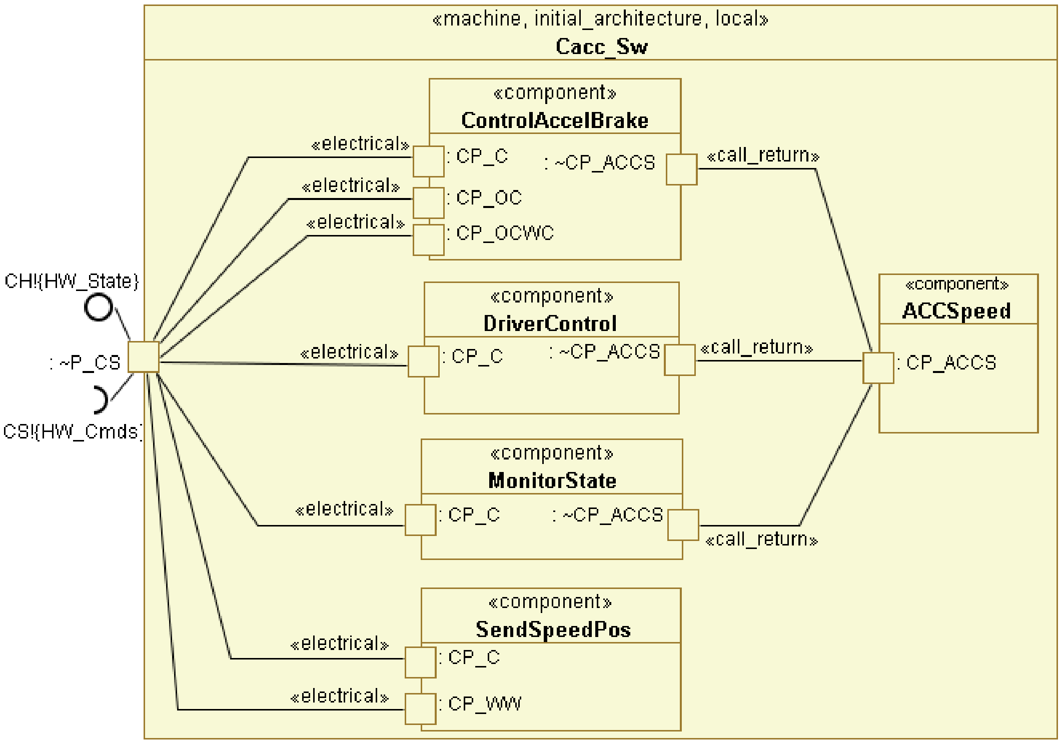


Figure 8.4.: *Initial Architecture of CACC*

The initial architecture of the CACC software is given in Fig. 8.4. Note that it is not necessary to split the machine into hardware and software beforehand: it is possible to create an initial architecture for the machine including hardware and software. This initial architecture can be split later according to a desired design in the next Step "Implementable Architecture". Starting from the technical context diagram in Fig. 8.3 on the preceding page, and the problem diagrams (see Figs. 4.21 on Page 46, 4.22 on Page 47, 4.23 on Page 48, and 4.24 on Page 48), the initial Cacc_Sw architecture has the stereotypes ≪machine,initial_architecture,local≫ and one external port typed with :P_CS that corresponds to the interface of the machine in the technical context diagram. The components ControlAccelBrake, DriverControl, MonitorState, and SendPosSpeed correspond to the submachines identified for this case study (see Figs. 4.21, 4.22, 4.23, and 4.24 on Page 48). The component ACCSpeed is included, because the technical context diagram in Fig. 8.3 on the preceding page expresses that it is part of the machine. Phenomena at the machine interface in the technical context diagram (CS!{HW_Cmds} and CH!{HW_State}) now occur in external interfaces of the machine. Phenomena controlled by the machine are associated with required interfaces (CS!{HW_Cmds}), and phenomena controlled otherwise (e.g.

by the user), are associated with provided interfaces (CH!{HW_State}).

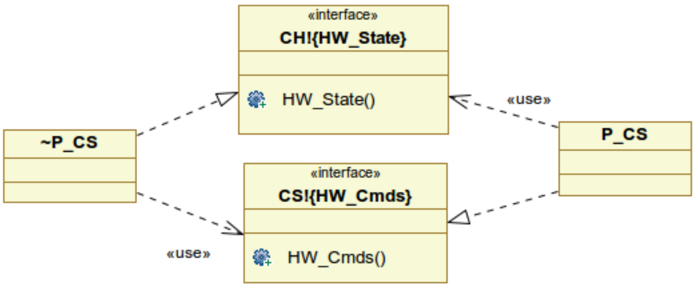


Figure 8.5.: *Port Type of P_CS*

Note that connections in the technical context diagram in Fig. 8.3 being not related to the Cacc_Sw (such as the one between Cacc_Hw and OtherCars) are not reflected in the initial architecture.

The ports have a class as a type. This class uses and realizes interfaces. For example, as depicted in Fig. 8.5, the class ~P_CS realizes the interface CH!{HW_State} and uses the interface CS!{HW_Cmds}. The ports with this class as a type (see Fig. 8.4) provide the interface CH!{HW_State} (depicted as a lollipop) and requires the interface CS!{HW_Cmds} (depicted as a socket). The definition of the other used port types is provided in Appendix B, Figures B.2 and B.3 on Page 234.

We have defined two sets of validation conditions for this first phase of our method. The first set is common to all architectures (and hence should be checked after each step of our method), whereas the second one is specialized for the initial architecture. We give a selection of the validations conditions in the following. Additional validations conditions can be found in Appendix C, Sections C.12 and C.13.

Following our approach, a strong relation between the problem analysis and the developed architectures exists, i.e., the machines of the subproblems yield components that are part of a machine or another component. Moreover, in all architectures, required operations must be provided by the connected component, and the composition hierarchy must be valid, e.g., a distributed network cannot be contained in a single process. We first present some conditions that should be satisfied by all architectures in our development approach. We then provide the OCL expression for condition VA.1 in Listing 8.1, as well as a way to find out where an error comes from if this condition is not satisfied.

Validation conditions for All architectures:

VA.1. Each machine in a problem diagram must be a component or a re-used component in the architectural description. For example in the CACC case study, MonitorState in the problem diagram in Fig. 4.23 on Page 48 has the stereotype ≪Component≫ (see Fig. 8.4 on the preceding page).

VA.2. All components in the model must be contained in a machine or another component.

VA.3. For each operation in a *required* interface of a port of a component, there exists a connector to a port *providing* an interface with this operation, or it is connected to a re-used component. For example in the CACC case study, the operation HW_Cmds() in the interface CS!{HW_Cmds} is required by the port with type ~P_CS of the component Cacc_Sw and provided by the port with type P_CS of the component Cacc_Hw.

VA.4. If a component port is connected to an external port of the machine, the components' interfaces must fit to the connected interfaces of the machine, i.e., each operation in a required or provided interface of a component port must correspond to an operation in a required or provided interface of a connected machine port. For example in the CACC case study, the operation CAN_message() required by the ports with the type P_C corresponds to the operation clHW_Cmds() the ports with the type ∼P_CS.

VA.5. Passive components cannot contain active components.

VA.6. Classes with stereotype ≪machine≫ must also have one of the stereotypes ≪Distributed≫, ≪Local≫, ≪Process≫, ≪Task≫, ≪Component≫, or ≪ResusedComponent≫.

VA.7. A class with the stereotype ≪Local≫ cannot contain classes with the stereotype ≪Distributed≫.

In the following, we present the OCL expression for **Condition VA.1**. All packages in the model with the stereotype ≪ProblemDiagram≫ (Listing 8.1, line 1) are selected. For these packages we collect the targets of the package dependencies with the stereotype ≪isPart≫ (line 2) and select those being classes and having the stereotype ≪machine≫ (lines 3 and 4). These classes must also have the stereotype ≪Component≫ or ≪ReusedComponent≫ (line 5).

```
1 Package.allInstances() ->select(getAppliedStereotypes().name
      ->includes(’ProblemDiagram’))
2   .clientDependency ->select(getAppliedStereotypes().name ->includes(’isPart’)).target
3   ->select(oclIsTypeOf(Class))
4   ->select(getAppliedStereotypes().name ->includes(’Machine’))
5   ->forAll(getAppliedStereotypes().name ->includes(’Component’) or
        getAppliedStereotypes().name ->includes(’ReusedComponent’))
```

Listing 8.1: *Machines in problem diagrams must be components*

To find out which problem diagram machines are not components, the same expression can be used with reject instead of forAll.

The initial architecture should fit to the technical context diagram, i.e, the external connections and interfaces of the architecture must correspond to associations and interfaces of the machines in the technical context diagram. The following conditions consider the initial architecture and its relationship with the technical diagram.

Validation conditions specific to the Initial architecture:

VI.1. For each provided or required interface of machine ports in the initial architecture, there exists a corresponding interface in the technical context diagram. Corresponding interface are

a) exactly the observed or controlled interfaces,

b) interfaces combined from observed or controlled interfaces,

c) interfaces contained in observed or controlled interfaces,

d) interfaces being refined or concretized by observed or controlled interfaces, or

e) interfaces refining or concretizing observed or controlled interfaces.

VI.2. For each observed or controlled interface of each machine in the technical context diagram, the corresponding machine component of the architecture has ports providing or requiring the corresponding interfaces. Corresponding interface are the same as in Condition VI.1

VI.3. Each phenomenon in a machine interface in the technical context diagram must correspond to an operation in some interface of the corresponding machine component of the architecture. For this condition we provide no OCL because operation X() may be refined by Y() and Z().

VI.4. For each machine in the technical context diagram:
each stereotype name of all associations to the machine (or a specialization of this stereotype) must be included in the set of stereotype names of the connectors from the internal components to external interfaces inside the machine. In the CACC case study, the association to the domain Cacc_Sw in the technical context diagram in Fig. 8.3 on Page 124 has the stereotype ≪electrical≫ and all connections to the corresponding port in Fig. 8.4 on Page 125 have the stereotype ≪electrical≫.

VI.5. Each stereotype name of the connectors from components to external interfaces inside an architectural machine component (or their supertypes) must be included in the set of associations to the corresponding machine domain in the technical context diagram.

In the following, we present the OCL expressions checking **Condition VI.1**. It checks that for each provided or required interfaces of machine ports, there exists a corresponding interface in the technical context diagram. In the OCL expression in Listing 8.2, we define the set of interfaces tcd_if as all interfaces being part of the technical context diagram (lines 1-5). We select all technical context diagrams[1] (line 6) and collect all parts of these diagrams (line 7) with the stereotype ≪machine≫ (line 8). For all technical context diagram machines (tcd_machine) (line 9) we define the set of interfaces tcd_machine_ifs to be all interfaces observed or controlled by tcd_machine (lines 10-17), we define the set of interfaces tcd_m_contained_ifs to be all interfaces contained in the interfaces in tcd_machine_ifs (lines 18-22), we define the set of interfaces tcd_m_combined_ifs to be all interfaces combining the interfaces in tcd_machine_ifs (lines 23-32), we define the set of interfaces tcd_m_concr_ifs to be all interfaces concretizing or refining the interfaces in tcd_machine_ifs (lines 33-39), and we define the set of interfaces tcd_m_abst_ifs to be all concretized or refined interfaces of tcd_machine_ifs (lines 40-51), and we define the set of interfaces tcd_machine_port_ifs to be all interfaces being provided or required by tcd_machine (lines 52-56). For each interface in the set tcd_machine_port_ifs (tmpi) (line 57) we check that it is

1. exactly the observed or controlled interface (line 58),
2. an interface combined from observed or controlled interface (line 59),
3. an interface contained in observed or controlled interface (line 60),
4. an interface being refined or concretized by observed or controlled interface (line 61), or
5. an interface refining or concretizing observed or controlled interface (line 62).

```
 1 let tcd_ifs: Set(Interface) =
 2   Package.allInstances() ->select(getAppliedStereotypes().name
         ->includes('TechnicalContextDiagram'))
 3   .clientDependency ->select(getAppliedStereotypes().name ->includes('isPart')).target
 4   ->select(oclIsTypeOf(Interface)).oclAsType(Interface)->asSet()
 5 in
 6   Package.allInstances() ->select(getAppliedStereotypes().name
         ->includes('TechnicalContextDiagram'))
 7   .clientDependency ->select(getAppliedStereotypes().name ->includes('isPart')).target
 8   ->select(getAppliedStereotypes().name ->includes('Machine')).oclAsType(Class)
 9   ->forAll(tcd_machine |
10     let tcd_machine_ifs: Set(Interface) =
11       tcd_machine.clientDependency
12       ->select(
13           getAppliedStereotypes().name->includes('observes') or
14           getAppliedStereotypes().name->includes('controls'))
15       .target.oclAsType(Interface)->select(mif | tcd_ifs->includes(mif))
```

[1] Several technical context diagrams may exist to describe different aspects or the context of different machines.

```
    ->asSet()
  in
  let tcd_m_contained_ifs: Set(Interface) =
    tcd_machine_ifs.member
    ->select(oclIsTypeOf(Property)).oclAsType(Property).type
    ->select(oclIsTypeOf(Interface)).oclAsType(Interface) ->asSet()
  in
  let tcd_m_combined_ifs: Set(Interface) =
    Interface.allInstances()->select(
      let comb_elem: Set(Interface) =
        member->select(oclIsTypeOf(Property)) .oclAsType(Property).type
        ->select(oclIsTypeOf(Interface)).oclAsType(Interface) ->asSet()
      in
        comb_elem->size()>0 and
        tcd_machine_ifs->includesAll(comb_elem)
    )
  in
  let tcd_m_concr_ifs: Set(Interface) =
    tcd_machine_ifs.clientDependency->select(
      getAppliedStereotypes().name ->includes('concretizes') or
      getAppliedStereotypes().name ->includes('refines')
    ).target
    ->select(oclIsTypeOf(Interface)).oclAsType(Interface) ->asSet()
  in
  let tcd_m_abstr_ifs: Set(Interface) =
    Interface.allInstances()->select(
      let abstr_ifs: Set(Interface) =
        clientDependency->select(
          getAppliedStereotypes().name ->includes('concretizes') or
          getAppliedStereotypes().name ->includes('refines')
        ).target
        ->select(oclIsTypeOf(Interface)).oclAsType(Interface) ->asSet()
      in
        tcd_machine_ifs->exists(ti | abstr_ifs->includes(ti))
      )
  in
  let tcd_machine_port_ifs: Set(Interface) =
    tcd_machine.member ->select(oclIsTypeOf(Port)) .oclAsType(Port).required->union(
    tcd_machine.member ->select(oclIsTypeOf(Port)) .oclAsType(Port).provided)
    ->asSet()
  in
    tcd_machine_port_ifs ->forAll( tmpi|
              tcd_machine_ifs  ->includes(tmpi) or
              tcd_m_contained_ifs ->includes(tmpi) or
              tcd_m_combined_ifs ->includes(tmpi) or
              tcd_m_concr_ifs ->includes(tmpi) or
              tcd_m_abstr_ifs ->includes(tmpi)
    )
)
```

Listing 8.2: *Machine port interfaces are in technical context diagram*

As already noticed, these validation conditions can be checked automatically, using the tool UML4PF described in Section 4.2 on Page 38.

8.2.3. Implementable Architecture

The purpose of this step is to introduce coordination mechanisms between the different submachine components of the initial architecture and its external interfaces, thus obtaining an implementable architecture. This implementable architecture can be either a system architecture or a software architecture. Components or machines with the stereotype ≪distributed≫ describe a system architecture and components or machines with the stereotypes ≪local≫, ≪process≫ or ≪task≫ describe a software architecture.

Moreover, we exploit the fact that the subproblems are instances of problem frames by applying architectural patterns that are particularly suited for some of the problem frames. We also decide if some domains should be merged or split. Finally, we decide whether the components should be implemented as active or passive components.

The *input* to this step are the initial architecture, the problem diagrams as instances of problem frames, and a specification of interaction restrictions[2]. The *output* is an architecture that is already implementable. It contains coordinator and facade components as well as architectural patterns corresponding to the used problem frames. The implementable architecture is annotated with the stereotype ≪implementable_architecture≫ to distinguish it from the final architecture.

The implementable architecture is set up as follows. When several internal components are connected to one external interface in the initial architecture, a *facade* component[3] is added. Such a component has one provided interface containing all operations of some external port and several used interfaces as provided by the submachine components. The implementable architecture for our case study is depicted in Fig. 8.6.

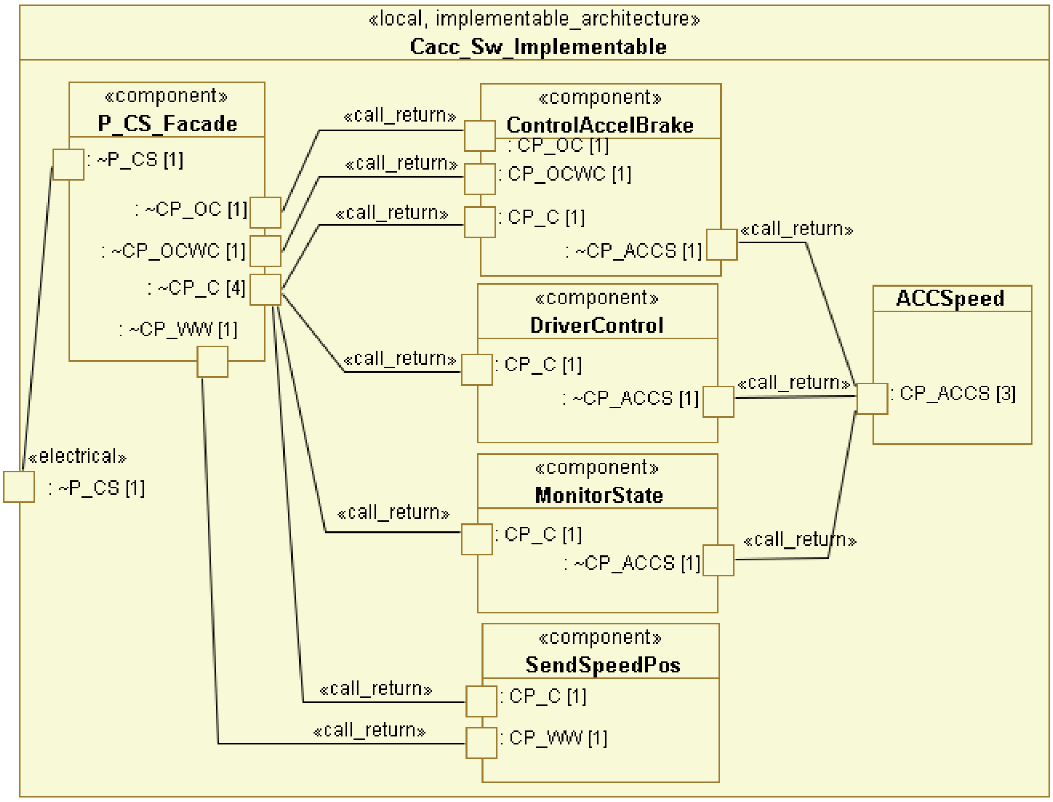


Figure 8.6.: *Implementable Architecture of CACC*

If interaction restrictions have to be taken into account, we need a component to enforce these restrictions. We call such a component a *coordinator* component. A coordinator component has one provided interface containing all operations of some external port and required interfaces containing all operations of some internal port. To ensure the interaction restrictions, a state machine can be used inside the component. Typically, coordinator components are needed for interfaces connected to biddable domains (also via connection domains). This is because often, a user must do things in a certain order. In our example, a user must first authenticate

[2] Our method does not rely on how these restrictions are represented. Possible representations are sequence diagrams, state machines, grammars, or life-cycle expressions as used in the PCS case study in Section 13.6

[3] See the corresponding design pattern by Gamma et al. (Gamma, Helm, Johnson, & Vlissides, 1995b): "Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystems easier to use."

before being allowed to enter a request to withdraw money. The coordinator components can be integrated into the facade components. Usually, it is efficient to integrate the coordinator component into the facade component.

For the case study presented here, no coordinator is necessary, because all components work in parallel. We introduce a facade named P_PC_Facade in Fig. 8.6 for the communication with other car components via CAN (Display, EngineActuator_Brake), WiFi_Wave and the distance measurement with a radar being part of the CACC hardware.

Additional to facade and coordinator components, additional architectural patterns can be applied. Problem frames can help to select appropriate patterns, because similar problems may have similar solutions. We identified the following architectural patterns that are related to problem frames:

- Fig. 8.7 shows an architectural pattern for transformation problems. It is a pipe-and-filter architecture.

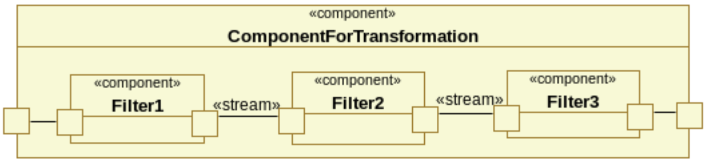


Figure 8.7.: *Pattern for Component Realizing Transformation Problem*

- The machine in a simple transformation problem frame (see Section 4.1.6) or the required behaviour problem frame (see Section 2.4) needs to trigger the activity on the constrained domain automatically. Therefore, the architectural pattern that can be applied to this machine is a decomposition into an active timer that triggers the activity and a passive component that performs the processing. The timer component is usually a re-used component.

- The machine in a commanded information problem (see Fig. 2.3 on Page 7) has to acquire information, store them and provide requested information. Therefore, the architectural pattern that can be applied for these machine is a decomposition into components for the acquisition, the storage, and the request of information.

After adding facade and coordinator components and applying architectural patterns related to problem frames, we have to decide for each component if it has to be active or not. In the case of the CACC, the components ControlAccelBrake and MonitorState are active components since they trigger actions themselves and work in parallel to the other components. The component DriverControl only reacts to actions performed by the driver. The component ACCSpeed just acts as a passive data storage, and the component P_CS_Facade just forwards information from and to the other components. For new connectors, their technical realization should be added as stereotypes. For the CACC case study, we use the stereotype ≪call_return≫ for all new connectors, because the connected parts are software and a call and return interface can be implemented easily. Finally, for all newly introduced components it has to be specified if they are a ≪Component≫ or a ≪ReusedComponent≫. In Figure 8.6, we have no re-used components.

To validate the implementable architecture, we have to check (among others) the following conditions (in addition to the conditions VA.1 to VA.7 given in Section 8.2.2).

Validation conditions for the iMplementable architecture:

VM.1. All components of the initial architecture must be contained in the implementable architecture.

VM.2. The connectors connected to the ports in the implementable architecture must have the same stereotypes or more specific ones than in the initial architecture. In the CACC case study, the connectors also have the stereotypes ≪electrical≫ and ≪call_return≫.

VM.3. The stereotypes ≪physical≫ and ≪ui≫, and their subtypes are not allowed between components.

VM.4. Ports of components of the implementable architecture that are connected to external ports, require and provide the same interfaces as the external ports. In the CACC case study, the type of both ports is ∼P_CS and therefore they require and provide the same interfaces.

VM.5. It is only possible to perform operation calls on one local machine. A shared memory and a unix pipe also cannot be used between different computers. Therefore, connectors with the stereotypes ≪shared_memory≫, ≪unix_pipe≫ and subtypes as well as the stereotype ≪call_return≫ do not connect different classes with the stereotype ≪Local≫ or ≪Distributed≫.

In the following Listing 8.3, we present the OCL expression checking **Condition VM.1**. It checks whether all components of the initial architectures are contained in the implementable architectures. For each intermediate architecture (line 1-3), the set of contained component is determined (comps, lines 4-7). For these components, the set of contained component is determined (comp_comps, lines 8-11). For each component in the initial (generalized) architecture (line 12), the components are collected (line 12) and it is checked that each of these components is part of the implementable architecture (line 15) or its components (line 16).

```
1  Class.allInstances()->select(
2    getAppliedStereotypes().name ->includes(’implementable_architecture’)
3  )->forAll(ia |
4    let comps: Set(Class) =
5      ia.ownedAttribute.type
6      ->select(oclIsTypeOf(Class) and oclAsType(Class) .getAppliedStereotypes().name
           ->includes(’Component’)).oclAsType(Class)->asSet()
7    in
8    let comp_comps: Set(Class) =
9      comps.ownedAttribute.type
10     ->select(oclIsTypeOf(Class) and oclAsType(Class) .getAppliedStereotypes().name
           ->includes(’Component’)).oclAsType(Class)->asSet()
11   in
12     ia.generalization.oclAsType(Class).ownedAttribute.type
13     ->select(oclIsTypeOf(Class) and oclAsType(Class).getAppliedStereotypes().name
           ->includes(’Component’)).oclAsType(Class)->asSet()
14     ->forAll( general_comp |
15       comps->includes(general_comp) or
16       comp_comps->includes(general_comp)
17   )
18 )
```

Listing 8.3: *Connectors in the specialized architecture must have the same components as the more general architecture*

8.2.4. Layered Architecture

In this step, we finalize the software architecture. We make sure to handle the external connections appropriately. For example, for a connection marked ≪gui≫, we need a component handling the input from the user. For ≪physical≫ connections, we introduce appropriate driver components, which are often re-used.

We arrange the components in three layers. The highest layer is the *application layer*. It implements the core functionality of the software, and its interfaces mostly correspond to high-level phenomena, as they are used in the context diagram. The lowest layer establishes the connection of the software to the outside world. It consists of user interface components and *hardware abstraction layer* (HAL) components, i.e., the driver components establishing the connections to hardware components. The low-level interfaces can mostly be obtained from the technical context diagram. The middle layer consists of *adapter* components that translate low-level signals from the hardware drivers to high-level signals of the application components and vice versa. If the machine sends signals to some hardware, then these signals are contained in a required interface of the application component, connected to an adapter component. If the machine receives signals from some hardware, then these signals are contained in a provided interface of the application component, connected to an adapter component.

The *input* to this step are the implementable architecture, the context diagram, the technical context diagram, and the interaction restrictions. The *output* is a layered architecture. It is annotated with the stereotype ≪layered_architecture≫ to distinguish it from the implementable architecture. Note, however, that a layered architecture can only be defined for a software, i.e., a machine or component with the stereotype ≪local≫, ≪process≫ or ≪task≫. For a distributed machine, a layered architecture will be defined for each local component.

If the implementable architecture is not a distributed architecture, the layered architecture is expressed by a composite structure diagram with the same stereotypes as the implementable architecture (≪machine≫ and, e.g., ≪local≫) and the stereotype ≪layered_architecture≫. To obtain the layered architecture, we assign all components from the implementable architecture to one of the layers. The submachine components will belong to the application layer. Each facade components will be split into smaller parts that can be assigned to the application layer, the adapter layer or the hardware abstraction layer. Some of these parts could be without functionality and left out. As already mentioned, connection stereotypes guide the introduction of new components, namely user interface and driver components. All component interfaces for the newly created parts must be defined. A heuristic for defining the interface of the application layer is, that they should correspond to the interfaces in the context diagram.

If the implementable architecture is a distributed architecture, we have to combine the components of the implementable architecture in new components that represent the local machines that should include a layered architecture. When a functionality should be realized by two or more components, the component that realizes the functionality must be split. In this case, new problem diagrams for split machines and requirements have to be created.

The stereotypes of connectors connecting the external interfaces of the implementable architecture support the developers in their design activities since the application should not access technical interfaces (e.g., gui, network-connection, electrical...) directly. In case of a connector with the stereotype ≪ui≫ or a more specific stereotype (e.g., ≪gui≫), a component implementing a user interface is necessary. This component has to handle the user input and output, coordinate the execution order and provide an interface to the application components with abstract operations necessary to fulfill the requirements. In case of a connector with the stereotype ≪physical≫, ≪electrical≫, ≪network_connection≫, or with a more specific stereotype (e.g., ≪smtp≫), a hardware abstraction layer component and an adapter have to be used. The hardware abstraction layer component is usually re-used in several projects, but the adapter has to transform the information that provided or required by the HAL into abstract abstract operations necessary to fulfill the requirements. It is possible to generate some of the adapter components automatically based on the stereotype, a pattern for the general mapping procedure and a description of the concrete information to be transformed. In case of a connector with the stereotype ≪call_return≫, ≪shared_memory≫, ≪stream≫, ≪event≫ or a more specific stereotype (e.g., ≪api≫) the components are usually able to handle these connectors directly, and no additional component need to be introduced.

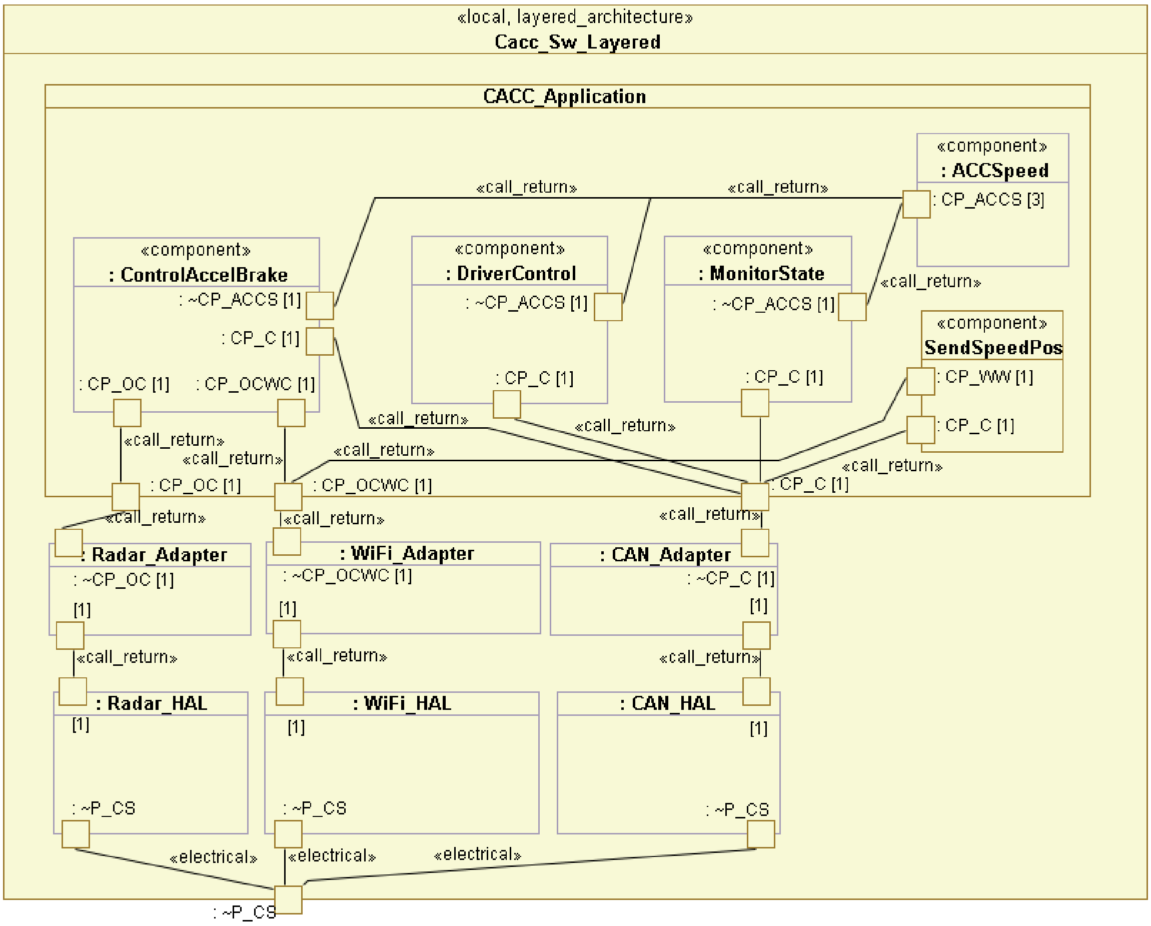


Figure 8.8.: *Layered Architecture of CACC*

The final software architecture of the CACC is given in Figure 8.8. Note that we have split the P_CS_Facade into three adapters and three hardware abstraction layer components to handle the electrical connection to the hardware. Using these adapters and hardware abstraction layer components, information from and to the CAN connection, radar, and WiFi is handled by separate components in order to achieve a separation of concerns.

The validation conditions to be checked for the layered architecture are similar to the validation conditions for the implementable architectures. Condition VM.3 must also hold for the layered architecture, and conditions VM.1 and VM.2 become

VL.1. All components of the implementable architecture must be contained in the layered architecture.

VL.2. The connectors connected to the ports in the layered architecture must have the same stereotypes or more specific ones than in the implementable architecture.

This final step could be carried out in a different way – resulting in a different final architecture – for other types of systems, e.g., when domain-specific languages are used.

8.3. Related Work

Since our approach heavily relies on the use of patterns, our work is related to research on problem frames and architectural styles. However, we are not aware of similar methods that provide such a detailed guidance for developing software architectures, together with the associated validation conditions.

The related work on problem analysis can be found in Section 4.4.

Aiming to integrate problem frames in a formal development process, Choppy and Reggio (2000) show how a formal specification skeleton may be associated with some problem frames. Choppy and Heisel give heuristics for the transition from problem frames to architectural styles. In (Choppy & Heisel, 2003), they give criteria for choosing between architectural styles that could be associated with a given problem frame. In (Choppy & Heisel, 2004), a proposal for the development of information systems is given, using *update* and *query* problem frames. A component-based architecture reflecting the repository architectural style is used for the design and integration of the different system parts.

The approach developed by Hall, Jackson, Laney, Nuseibeh, and Rapanotti (2002); Rapanotti, Hall, Jackson, and Nuseibeh (2004) is quite complementary to ours, since the idea developed there is to introduce architectural concepts into problem frames (introducing "AFrames") so as to benefit from existing architectures. In (Hall et al., 2002), the applicability of problem frames is extended to include domains with existing architectural support, and to allow both for an annotated machine domain, and for annotations to discharge the frame concern. In (Rapanotti et al., 2004), "AFrames" are presented corresponding to the architectural styles Pipe-and-Filter and Model-View-Controller (MVC), and applied to transformation and control problems.

Barroca, Fiadeiro, Jackson, Laney, and Nuseibeh (2004) extend the problem frame approach with *coordination* concepts. This leads to a description of *coordination interfaces* in terms of *services* and *events* together with required properties, and the use of *coordination rules* to describe the machine behavior. Their approach complements our approach and is especially helpful for specifying the behavior of the coordinator component.

Hofmeister, Nord, and Soni (1999) describe software architectures in four views (conceptual, module, execution, and code) with UML and stereotypes. Five industrial architecture design methods are compared in (Hofmeister et al., 2007), and a general approach is extracted where the design activities are the architecture analysis, synthesis (i.e. the core of the design) and evaluation. We may consider that, although our approach is quite different, it complies with these design activities.

Lavazza and Bianco (2006) also represent problem diagrams in a UML notation. They use component diagrams (and not stereotyped class diagrams) to represent domains. Jackson's interfaces are directly transformed into used/required classes (and not ≪observe≫ and ≪control≫ stereotypes that are translated in the architectural phase). In a later paper, Lavazza and Bianco (2008) suggest to enhance problem frames with scenarios and timing.

Hall, Rapanotti, and Jackson (Hall, Rapanotti, & Jackson, 2008) describe a formal approach for transforming requirements into specifications. This specification is then transformed into the detailed specifications of an architecture. We intentionally left out deriving the specification describing the dynamic behavior of the components within this thesis and focus on the static aspects of the architecture.

8.4. Conclusions and Future Work

We have shown how software architectures can be derived in a systematic way from problem descriptions as they are set up during the requirements analysis phase of software development. In particular, our method builds on information that is elicitated when applying (an extension of) the problem frame approach. The method consists of three steps, starting with a simple initial architecture. That architecture is gradually refined, resulting in a final layered architecture. The refinement is guided by patterns and stereotypes. The method is independent of system characteristics – it works e.g., for embedded systems, for web-applications, and for distributed systems as well as for local ones. Its most important advantages are the following:

- The method provides a systematic approach to derive software architectures from problem descriptions. Detailed guidance is given in three concrete steps.
- Problem descriptions and architectural descriptions can both be expressed as UML models. This is possible by defining appropriate UML profiles.
- Because all models are expressed in UML, we have been able to develop a tool for automatically checking the integrity conditions that is based on existing UML tools.
- We have presented a number of semantic integrity conditions that must be fulfilled if our method is applied correctly. These integrity conditions serve to detect any incoherences between the models developed during requirements analysis and architectural design. These conditions can be checked automatically using our tool UML4PF.
- The OCL conditions are quite technical and not easy to read. However, users of UML4PF usually do not need to inspect them. They have to be familiar with our method and must be able to use an editor to set up the different diagrams. Then, they just press the "validator" button. If a condition does not hold, the user is provided with the natural-language description of the condition, and the wrong model element is pointed out to him or her. Therefore, no deep knowledge of OCL and formal methods is necessary to apply our method and use our tool.
- The subproblem structure can be exploited for setting up the architecture.
- Most interfaces can be derived from the problem descriptions.
- Only one model is constructed containing all the different development artifacts. Therefore, traceability between the different models is achieved, and changes propagate to all graphical views of the model.
- Frequently used technologies are taken into account by stereotypes. The stereotype hierarchy can be extended for new developments.
- Stereotypes guide the introduction of new components.
- The application components use high-level phenomena from the application domain. Thus, the application components are independent of the used technology.
- Our methods, integrity conditions, and tool support can easily be extended to cover further diagram types (e.g., sequence diagrams) and development phases, for the example specifying the internal behavior of the architectural components.

An interesting topic for further investigation is, to extend our approach to support the development of design alternatives according to quality requirements and to support software evolution. On the long run, the method can also be extended to cover further phases of the software development lifecycle. For generating adapters automatically, further research and development is necessary. In the next chapter, we extend the approach to handle dependability requirements systematically.

Chapter 9

Development of Architectures for Dependable Systems

Taking dependability requirements into account when developing an architecture is a demanding task, for which satisfactory solutions are still sought for. There are several reasons for this situation. First, dependability requirements must be elicitated, analyzed, and documented as thoroughly as functional ones, which is often not the case. Second, requirements engineering and architectural design must be integrated in such a way that the knowledge gained in the requirements engineering phase is used in a systematic way when developing a software architecture, which cannot be taken for granted. Third, the current techniques for incorporating dependability requirements into architectures are even less developed than the ones that concentrate on functional requirements only.

In this chapter, we want to contribute to improve this situation. We present a method that

- takes dependability requirements into account explicitly,
- is model- and pattern-based, and for which
- tool support exists.

This chapter is based on joint work and is published in (Alebrahim et al., 2011). The published version is about quality requirements, focuses on performance and security requirement, and applies the approach to a Chat System case study. In this chapter, we consider dependability requirements (including security requirements, but no performance requirements) and we apply the approach on the CACC case study. Since dependability requirements are specific quality requirements, the same procedure can be applied. We published first ideas for handling dependability requirement within the architectural design in (Hatebur & Heisel, 2005b), (Lanoix, Hatebur, Heisel, & Souquières, 2007), and (Hatebur et al., 2008a).

As a basis for requirements analysis, we use the approach presented in Chapters 4, 5, and 6. As a basis for architectural design, we use a method we developed for deriving architectures based on functional requirements (see Chapter 8).

In this chapter, we extend our previous requirements analysis and architectural design methods by explicitly taking into account dependability requirements. The analysis documents are extended by dependability requirements that complement functional ones. The so enhanced problem descriptions (see Chapter 5) form the starting point for architectural design. As described in Chapter 8, in a first step, we define an initial software architecture that is oriented on the decomposition of the overall software development problem into subproblems. In a second step, we transform that architecture according to the dependability requirements to be considered. For this purpose, we apply appropriate dependability patterns, or we introduce components providing proven dependability mechanisms, as described in Chapter 6. As described in Chapter 8, we apply functional design patterns (Gamma et al., 1995b), such as *Facade*, to obtain a clean and modular software architecture. Finally, we have defined dependability stereotypes that serve as hints for implementers (e.g., for reliability or confidentiality).

This chapter is organized as follows: Section 9.1 is devoted to describing our method in detail. Related work is discussed in Section 9.2, and conclusions and future work are given in Section 9.3.

9.1. Deriving quality-based Architectures

We first give an overview of our method illustrated in Fig. 9.1 on the next page. In this thesis, we do not consider the question-based catalog. Then we apply our method to the CACC case study described in Section 4.3.

We first decompose the overall problem into subproblems (*Problem Diagrams*), each of which is related to one or more functional requirements (see Chapter 4). Then we annotate each subproblem by complementing functional requirements with related dependability requirements (*Quality Problem Diagrams*, see also Chapters 5 and 6). In the next step we take a design decision concerning the kind of distribution of the software architecture (*Choose Design Alternative*). Then we go back to the requirements descriptions and regarding the design decision split the problem diagrams accordingly (*Split Problem Diagrams*). Analogously to splitting the problem diagrams and so splitting the functional requirements, we also have to split the corresponding dependability requirements (*Split Quality Requirements*). Then we set up an initial architecture by mapping each machine domain in a problem diagram to a component (*Initial Architecture*). After that we elaborate the problem diagrams annotated with dependability requirements by introducing domains reflecting specific solution approaches (*Concretized Quality Problem Diagrams*). In the next step we derive an architecture, which is implementable and addresses the dependability requirements (*Implementable Architecture*). We make use of problem diagrams annotated with dependability requirements and concretized quality problem diagrams. To obtain the implementable architecture, we first merge related components (*Merge Components*). Next, we apply appropriate design patterns (*Apply Design Patterns*). Finally we make use of mechanisms and patterns and the concretized quality problem diagrams (*Apply Quality Mechanisms/Patterns*). We now present the architectural design steps in detail.

Setting up the problem diagrams and quality problem diagrams is described in Chapters 4, 5, and 6. The next step is to choose a design alternative.

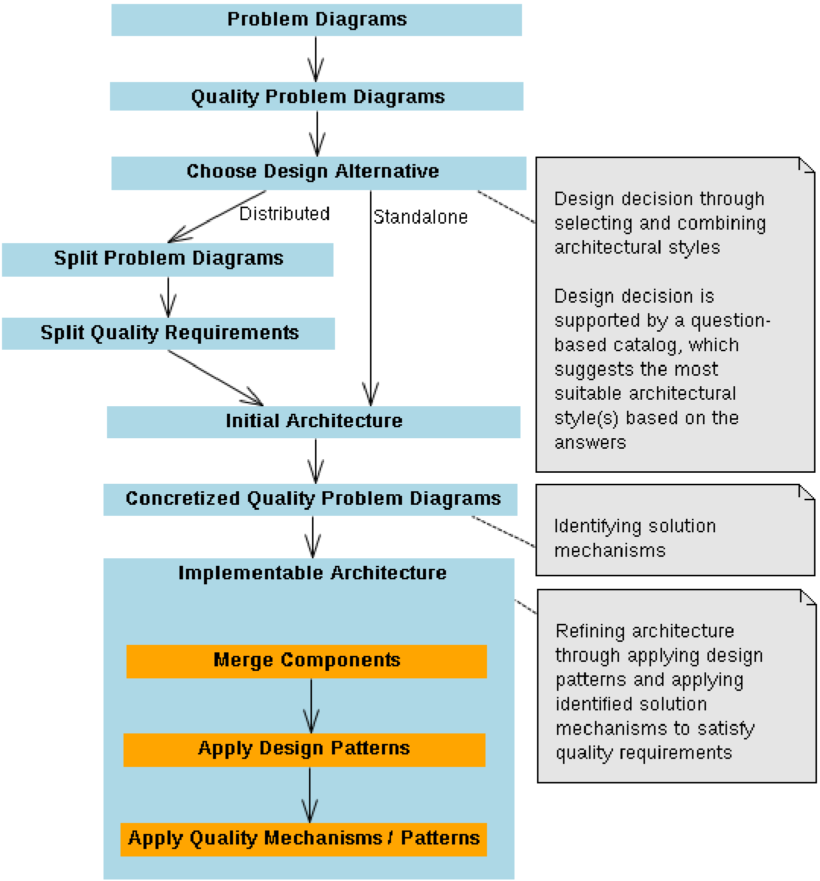
9.1.1. Choose Design Alternative

When we have set up the problem diagrams and have annotated the dependability requirements, we need to take a design decision concerning the kind of distribution of the software to be developed, e.g., client-server, peer-to-peer, or standalone. This decision is either taken by the stakeholder or by the software architect. In this chapter, we do not discuss how this decision is taken.

For standalone applications, we can skip two steps and continue with the step “Initial Architecture”. In the CACC case study, we consider one standalone application that operates in a distributed environment. Therefore, the problems have already been split into subproblems for sending and receiving position and speed. The subproblem for sending is depicted in Fig. 4.24 on Page 48 and for receiving is depicted in Fig. 4.21 on Page 46.

9.1.2. Split Problem Diagrams

After having chosen the architecture for the distributed system, we go back to the requirements descriptions and decompose the problem diagrams in such a way that each subproblem is allocated to only one of the distributed components. This may lead us to introduce connection


Problem Diagrams
Quality Problem Diagrams
Choose Design Alternative
Distributed
Standalone
Split Problem Diagrams
Split Quality Requirements
Initial Architecture
Concretized Quality Problem Diagrams
Implementable Architecture
Merge Components
Apply Design Patterns
Apply Quality Mechanisms / Patterns

Design decision through selecting and combining architectural styles

Design decision is supported by a question-based catalog, which suggests the most suitable architectural style(s) based on the answers

Identifying solution mechanisms

Refining architecture through applying design patterns and applying identified solution mechanisms to satisfy quality requirements

Figure 9.1.: *Method for Derivation of Architectures based on Dependability Requirements (cf. (Alebrahim et al., 2011))*

domains, e.g., networks (see Wifi_Wave in Fig. 4.24 on Page 48).

9.1.3. Split Quality Requirements

Analogously to splitting the problem diagrams and so splitting the functional requirements, we also have to split the corresponding quality and dependability requirements. Several solutions are possible: For example, confidentiality can either be preserved by all subproblems and additional domain knowledge about the newly introduced domains (stating that no information is leaked), or confidentiality can be preserved by sender and receiver and established sending encrypted data.

The quality problem for secret distribution was described in Chapter 6, Fig. 6.3 on Page 95 and also the mechanism was chosen there. Figures 9.2 and 9.3 on the following page depict the corresponding concretized dependability problems with the secret distribution mechanism (and the authentication keys as domains) as described in Fig. 7.7 on Page 117. Each of these subproblems is already assigned to exactly one machine in the technical context diagram and therefore does not need to be split.

The quality problem for checking the integrity considering an attacker was described in Chapter 5, Fig. 5.14 on Page 73. In Chapter 6, the MAC mechanism was chosen. Figure 9.4 on the next page depicts the corresponding concretized quality problem with the considered session keys as domains. The mechanism is described in the second part of Fig. 7.7 on Page 117. The quality problem for creating the signature for the the MAC mechanism was identified in Chapter 6, Figure 6.5 on Page 96. The mechanism is described in the second part of Fig. 7.7 on Page 117. Each of these subproblems is already assigned to exactly one machine in the technical

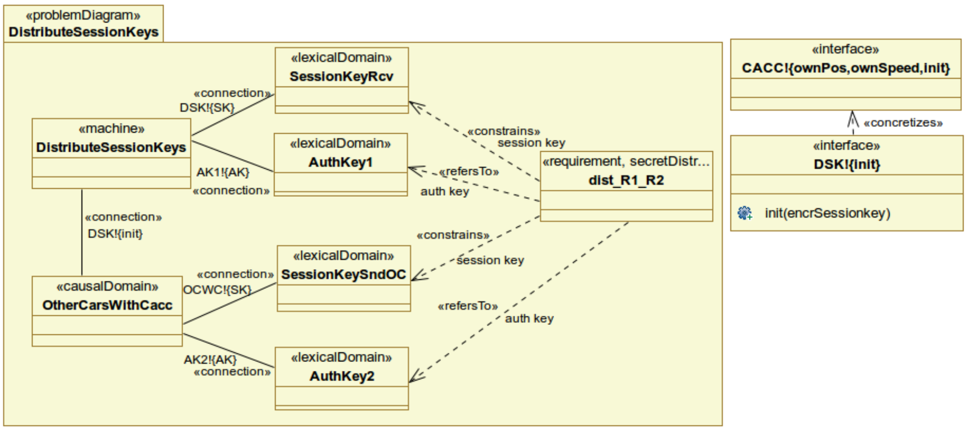


Figure 9.2.: *Concretized Secret Distribution Problem Diagram (with Signing Mechanism)*

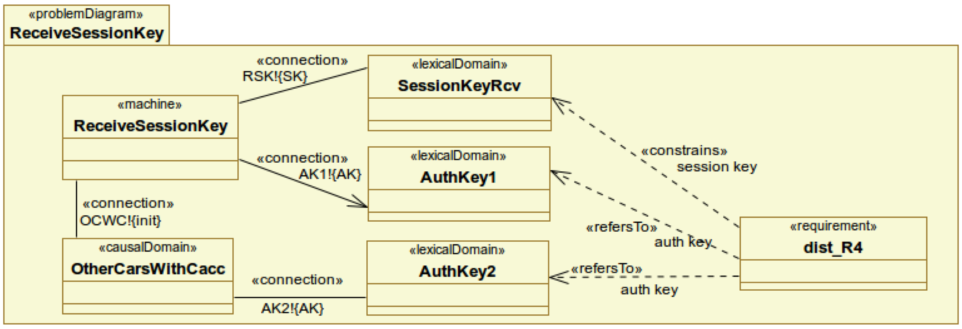


Figure 9.3.: *Concretized Secret Receiving Problem Diagram (with Signing Mechanism)*

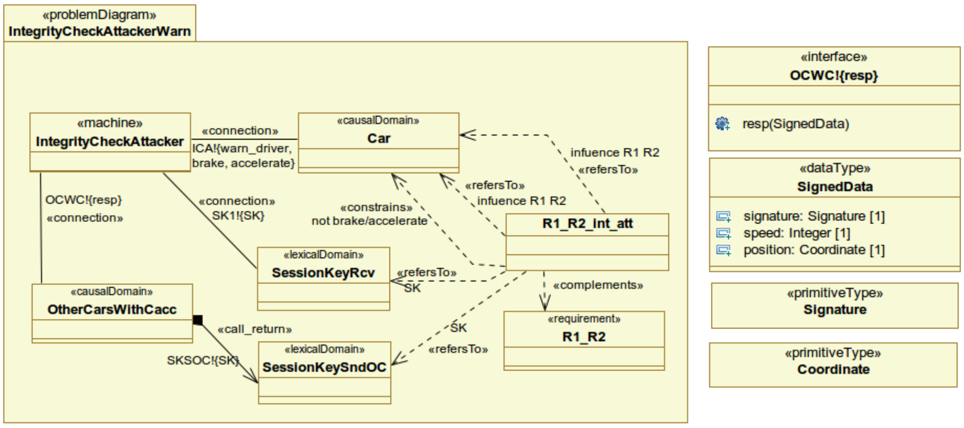


Figure 9.4.: *Concretized Integrity Check Problem Diagram (with Signing Mechanism)*

context diagram and therefore does not need to be split.

9.1.4. Initial Architecture

The initial architecture consists of one component for the overall machine (in our case, *CACC*), with stereotypes ≪machine≫ and ≪initial_architecture≫. In the case of a distributed architecture, we add the stereotype ≪distributed≫ to the architecture component and in case of a standalone system (like the CACC), we add the stereotype ≪local≫. The procedure is the same as described in Chapter 8 but the problem diagrams for dependability are also considered, as depicted in Fig. 9.5.

The machines ControlAccelBrake, DriverControl, MonitorState, and SendSpeedPos in the subproblem describing functional requirements are represented as components. All lexical domains AccSpeed, SessionKeySnd, SessionKeyRcv, and AuthKey1 are also considered to be part of the machine. The machines CalcSignature, ReceiveSessionKey, IntegrityCheckAttacker, DistributeSessionKey, and IntegrityCheckRandom in the dependability subproblems are additionally represented as components.

Compared to Fig. 8.4 on Page 125 in Chapter 8, we additionally have

- the component CalcSignature,
- the component ReceiveSessionKey,
- the component IntegrityCheckAttacker,
- the component DistributeSessionKey, and
- the component IntegrityCheckRandom.

9.1.5. Concretized Quality Problem Diagrams

The goal of this step is to find solution approaches in terms of mechanisms and patterns to prepare for solving the given dependability problems. We have given examples of such solutions in Chapter 6 and depict the missing problem diagrams in Section 9.1.3. We elaborate the problem diagrams annotated with quality requirements by introducing domains reflecting specific solution approaches. We call the elaborated problem diagrams containing solution approaches *concretized quality problem diagrams*.

9.1.6. Implementable Architecture

The purpose of this step is to derive an architecture, which is implementable and fulfills the performance and security requirements.

9.1.6.1. Merge Components

Related components that realize a similar functionality and contain at least one similar domain in their problem diagrams can be merged to one component. In the CACC case study, we merged the components IntegrityCheckAttacker, DistributeSessionKey, and SessionKeyRcv into the component ControlAccelBrake. We also merged the components ReceiveSessionKey, CAlcSignature, and SessionKeySnd into the component SendSpeedPos.

In general, the decision about the merging of components should be taken by an experienced architect.

9.1.6.2. Apply Design Patterns and Quality Patterns

We introduce a *Facade* component (Gamma et al., 1995b), if several internal components are connected to one external interface in the initial architecture. As *Facade* components, we introduce the P_CS_Facade component and the P_CS_HWF component in order to prevent each single component from communicating with the hardware directly.

If interaction restrictions have to be taken into account, i.e., actions have to happen in a certain order, we have to add one or more *Coordinator* components. In our CACC case study, The session keys have to be distributed before the communication can be established. Therefore, the

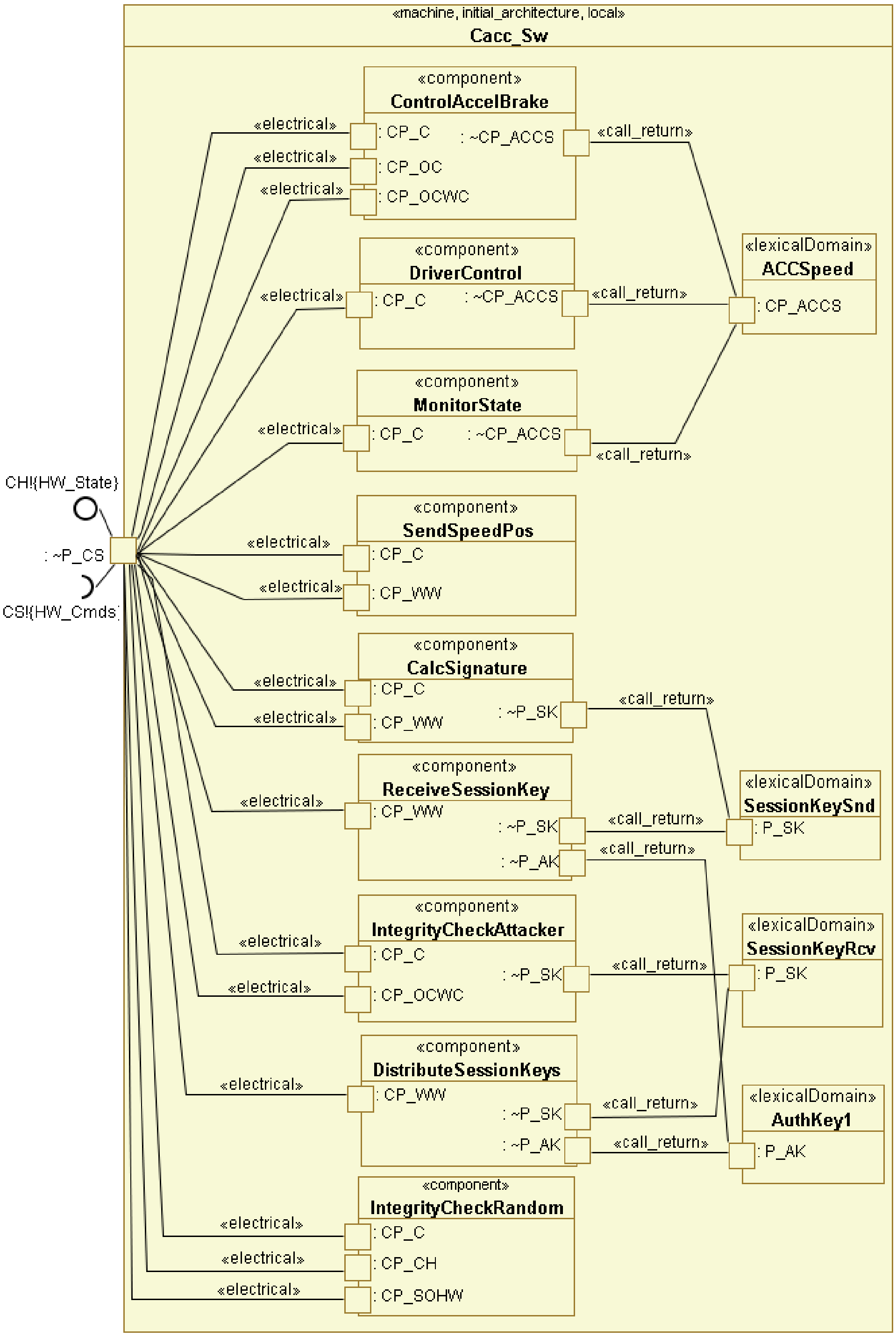


Figure 9.5.: *Initial Architecture with Dependability Components*

components RcvFacadeCtrl and SndFacade are introduced. They also perform the functionality described in subproblem PD1 (see Fig. 4.21 on Page 46) and PD4 (see Fig. 4.24 on Page 48).

If a redundancy is selected as a generic mechanism in the analysis phase, the machine has to be split into a certain number of independent channels that all can perform the critical functionality, monitors that may be integrated into these channels, and a kind of voter component that ensures that a correct output is given to the actuators. Usually, each channel is build with an individual hardware (microprocessor, memory). The monitors have to check the hardware of the individual channels, check input signals, check output, may check other channels, and support the voter component. In our second case study in Figures 13.21 on Page 199 and 13.23 on Page 201), we present an instance of such a pattern.

The implementable architecture after applying these patterns is shown in Fig. 9.6 on the following page. Compared to Fig. 8.6 on Page 130 in Chapter 8, we additionally have the following components

- the component IntegrityCheckRandom (as in the initial architecture),
- the facade component P_CS_HWF,
- the storage for the authentication key (AuthKey1) that is used by the modified components SendSpeedPos and ControlAccelBrake from the initial architecture.

The component SendSpeedPos contains the component SndFacade with a coordinator, the component ReceiveSessionKey, the component CAlcSignature, and the storage for the SessionKeySnd. The component ControlAccelBrake contains the component RcvFacadeCtrl with a coordinator, the component IntegrityCheckAttacker, the component DistributeSessionKey, and the storage for the SessionKeyRcv. The facade component P_CS_Facade has an additional port connected component IntegrityCheckRandom. This port is used to send error messages.

9.1.7. Layered Architecture

The layered architecture can be designed in the same way as described in Chapter 8. The layered architecture including all components realizing the mechanisms for dependability is depicted in Fig. 9.7 on the following page. In this architecture the AuthKey1 component and the IntegrityCheckRandom component from the implementable architecture are included in the application layer. The facade component P_CS_HWF is split into a part for switching off the CACC (SO_HAL) and a part checking the hardware integrity (e.g. RAM and ROM checks) and assigned to the hardware abstraction layer. Compared to Fig. 8.8 on Page 134 in Chapter 8, we additionally have the following components

- the application component IntegrityCheckRandom (as in the implementable architecture),
- the hardware abstraction layer component SO_HAL,
- the hardware abstraction layer component Check_HAL, and
- the storage for the authentication key (AuthKey1) that is used by the application components SendSpeedPos and ControlAccelBrake from the implementable architecture.

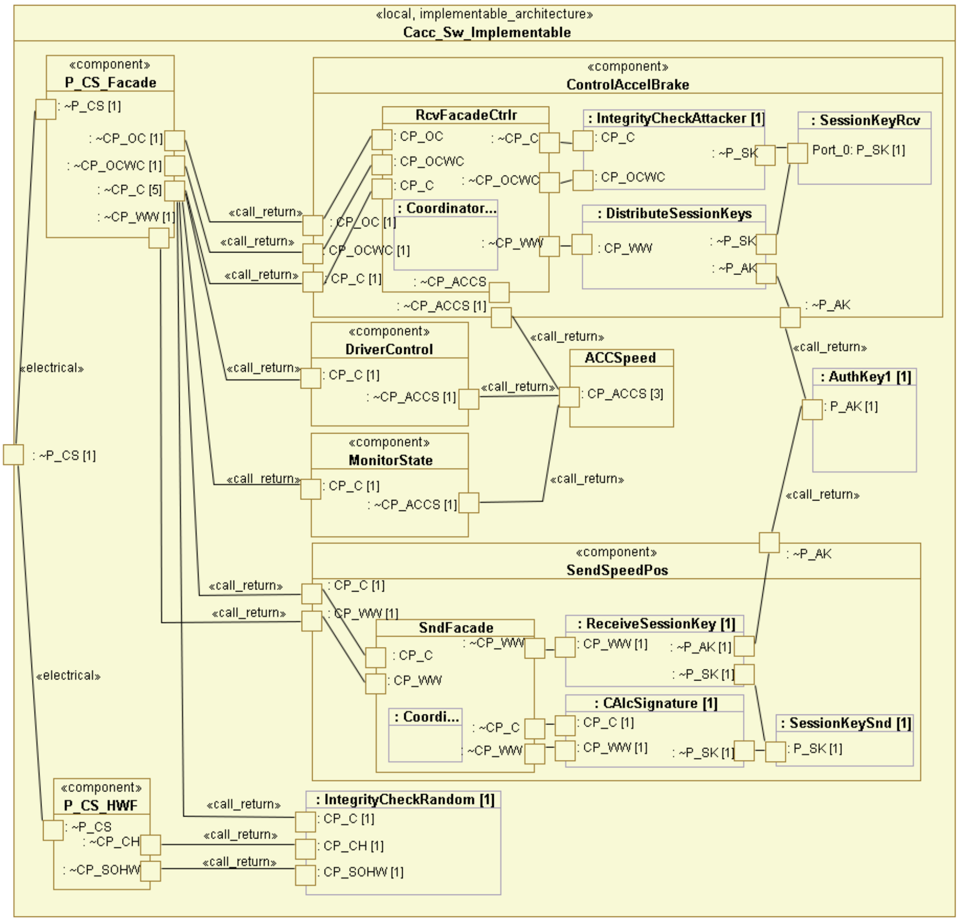


Figure 9.6.: *Implementable Architecture of CACC for Dependability*

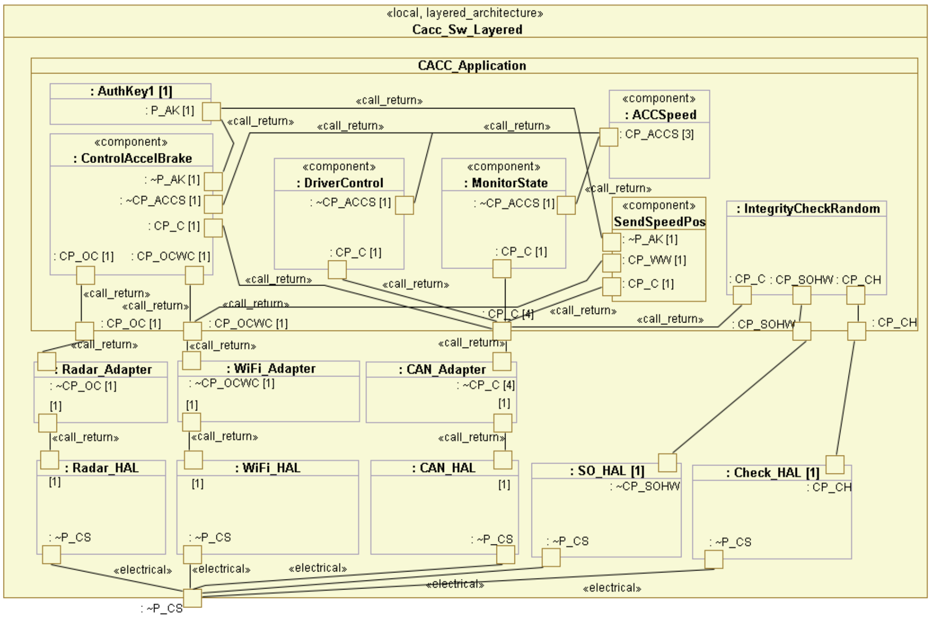


Figure 9.7.: *Layered Architecture of CACC for Dependability*

9.2. Related Work

Consideration of software quality during the software development process, especially in the requirement analysis phase, is still a challenging research problem. There are approaches that deal with only one type of quality requirement, e.g., security.

An approach to transform security requirements to design is provided by Mouratidis and Jürjens (2010b). It starts with the goal-oriented security requirements engineering approach Secure Tropos (Mouratidis, 2004b), and connects it with a model-based security engineering approach, namely UMLsec (Jürjens, 2005). UMLsec is a UML profile for representing security properties in UML diagrams. It does not provide support for the analysis phase of the software development process. Thus a seamless integration of requirements analysis and architectural design is not supported by UMLsec.

Schmidt and Wentzlaff Schmidt and Wentzlaff (2006) develop architectures from requirements based on the problem frame approach, taking into account usability and security. By way of an example, they show how to balance security and usability requirements.

Attribute Driven Design (ADD) (Wojcik et al., 2006) is a method to design a conceptual architecture. It focuses on the high-level design of an architecture, and hence does not support detailed design. Identifying mechanisms to achieve quality attributes relies on the architect's expertise.

Q-ImPrESS (Becker, Dešić, et al., 2009) is a project that focuses on the generation and evaluation of architectures according to quality properties, in particular performance. The phases design and implementation of the software development process are particularly in focus. In contrast to our contribution, it does not use requirements descriptions as a starting point.

The notation and evaluation of performance attributes of an architecture is the focus of the component model Palladio (Becker, Koziolek, & Reussner, 2009), which is also included in the project Q-ImPrESS. In Palladio, a set of notations, concepts and a tool are provided, which allow its users to model and simulate architectures for performance evaluation. The tool could be used for simulating and thus evaluating software architecture performance. The concepts and the included tool, however, cannot be used to evaluate an architecture's dependability.

9.3. Conclusion

In this chapter, we have presented a detailed, UML-based and tool-supported method to derive software architectures from requirements documents, thereby taking dependability requirements into account. Our method addresses all the problems we identified in the introduction:

- We achieve a seamless transition from requirements analysis to architectural design. The two phases are not separated, but intertwined. An architectural decision drives the revision of problem descriptions, and concretized problem descriptions lead directly to architectural components and connections.

- dependability requirements are explicitly considered. Our method builds on established approaches to achieve dependability properties, such as encryption or redundancy. The application of these mechanisms or patterns is directly visible in the software architecture.

Our method is based om problem frames, and known dependability patterns. Its novelty is the fact that the different approaches are integrated and intertwined explicitly by an underlying methodology and a common notation. The notation as well as the methodology are open and can be developed further to enhance the power and breadth of the approach.

Our approach is limited on structural descriptions of software architectures. It is interesting to investigate, how to extend our method to also support deriving behavioral descriptions for the developed architectures and automatically checking their coherence with the structural descriptions. It is also possible to give rules for selecting appropriate design alternatives, evaluate the approach for different dependability requirements and describe additional patterns for architectural design.

Chapter 10

Implementing Dependable Systems

In this chapter, we present a methodology to implement an architecture with Java considering dependability requirements.

In Section 10.1, we present a method for implementing components in Java without using a component framework. The components implemented with this method are separately testable, but cannot be compiled and linked separately. This method is emerged from different industrial projects. We are not aware on any publication describing this method. Documenting and refining this method is joint work and only published within the lecture note of different lectures (Heisel & Hatebur, 2008; Heisel, 2011). This section describes the foundation to implement dependable software. In Section 10.2, we present rules for developing secure software. It just summarizes the content of the Common Criteria (International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC), 2009a). In Section 10.3, we present rules for developing safe software. It addresses the dependability requirements considering integrity, availability, and reliability considering random faults and also tries to prevent systematic faults. The rules are derived from the rules in the ISO/IEC 61508 (International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC), 2000) and adjusted for object-oriented software. The presented results are based on preliminary results of joint work with the OOSE subgroup of EWICS (Bitsch et al., 2011). No dedicated section discussing related work is given since the whole chapter only describes related work. In Section 10.4, we give a summary of this chapter and point out directions for future work.

10.1. Implementation of Components

In this section, we describe a methods for structuring object-oriented software systems into (white-box) components. Other methods for implementing components are e.g., OSGi (Wütherich, Hartmann, Kolb, & Lübken, 2008) or Enterprise JavaBeans (EJB) (Burke & Monson-Haefel, 2006). We describe a method for implementing architectures in this section, since it is supports the dependability and is applied commonly in industry but we are not aware of any publication. The advantage of such an implementation is, that it supports the testability by explicit required interfaces and supports the localization of implemented dependability features in the source code.

Within the implementation of the machine, the structure of the software may be not given explicitly. In the source code and in UML models, we usually have associations between classes. These associations can be either references to other components, or the referenced objects are part of one component. For the following code, the class diagram is depicted in Fig. 10.1. This class diagram may implement different architectures:

- One design alternative that is implemented could be a component typed by ClassA containing components typed by ClassB and ClassC. Both components ClassA and ClassB provide the interface InterfaceI. The component ClassA implements the interface by providing the implementation of component ClassB.

- Alternatively, the component ClassC could be external with respect to the component ClassA.
- It is also possible, that all classes represent separate components. In this case, the component ClassA uses the components ClassB and ClassC and both components ClassA and ClassB provide the interface InterfaceI with individual implementations.

```
class ClassA implements InterfaceI{
  private ClassB b;
  private ClassC c;
}

class ClassB implements InterfaceI\{
  private ClassC c;
}

class ClassC {
...
}
```

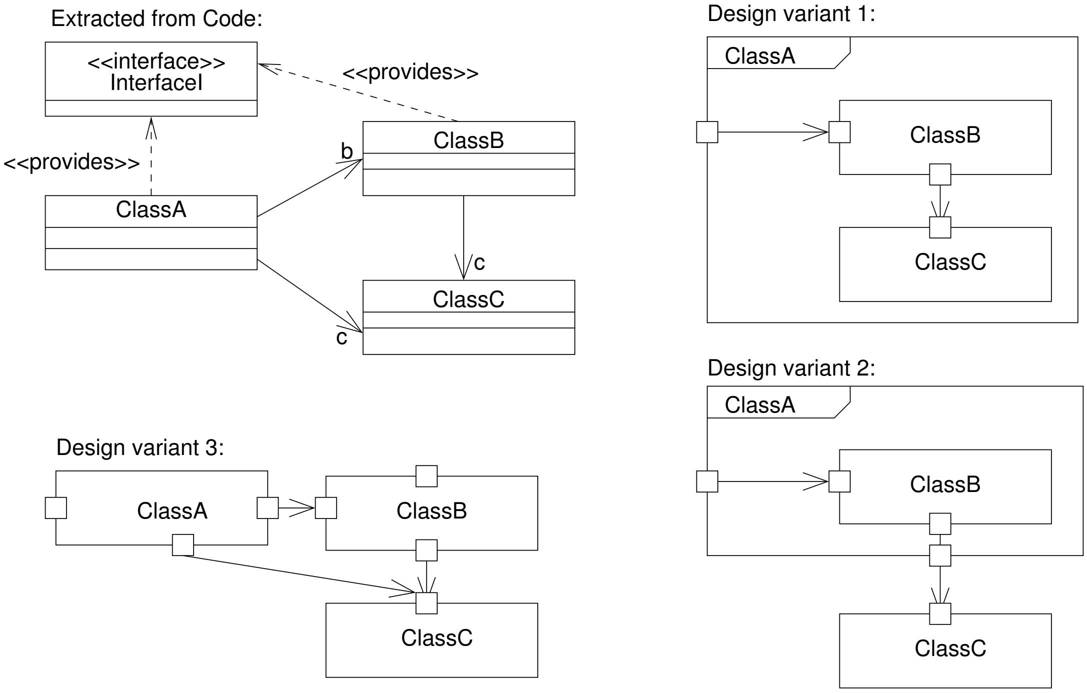


Figure 10.1.: *Translation from Class Diagram to Composite Structure Diagram*

Note that not all objects (typed by a class) can be clearly associated to a certain component: some objects are used to exchange complex data between components and passed as parameters, e.g. a user object is created in the user interface component and sent to the application component for further processing. In Fig. 10.2 an example with address data to be exchanged is shown. In the class diagram on the left-hand side, the class Address has an association to both components Class1PartOfA and Class2PartOfA. This class represents a data type used as a parameter in the interface shown below the composite structure diagram on the right hand side.

Several component coupling levels exist:

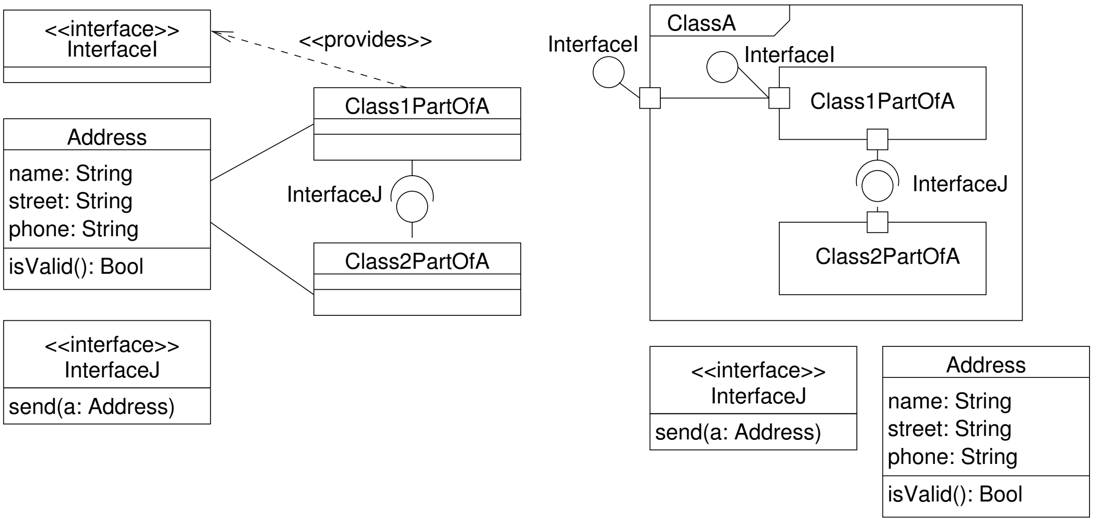


Figure 10.2.: *Translation from Class Diagram to Composite Structure Diagram – Data*

- A component may be just an object of a class. In this case, it may use other objects to provide its functionality, and the public operations of the class represent the component. But it is not clear which of the objects used to provide the functionality (associated) are part of that component, and which are not. When analyzing software consisting of such components, other objects created by this object can be considered to be part of the component. Usually components of this type are not built separately.

- A component may be an object with explicit provided interfaces. In this case, it may also use other object to provide its functionality. It is still not clear which of the objects used to provide the functionality are part of that component. An advantage is that the implementation can be better replaced. Usually components of this type are also not built separately. If this coupling level is used, the notations depicted in Fig. 10.3 are appropriate. This coupling level is also used by Cheesman and Daniels (2001).

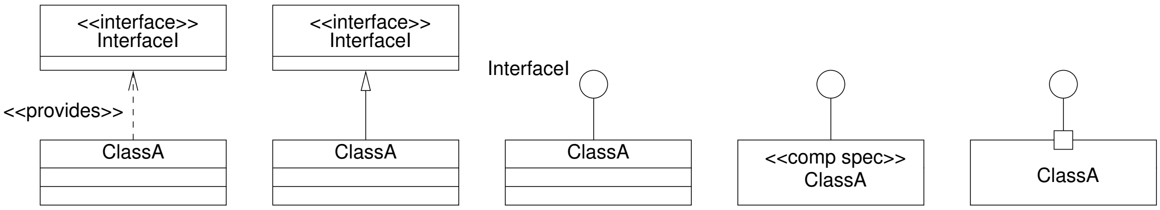


Figure 10.3.: *Notation for Class with Provided Interface / Lollipop Notation / Component According Cheesman, Daniels 2001*

- A component may be an object with explicit provided and required interfaces. This kind implements a loose coupling, i.e., other components used to provide the functionality are connected during instantiation or initialization. An advantage is that the components can be easily tested separately. Other object used to provide the functionality may be created, and they are considered to be part of the component. If this coupling level is used, the notations depicted in Fig. 10.4 is appropriate. This coupling level is also used in Fig. 10.2.

- A component may be object with explicit provided and required interfaces that makes use of a component standard (e.g., providing events or messages) to communicate with other

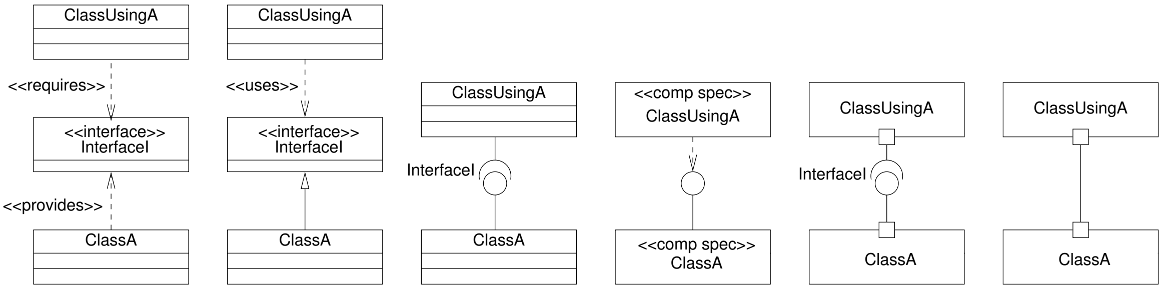


Figure 10.4.: *Notation for 2 Connected Classes / Lollipop Notation / Component according Cheesman and Daniels (2001) / Composite Structure*

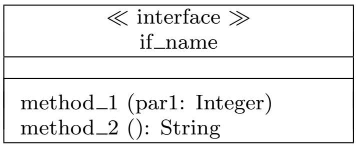
components. This is also a loose coupling approach and the components are connected at run-time with the advantage that the components can be easily tested separately. In contrast to the coupling levels above, usually these components can be built separately. The same notation as for objects with explicit provided and required interfaces can be used.

- It is also possible that all components are separate processes that communicate using events or messages. The component have the same properties as the components above, but a crashing component here may not affect the other components.

The following paragraphs describes an implementation method for components in Java with explicit provided and required interfaces. This implementation method is illustrated on a small example of a stereo. As partly mentioned above, the implementation has the following properties:

- A component has provided and required interfaces that can be connected with other components.
- A component only uses functionality from its required interfaces, from the programming language, and a limited set of operations of the operating system (e.g., tasks, threads, memory allocation, timers, messages, synchronization mechanisms).
- Provided and required interfaces are represented by interface classes.
- Interface operations are called synchronously.
- Advantage: These classes / components can be easily tested separately.

For such an implementation, first, the interface of the architecture have to translated into Java interfaces as shown for the following example:



```
package project_name;
public interface if_name {
    public void method_1 (int par1);
    public String method_2 ();
}
```

The interfaces in the architecture can be directly represented as Java interfaces. Note that the project name should be added as a package for small projects and note that the type int is a simple data type, whereas the type String is a class.

For the architecture given in Fig. 10.5, each provided interface is defined as an interface class, e.g.:

```
public interface LineInOut {
   public void transmitMusic();
```

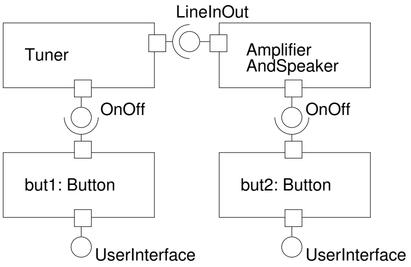


Figure 10.5.: *Stereo Example Architecture*

```
}

public interface OnOff {
   public void pressed();
}
```

A component can implement / provide several interfaces. For each interface, all provided operations have to be implemented as methods, e.g. the AmplifierAndSpeaker implements the interfaces LineInOut and OnOff with the operations transmitMusic() and pressed():

```
public class AmplifierAndSpeaker implements
                           LineInOut, OnOff  {
   public AmplifierAndSpeaker (){} //constructor

   public void transmitMusic() { Play;}
   public void pressed() { Action2;}
}
```

A component can use / require several interfaces, defined as interface classes. Required interfaces are implemented in Java as shown in the following example:

```
public class Tuner implements OnOff {
   private LineInOut outputDevice;
   private boolean isOn;

   public Tuner(){ outputDevice = null; isOn = false; }

   public void connectTo(LineInOut par) {outputDevice = par;}

   public void pressed() {
     isOn = !isOn;
     if (isOn && outputDevice!=null) {
        outputDevice.transmitMusic();
     }
   }
}
```

Required interfaces of a component become private attributes. In the example, the Tuner component has a private attributes outputDevice of type LineInOut. The component has to provide methods to connect the component to the required components (connectTo). In these connect methods, the private attributes are initialized. Via these private attributes, the connected components can be used. They should only be used if they are initialized (if (outputDevice!=null) ...). Alternatively, it is possible to leave out the method connectTo and initialize the connected interface in the constructor. The component Tuner also provides the interface OnOff and implements the method pressed.

In the example, the component Button forwards buttonPressed from UserInterface to the connected device. It requires the interface OnOff:

```
public class Button implements UserInterface {
   private OnOff connectedDevice;

   public Button(){ connectedDevice=null }

   public void connectTo(OnOff conDev) {
        connectedDevice = conDev;
   }

   public void buttonPressed() {
     connectedDevice.pressed();
   }
}
```

The components but1, but2, myTuner, and myAmp can be connected as follows:

```
AmplifierAndSpeaker myAmp = new AmplifierAndSpeaker();
Tuner myTuner = new Tuner();
Button but1 = new Button();
Button but2 = new Button()
myTuner.connectTo(myAmp);
but2.connectTo(myAmp);
but1.connectTo(myTuner);
```

10.2. Implementation of Secure Software

The Common Criteria from International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC) (2009a) defines some rules for implementation in order to "determine that the implementation representation made available by the developer is suitable for use in other analysis activities. Suitability is judged by its conformance to the requirements for this component."

The following aspects are required by the Common Criteria (International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC), 2009a).

- Any programming language used must be well defined with an unambiguous definition of all statements, as well as the compiler options used to generate the object code.

- The mapping between the Target of Evaluation (TOE) [1] design description and the sample of the implementation representation shall demonstrate their correspondence. This mapping can be supported by using the method described in Section 10.1.

- The implementation representation shall be internally consistent. For example, if one portion of the source code includes a call to a subprogram in another portion, the arguments of the calling program must match the called program's handling of the arguments.

- The implementation representation shall accurately instantiate the TOE security functional requirements.

- The part of the implementation that do not implement any TOE security functional requirements should not interfere with the portions that do.

[1] In the Common Criteria the machine is called *Target of Evaluation (TOE)*.

10.3. Implementation of Safe and Reliable Software

The following rules are considered by Bitsch et al. (2011) to be important for developing object-oriented software for safety-related system with object-oriented programming languages. Following these rules also supports the reliability of the software part of the system. The rules refer to the safety integrity level (SIL) defined in ISO/IEC 61508 (International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC), 2000). SIL 1 is the lowest SIL and SIL 4 is the highest SIL. The level depends on the necessary failure reduction for the functionality. The rules are also consider the rules given in the ISO/IEC 61508 International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC) (2000, Part 7, Annex G).

- Programming shall be essentially against the fixed interfaces of the neighboring classes. Using fixed interfaces of the neighboring classes facilitate understanding of the code. This rule is recommend for all safety integrity levels (SILs). (See Section 10.1 for one possible solution approach.)

- Overriding of methods shall be used only, if understanding the code is thereby simplified. This is reached by keeping the essential meaning of the related operation. This rule is recommend for SIL 1 and highly recommend for higher SILs.

- Visibility of attributes (variables) from outside shall be as much restricted as possible. This rule is highly recommend for all SILs.

- Inheritance shall only be used, if the derived class is a specialization of its basic class. All other use of inheritance would make it more difficult to understand and to maintain the code. In case of need another class or class hierarchy should be used. Deep inheritance entails poor testability and difficulties in understanding. This rule is recommend for SIL 1 and highly recommend for higher SILs.

- No automated type conversion shall be used. Automated type conversions have sometimes entailed run-time failures. This rule is recommend for SIL 1 and 2 and highly recommend for higher SILs.

- Side effects to other inheritance hierarchies shall be avoided. All such side effect shall be documented. This rule is recommend for SIL 1 and highly recommend for higher SILs.

- The interface of any class shall be complete and minimal. This rule is recommend for all SILs and highly recommend for SIL 4.

- Recursion shall be avoided. This rule is highly recommend for all SILs.

- Temporary objects shall be banned. The necessary objects should rather be instantiated at program start. Safety-related software should work with static binding in the first place. This rule is recommend for SIL 1 and 2 and highly recommend for higher SILs.

- If temporary objects are not banned, the number of temporary objects shall be minimized. This rule is highly recommend for SIL 1 and 2 and not applicable for higher SILs since temporary objects are not allowed for SIL 3 and SIL 4.

- If temporary objects are not banned, an object that is no longer used, shall be deleted immediately after its last usage. Some safety-related programs can rest for years in the equipment, they are controlling without any humanly executed maintenance. Gathering of unused objects over time in such a program can have devastating effects. This rule is mandatory for all safety levels and not applicable for SIL 3 and SIL 4 since dynamic memory allocation shall be not allowed for SIL 3 and SIL 4.

- It shall be visible at each object and each variable whether or not it can be affected by another thread. This rule is recommend for SIL 1 and highly recommend for higher SILs.
- All error information that is returned by a method shall be processed. This rule is highly recommend for all SILs.
- Small concrete parameters shall be passed by value. This rule is recommend for all SILs and highly recommend for SIL 4.
- Assertions shall be used, where appropriate to document invariants, preconditions, and postconditions. Assertions can be used during development to test invariants, preconditions, and postconditions. This rule is recommend for all SILs.
- Preconditions shall be as weak as allowed in view of the considered functionality. This rule is recommend for all SILs.
- Postconditions shall be as strong as allowed in view of the considered functionality. This rule is recommend for all SILs.

10.4. Conclusions and Future Work

In this chapter, we have shown how software components can be implemented in a systematic way and we summarized rules for implementing secure and safe software.

The method for implementing software components is limited to Java and does not have direct impact to the dependability of the software. The method can be easily extended to support other object-oriented languages. A well-structured software (as derived by this method) supports the localization of functionality implemented to realized dependability requirements.

The rules for security software realization just describes the state of the art for security software.

The rules for safe and reliable software are dedicated to object-oriented software. These rules here are not limited to a certain programming language. Rules for non-object-oriented software are given in the ISO/IEC 61508 (International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC), 2000). The presented rules address the avoidance of systematic faults.

It is interesting to elaborate a more comprehensive set of rules from different standards addressing all dependability requirement types.

Chapter 11

Testing of Dependable Systems

Model-based software development proceeds by setting up *models* of the software to be constructed. This approach has proven useful, because it allows developers to first elaborate the most important properties of the software before proceeding with the implementation. Often, software models are also used for code generation. In this case, however, a problem arises: it does not make sense any more to test the software against its models, because these were already used to generate it. We therefore propose to test the software not only against its *specification* (i.e., against the models), but also against its *requirements*, which describe the how the *environment* should behave in which the software will be operating (acceptance testing). For this purpose, we have to set up a model of the environment, too.

In this chapter, we describe how UML state machines (with a corresponding support tool Teager (Santen & Seifert, 2006)) can be used to realize the described approach in the area of reactive and/or embedded systems. For this kind of system, state machine models are particularly useful. We elaborate on two different testing approaches:

On-the-fly testing: Here, generating and executing test cases is intertwined. This has the advantage that state explosion is not a problem, but the disadvantage that for non-deterministic systems the tests may not be repeatable.

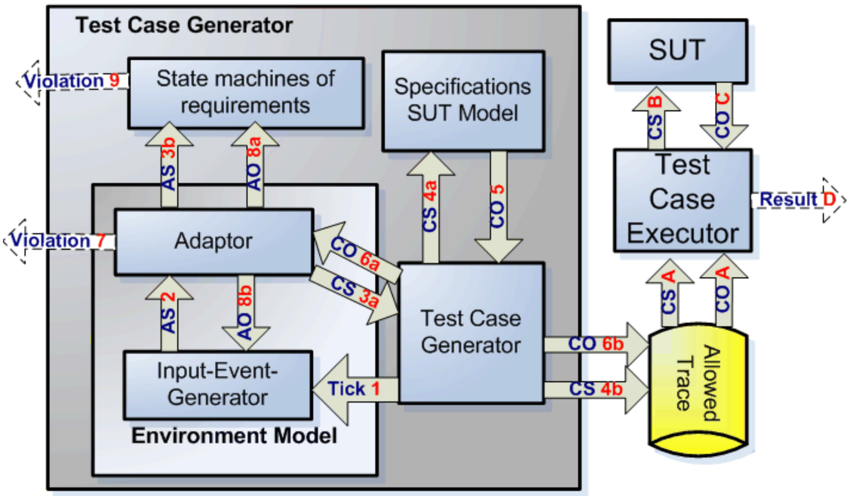
Batch testing: Here, test cases are generated and stored for later execution. This has the advantage that regression tests become possible but the disadvantage that all possible behavior variants must be computed.

This chapter is based on Heisel, Hatebur, Santen, and Seifert (2008b) and Heisel, Hatebur, Santen, and Seifert (2008a). The approach has been verified and improved by Sun (2008), Naveed (2010), and Mohammadi (2010) in their master thesis documentations.

In Section 11.1, we describe our approach for automatic test case generation using UML-Models. Section 11.2 contains patterns for modeling the requirements and the environment for testing. In Section 11.3, we describe the tool support for automatic test case generation using UML-Models. The models for the CACC case study are presented in Section 11.4. Section 11.5 discusses related work, and in Section 11.6, we give a summary of our achievements and point out directions for future work.

11.1. Automatic Test Case Generation using UML-Models

To test the CACC system (SUT, system under test), usually, conformance with the specification would be checked. However, if the specification was not correctly derived from the requirements, the SUT would pass the test nevertheless. We therefore propose to test the SUT against the requirements. We showed in Section 7.1 how the specification is derived from the requirements. Besides detecting errors made in transforming requirements into specifications, testing against requirements allows us to verify that customer needs are satisfied (acceptance test). In order to test the SUT against its requirements, we need a model of the environment, because the requirements refer to the environment and not to the machine. Much like the SUT, the environment can be modeled using UML state machines. The model explicitly contains the facts and the assumptions about the environment. The environment model consists of *adaptors*[1] and the *input event generator*: Adaptors transform abstract events such as *decelerate car* into concrete ones, such as *send braking CAN message*. The input event generator produces abstract events. To capture stochastic properties of the environment probabilistic state machines of TEAGER can be used.



CO: Concrete Observation AO: Abstract Observation
CS: Concrete Stimulus AS: Abstract Stimulus
tick: Request for new Stimulus Violation: Test Result

Figure 11.1.: *Architecture of* Batch *testing*

The requirements are translated into state machines, too. These state machines serve to inform the tester when a requirement is violated. They observe the stimuli and SUT outputs at an adequate level of abstraction. As shown in Fig. 11.1, the Test Case Generator component of the tool TEAGER can be used to simulate the environment model and to check the requirements. To calculate test cases, for each tick (1) an abstract stimulus (2) is generated by the Input-Event-Generator in the environment model. Adaptors transform the abstract stimuli into concrete stimuli for the Test Case Generator (3a) and send the abstract stimuli to the State machines of requirements (3b). The Test Case Generator sends the concrete stimuli to the Specifications SUT Model (4a), which determines suitable responses (5), and it stores the concrete stimuli (4b) and the determined concrete observations (6b). The Adaptors transform the concrete observations (6a) into abstract observations that are checked by the State machines of requirements (8a) and used to generate reasonable stimuli (8b). Violations can be detected by the State machines of requirements (9) and by the Adaptors while transforming concrete observations into abstract ones (7). After the requirements are checked, a new tick (1) is generated. The concrete stimuli (4b) and the concrete observations (6b) form the Allowed Traces that can be used for testing.

The generated test cases (Allowed Traces) can be used to test the SUT with the Test Case Executer. Concrete stimuli and observations in the allowed traces (A) are used to stimulate the SUT (B) and check the responses (C). Test results (D) are the output of the Test Case Executer.

Alternatively, the environment model can be directly connected to the SUT, and within the simulated environment the requirements are checked at runtime. In this case no SUT model is necessary. This scenario is especially useful for acceptance tests. The test system architecture – annotated with sample observations and stimuli and with the execution order – for this "on the fly"-testing approach is shown in Fig. 11.2.

[1] We use "adaptor" for the environment models and "adapter" for the machine models.

figures/on-the-fly.pdf

CO: Concrete Observation AO: Abstract Observation
CS: Concrete Stimulus AS: Abstract Stimulus
tick: Request for new Stimulus Violation: Test Result

Figure 11.2.: *Architecture of* On-the-fly *testing*

11.2. Patterns for Modeling Requirements and Environment for Testing

Setting up the state machines for the environment model is not a trivial task. However, we can identify different patterns for setting up environment models, especially for expressing requirements (R_i) as state machines. The overall structure of the state machine consists of parallel regions. That is, the environment model is in all of the parallel machines R_i, Input-Event-Generator and Adaptor at the same time, and the different sub-machines communicate with each other via common events. For the Input-Event-Generators and the Adaptors it is hard to identify patterns. A general rule is that for each domain in the environment controlling a phenomenon observed by the SUT, an Input-Event-Generator has to be created and that all concrete phenomena between machine and environment have to be transformed into abstract ones (or vice versa) by adaptors. Figure 11.3 on Page 159 shows an example of an input event generator. The Input-Event-Generator is separated into 3 parts that work in parallel.

- The first Input-Event-Generator serves to calculate a speed value. It simulates in a very simplified way, how the speed is derived from the position of the accelerate and brake pedal position or the corresponding messages that can also be generated by the CACC. For calculation, it takes into account that the car can accelerate and decelerate. In the accelerate state, the speed of the vehicle (vspeed, measured in kilometers per hour) is increased or decreased stepwise to the speed given by the accelerate pedal position (apos) combined with a scale_factor. The scale_factor is modeled a constant, but in reality depends on the selected gear, the speed and the road. In the decelerate state, the speed is decrease by the brake pedal position value that is between 0 and 20. It is possible to add in this state machine probabilistic values that e.g. simulates hills and for road surface.
- The second Input-Event-Generator simulates a driver with his/her actions. The driver can either do normal driving or let the CACC adjust the speed of the car in an automatic

mode. While normal driving the driver may press the brake or the accelerate pedal. When the CACC sends no messages (while normal driving), the engine actuator receives the brake pedal position directly. This is modeled with the effects accelerate and brake. All transitions are triggered automatically according to the probabilities we assigned. We decided that with a probability of 40 %, the accelerate_pedal is pressed, with a probability of 20 % each of the other transitions should be taken. In general, the probabilities can be chosen according to the reality of to reach a certain test criteria. With resume or set_speed the driver enters the automatic mode. In this mode, it is possible to increase or decrease the desired speed using the dedicated buttons. In this mode, the CACC will measure the distance to the car ahead. The distance measurement is modeled by sending the message distance with a parameter (meter) that changes randomly. The driver can also press the accelerate_pedal to override the CACC. If the driver presses the brake_pedal or explicitly presses the deactivate button, it is expected that the CACC enters the normal driving mode. In this mode, we assign to the transition that sends a new distance value to the SUT a probability of 30 %. For all other transitions, we assign a probability of 10 %.

- Like the distance, also the own coordinates (latitude and longitude), the coordinates of the car ahead, the speed of the car ahead, and the position of the accelerate pedal of the driver may change. Here again, we use automatic transitions that are triggered according to the assigned probabilities.

This state machine can be processed by the TEAGER tool. As an example of an adaptor, we present the motor adaptor, which transforms concrete observations into abstract ones (Fig. 11.4 on Page 160). It specifies how speed commands correspond to wheel pulses that could be visible at the external interface of the SUT. The Adaptor receives the signal current_speed and stores the parameter value into the attribute speed. If the attribute speed is not 0, a wheel_pulse is generated every 200/speed milliseconds. This results in 1000 pulses per second for a speed of 200 kilometers per hour, 100 pulses per second for a speed of 20 kilometers per hour, 10 pulses per second for a speed of 2 kilometers per hour, and no pulses in standstill.

For modeling requirements, we have developed several patterns:

11.2.1. Single Event – In State

The pattern represented in the state machine shown in Fig. 11.5 is usable when the requirement has the form “When [*eventR*$_i$] happens, [controlled domain] should be in [*desiredStateR*$_i$]”. When the event of interest happens, then the precondition of the requirement is fulfilled, and the event *checkR*$_i$ is generated. The state machine representing the postcondition contains the desired state and may also contain other states. Only if it is in the desired state, the test passes; otherwise, a violation is determined, or the test is inconclusive. The latter happens, for example, if the actual state of the system is not known. Then, the result of checking a requirement should neither be pass nor fail. In many cases, we do not initially know the (physical) state of the SUT. Hence, we introduce an ”unknown state” (denoted by ANY) expressing this situation. Checking requirement R_1 in this state yields an inconclusive result.

11.2.2. Multiple States, Single Event – Output Event

The pattern represented in the state machine shown in Fig. 11.6 is usable when the requirement has the form “When [domains] are in state [*Check_{i.1}*] and ... and in state [*Check_{i.n-1}*] and [*inEventR_i*] happens, [*outEventR_i*] should happen after a certain time”. The state machines in the regions with the prefix Pre monitor the relevant states by observing the corresponding events. The the state machine in the region Pre-R{1.n} additionally generates the event checkRi when the [*inEventR_i*] happens. The state machine checking the postcondition of the requirement

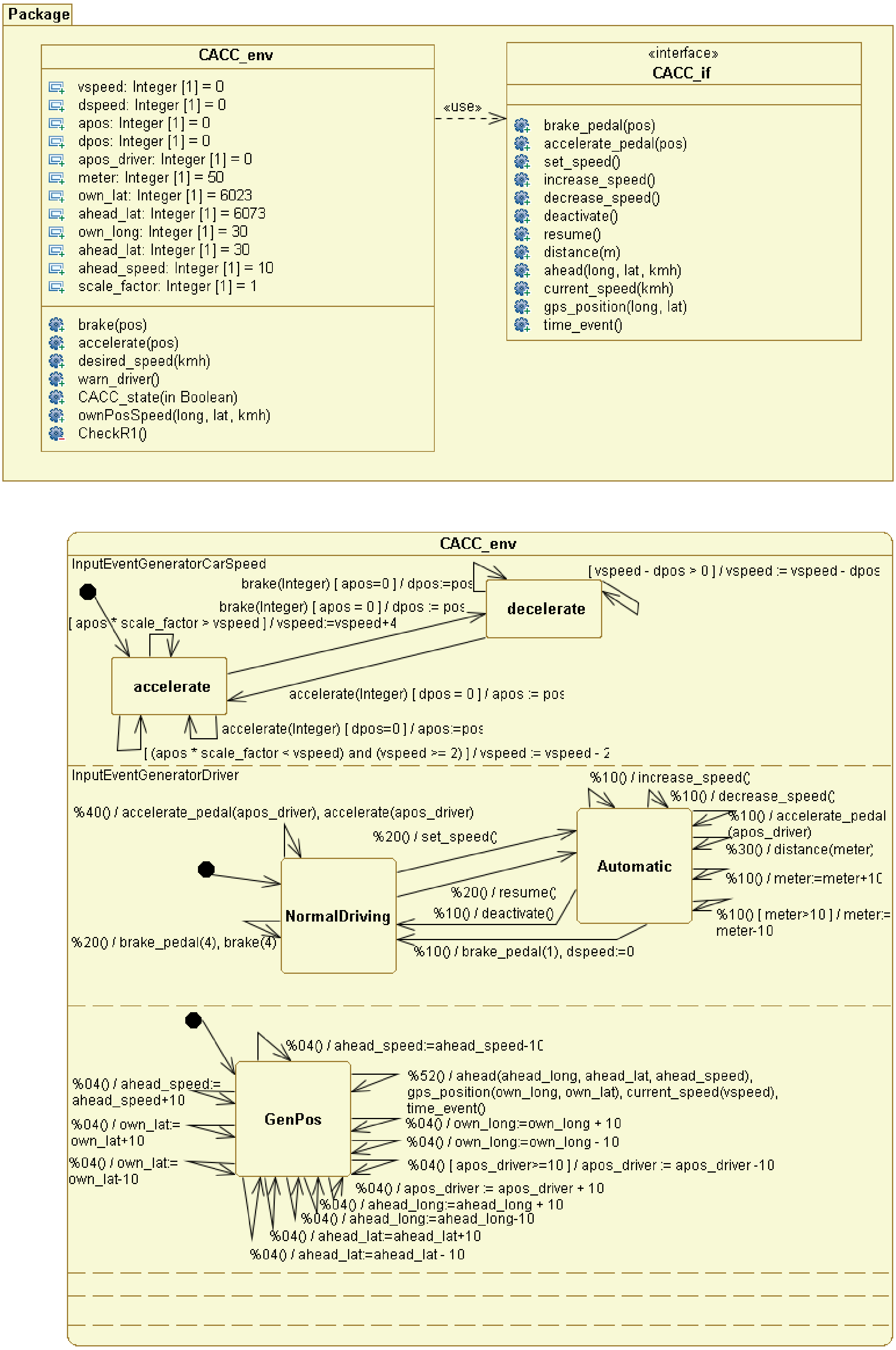

Package
CACC_env
vspeed: Integer [1] = 0
dspeed: Integer [1] = 0
apos: Integer [1] = 0
dpos: Integer [1] = 0
apos_driver: Integer [1] = 0
meter: Integer [1] = 50
own_lat: Integer [1] = 6023
ahead_lat: Integer [1] = 6073
own_long: Integer [1] = 30
ahead_lat: Integer [1] = 30
ahead_speed: Integer [1] = 10
scale_factor: Integer [1] = 1
brake(pos)
accelerate(pos)
desired_speed(kmh)
warn_driver()
CACC_state(in Boolean)
ownPosSpeed(long, lat, kmh)
CheckR1()
«use»
«interface»
CACC_if
brake_pedal(pos)
accelerate_pedal(pos)
set_speed()
increase_speed()
decrease_speed()
deactivate()
resume()
distance(m)
ahead(long, lat, kmh)
current_speed(kmh)
gps_position(long, lat)
time_event()
CACC_env
InputEventGeneratorCarSpeed
brake(Integer) [apos=0] / dpos:=pos
[vspeed - dpos > 0] / vspeed := vspeed - dpos
brake(Integer) [apos = 0] / dpos := pos
decelerate
[apos * scale_factor > vspeed] / vspeed:=vspeed+4
accelerate
accelerate(Integer) [dpos = 0] / apos := pos
accelerate(Integer) [dpos=0] / apos:=pos
[(apos * scale_factor < vspeed) and (vspeed >= 2)] / vspeed := vspeed - 2
InputEventGeneratorDriver
%10() / increase_speed()
%10() / decrease_speed()
%40() / accelerate_pedal(apos_driver), accelerate(apos_driver)
%10() / accelerate_pedal (apos_driver)
%20() / set_speed()
%30() / distance(meter)
Automatic
%10() / meter:=meter+10
%20() / resume()
%10() / deactivate()
%10() [meter>10] / meter:= meter-10
NormalDriving
%20() / brake_pedal(4), brake(4)
%10() / brake_pedal(1), dspeed:=0
%04() / ahead_speed:=ahead_speed-10
%52() / ahead(ahead_long, ahead_lat, ahead_speed), gps_position(own_long, own_lat), current_speed(vspeed), time_event()
%04() / ahead_speed:= ahead_speed+10
GenPos
%04() / own_long:=own_long + 10
%04() / own_lat:= own_lat+10
%04() / own_long:=own_long - 10
%04() / own_lat:= own_lat-10
%04() [apos_driver>=10] / apos_driver := apos_driver -10
%04() / apos_driver := apos_driver + 10
%04() / ahead_long:=ahead_long + 10
%04() / ahead_long:=ahead_long-10
%04() / ahead_lat:=ahead_lat+10
%04() / ahead_lat:=ahead_lat - 10

Figure 11.3.: *Input Event Generator for CACC*

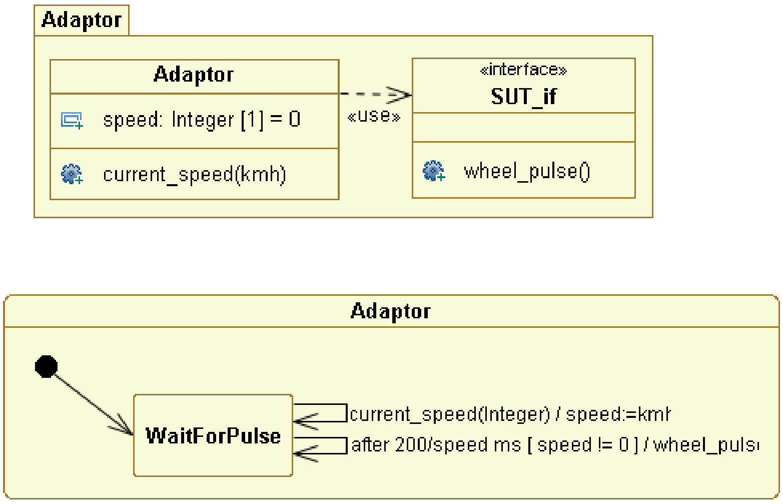


Figure 11.4.: *Adaptor for CACC*

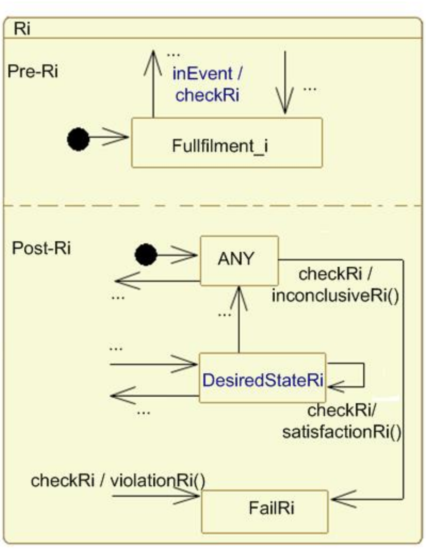


Figure 11.5.: *Pattern for* SingleEvent-InState *(cf. (Naveed, 2010))*

in region Post-Ri starts in the state ANY. If the generated event checkRi is observed and the defined states are active, the state machine enters the state CheckingRi. Then either the desired outEvent can be observed and the requirement is rated as satisfied, or after the specified time interval the state FailR1 is entered. When the precondition is established again after a received outEvent, the state CheckingRi is entered again. This pattern is applied for R01 of the CACC case study in Section 11.4.

11.2.3. Multiple States – Output Event

The pattern represented in the state machine shown in Fig. 11.7 is usable when the requirement has the form "When [domains] are in state [*Check_{i.1}*] and ... and in state [*Check_{i.n-1}*], [*outEventR_i*] should happen after a certain time".

The state machines is similar to the state machine in Fig. 11.6 on the facing page, but no input event is observed by the state machines for the precondition.

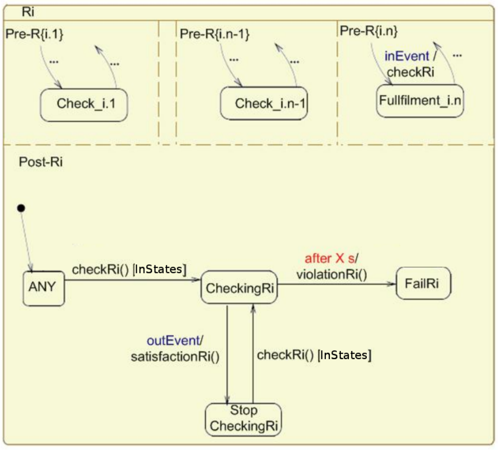


Figure 11.6.: *Pattern of* MultiStatesSingleEvent-OutEvent *(cf. (Naveed, 2010))*

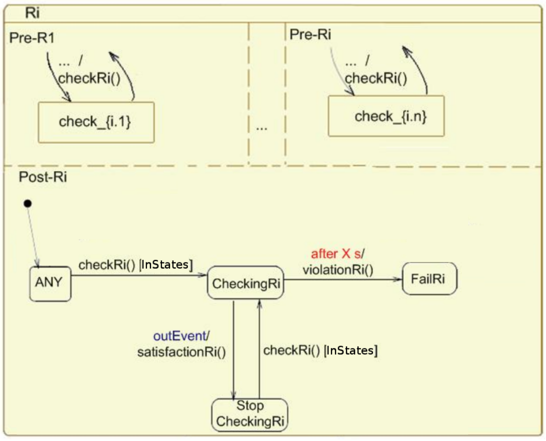


Figure 11.7.: *Pattern of* MultiStates-OutEvent *(cf. (Naveed, 2010))*

11.2.4. Multiple States – No Event

The pattern represented in the state machine shown in Fig. 11.8 is usable when the requirement has the form "When [domains] are in state [*Check_{i.1}*] and ... and in state [*Check_{i.n-1}*], [*outEventR_i*] should not happen within a certain time". This state machine again has the initial state ANY. In this state, the state of the environment model is not yet determined. If the *outEventR_i* happens at this point of time, the state machine results an InconclusiveRi. When the preconditions are fulfilled, i.e., the environment is in the given state, the state chechkingRi is entered. This state is either left with the output event satisfactionRi if the precondition is no longer fulfilled or with output event failR1 if the event that should not happen is observed.

11.2.5. Other Combinations, e.g., Single Event – Not In State

The list of patterns given here is not complete: pattern for other combinations of states and events can be created by using the basic elements used by the given patterns, see Sun (2008).

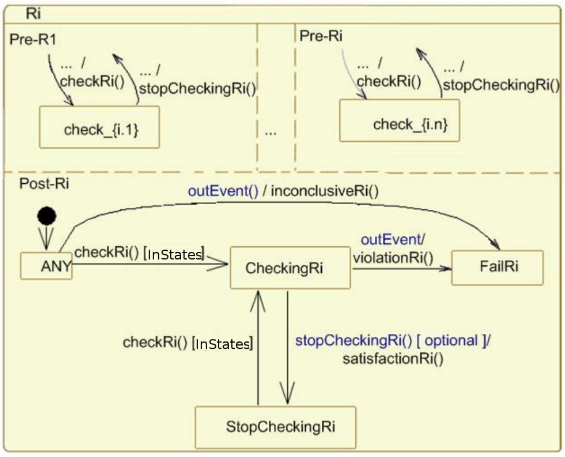


Figure 11.8.: *Pattern for* MultiStates-NoEvent *(cf. (Naveed, 2010))*

11.3. Tool Support for Automatic Test Case Generation using UML-Models

The approach can be realized by using the tool Teager (Seifert, 2009). Teager is a tool automating tests based on UML state machines, with regard to the UML semantics definition (UML Revision Task Force, 2010c).

It provides features for non-determinism in models to be executed. The generation of test cases is in a batch fashion using several probabilistic strategies. It has a simulator for state machines, which enables validation of state machine models. This section discusses the realization in Teager.

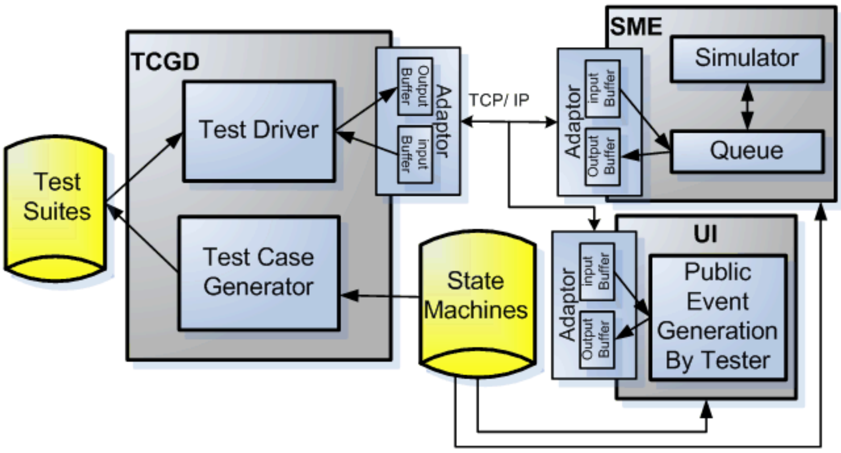


Figure 11.9.: *The architecture of* Teager

We can use Teager on the one hand as a test environment for test case generation and execution, and on the other hand, as an environment for the simulation of state machines and testing the SUT against state machines.

As shown in Fig. 11.9, Teager consists of three main components TCGD (Test Case Generator and test Driver), SME (State Machine Executor) and UI (User Interface), which communicate via the TCP/IP protocol.

The TCGD enables test case generation and test case execution. The test cases are generated by stimulating a UML state machine with random stimuli. Stimuli and the responses according to the UML model (including alternatives in case of nondeterministic state machines) are stored as a test suite. The TCGD can execute the test cases in the suite in order to check the SUT by triggering an SUT and matching the responses of the SUT with the results the test case description. The SUT could be the implementation or also another model.

The SME can execute UML state machines. It calculates the responses to given stimuli. In case of nondeterministic state machines, the responses are chosen randomly. Additionally, it is possible to annotate probabilities at transitions without a trigger. In this case, these probabilities are used for calculating the responses. We extended TEAGER with the capability to handle transitions that are triggered automatically after a given time interval.

The UI also has the state machine as an input. The state machine has to be integrated into a class with a used interface. For all operations in the class, the UI, generates buttons and for the given parameters, input fields are generated. The UI can be used for a manual validation of the state machine.

The communication between the test driver (in the TCGD), the SUT and UI takes place asynchronously over a *TCP/IP* connection which uses two buffers for incoming and outgoing events (messages).

TEAGER is limited to a subset of the UML state machine specification (UML Revision Task Force, 2010c, Chapter 15). It cannot handle the UML state machine elements shallow history, deep history, entry point, exit point, junction, fork, joint, choice pseudo state, and terminate note. But the most important elements can be simulated by other elements, e.g. a choice pseudo state can be also realized by guarded transitions and a fork can be also realized by a transition to a state with several regions in a hierarchical state machine. It is only only allowed to use the data types boolean and integer for state machines used by TEAGER. TEAGER can handle state machines in different representations:

- Files with the extensions *.spec* and *.impl* contain textual representation of the state machine. These files can be directly loaded into editor panel of the TEAGER components (TCGD, SME or UI). The format of the textual representation is defined in Seifert (2007).

- Files with the extension *.uml* can be created with an EMF-based UML tool, e.g. *Papyrus UML*. These files can be imported and are transformed into the textual representation of these UML state machine. To import the state machine, TEAGER makes use of of EMF framework.

11.4. CACC Case Study

To check behavior of the machine CACC in the CACC case study, the state machine of the CACC (here SUT), the state machine for the input event generator, and the state machine for the requirements have to be defined.

The state machines for the CACC have been defined in the design step D4 (see Appendix A.12). The state machines can be processed by TEAGER if the following conditions are true:

- A class diagram exists where the SUT is represented as a class with operations that correspond to the observed phenomena of the machine (or a submachine according to a mapping diagram). Parameters have been added according to the analysis steps and the provided interfaces in the design.

- An interface used by the class representing the SUT exists. The operations in this interface correspond to the controlled phenomena of the machine. Parameters have been added according to the analysis steps and the used interfaces in the design.

- Since Teager currently cannot handle the data type “String” and complex data types, parameters with these types have to be replaced by parameters of type “Integer”.
- The class representing the SUT contains the CACC state machine. All operations of CACC are trigger events for the transitions (e.g., time_event() in Figures 11.10 and 11.11)
- In the effect specification of the triggers (e.g., apos:=apos+10, accelerate(apos) in Fig. 11.11), the operations of the used interface are used or/and the variables of the class are changed.
- The elements in optional transition guards refer to parameters or attributes of the class (e.g., [(desired_speed>own_speed) and (dist>own_speed/2)] in Fig. 11.11).

The class diagram for the CACC as system under test is shown in Fig. 11.10. The state machine of the CACC is depicted in Fig. 11.11. The diagram fulfills all rules given above. It contains the same attributes desired_speed and last_desired_speed of the lexical domain ACCSpeed in Fig. 4.22 on Page 47. The parameters given in the operation gps_position are stored in the attributes own_latitude and own_longitude. The parameter given in the operation current_speed is stored in own_speed. The position to be given to the engine actuator is stored in apos and the position to be given to the brake is is stored in bpos. The distance is either given by the operation distance or calculated from own_latitude, own_longitude, own_speed and the parameters of the operation ahead.

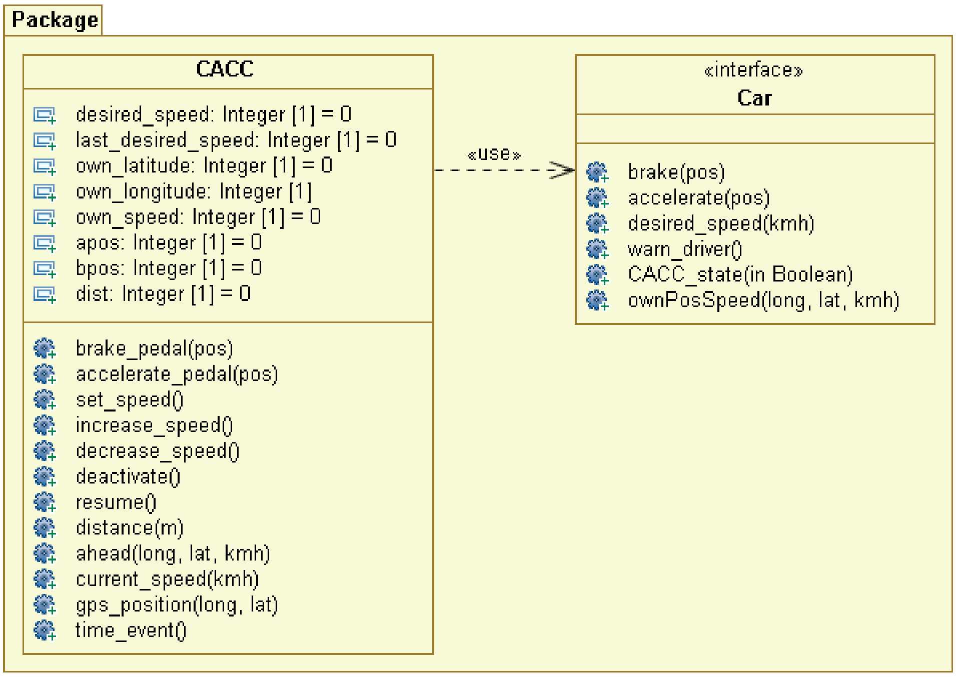


Figure 11.10.: *Class Diagram for CACC as SUT*

The state machine depicted in Fig. 11.11 models the attribute activated of the lexical domain ACCSpeed in Fig. 4.22 on Page 47 by the states CACC_deactivated and CACC_activated since the behavior of the CACC is different. In the state CACC_deactivated, just the current pedal positions are stored. The attribute apos represents the acceleration pedal position with a range from 0-200. The attribute dpos represents the break pedal position with a range from 0-200. With resume or set_speed the CACC_activated state is entered. In this state, the driver’s buttons increase_speed and decrease_speed are used to adjust the desired speed, the current speed and

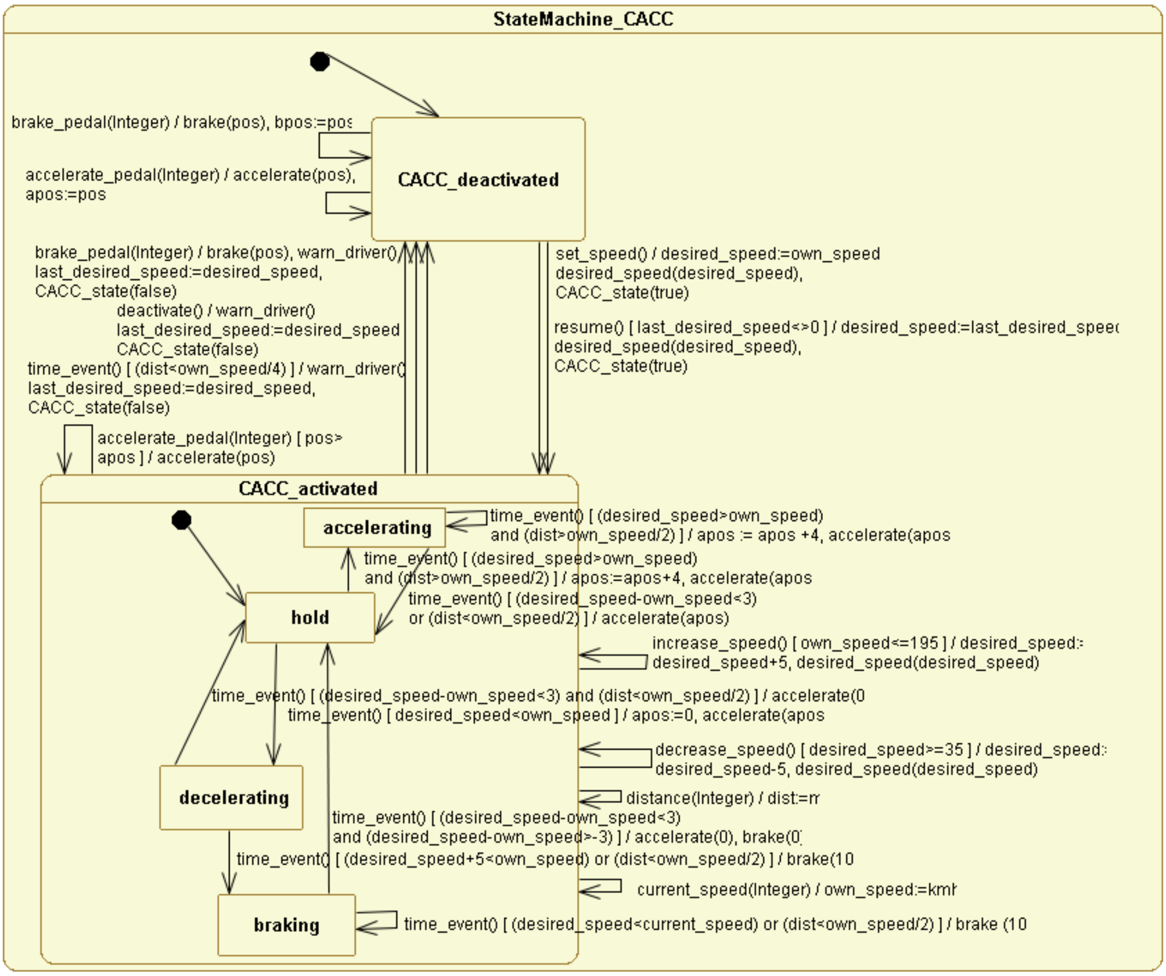

StateMachine_CACC
brake_pedal(Integer) / brake(pos), bpos:=pos
accelerate_pedal(Integer) / accelerate(pos),
apos:=pos
CACC_deactivated
brake_pedal(Integer) / brake(pos), warn_driver()
last_desired_speed:=desired_speed,
CACC_state(false)
deactivate() / warn_driver()
last_desired_speed:=desired_speed
CACC_state(false)
time_event() [(dist<own_speed/4)] / warn_driver()
last_desired_speed:=desired_speed,
CACC_state(false)
set_speed() / desired_speed:=own_speed
desired_speed(desired_speed),
CACC_state(true)
resume() [last_desired_speed<>0] / desired_speed:=last_desired_speed
desired_speed(desired_speed),
CACC_state(true)
accelerate_pedal(Integer) [pos>
apos] / accelerate(pos)
CACC_activated
accelerating
time_event() [(desired_speed>own_speed)
and (dist>own_speed/2)] / apos := apos +4, accelerate(apos
time_event() [(desired_speed>own_speed)
and (dist>own_speed/2)] / apos:=apos+4, accelerate(apos
time_event() [(desired_speed-own_speed<3)
or (dist<own_speed/2)] / accelerate(apos)
hold
increase_speed() [own_speed<=195] / desired_speed:=
desired_speed+5, desired_speed(desired_speed)
time_event() [(desired_speed-own_speed<3) and (dist<own_speed/2)] / accelerate(0
time_event() [desired_speed<own_speed] / apos:=0, accelerate(apos
decrease_speed() [desired_speed>=35] / desired_speed:=
desired_speed-5, desired_speed(desired_speed)
decelerating
distance(Integer) / dist:=m
time_event() [(desired_speed-own_speed<3)
and (desired_speed-own_speed>-3)] / accelerate(0), brake(0)
time_event() [(desired_speed+5<own_speed) or (dist<own_speed/2)] / brake(10
current_speed(Integer) / own_speed:=kmh
braking
time_event() [(desired_speed<current_speed) or (dist<own_speed/2)] / brake (10

Figure 11.11.: *State Machine of CACC as SUT*

current distance are stored, and the driver can override the automatic control by pressing the accelerate_pedal. When the brake_pedal is pressed, the deactivate button is pressed or the distance to the car ahead is below a critical distance, the driver is warned and the state CACC_deactivated is entered.

The state CACC_activated has the sub-states accelerating, hold, decelerating, and braking. The initial state is hold. The transitions between the sub-states are triggered by a time_event. This event is sent every 20 ms by the CACC hardware. The state changes to accelerating if desired_speed is higher than own_speed and the distance to the car ahead is safe. In this case, the value apos is increased by 10 and passed on to the engine. The same activity is performed in the state accelerating when the time_event is received. The state accelerating is left and hold is entered again if the distance to the car ahead becomes too small or the desired speed is reached. The state changes to decelerating if desired_speed lower higher than own_speed or the distance to the car ahead is not safe. In this case, the an acceleration value of 0 is passed on to the engine. The state decelerating is left and hold is entered again if the distance to the car ahead becomes safe enough or the desired speed is reached. The state decelerating is left and braking is entered if the desired speed is much lower than own_speed or the distance to the car ahead becomes much too small. In this case the brake is activated. The constant brake parameter value of 10 is used here to simplify the model. This state braking is left and the state hold is entered if the distance to the car ahead is again safe and the desired speed is reached. Note the handling of position operator gps_position is not included in the state machine diagram.

To test the requirements, the input event generators, the adaptors and the requirements have to be modeled as state machines. In this chapter we omit the adaptors and present the state machine for one requirement. The other requirements are also instances of the patterns in Section 11.2. Additionally, a class diagram defining the interfaces have to be specified. In this class diagram (see Fig. 11.12 the used interfaces of the SUT become operations of the environment and operations of the SUT become used interfaces.

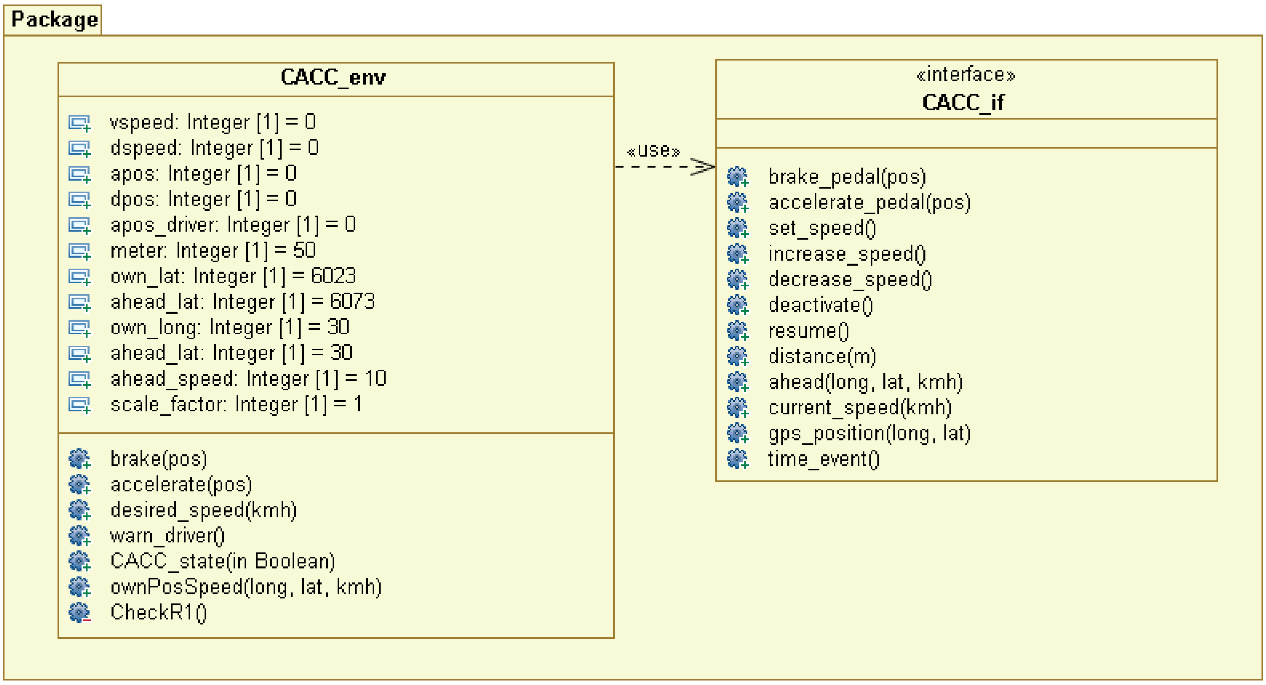


Figure 11.12.: *Class Diagram for Environment Model*

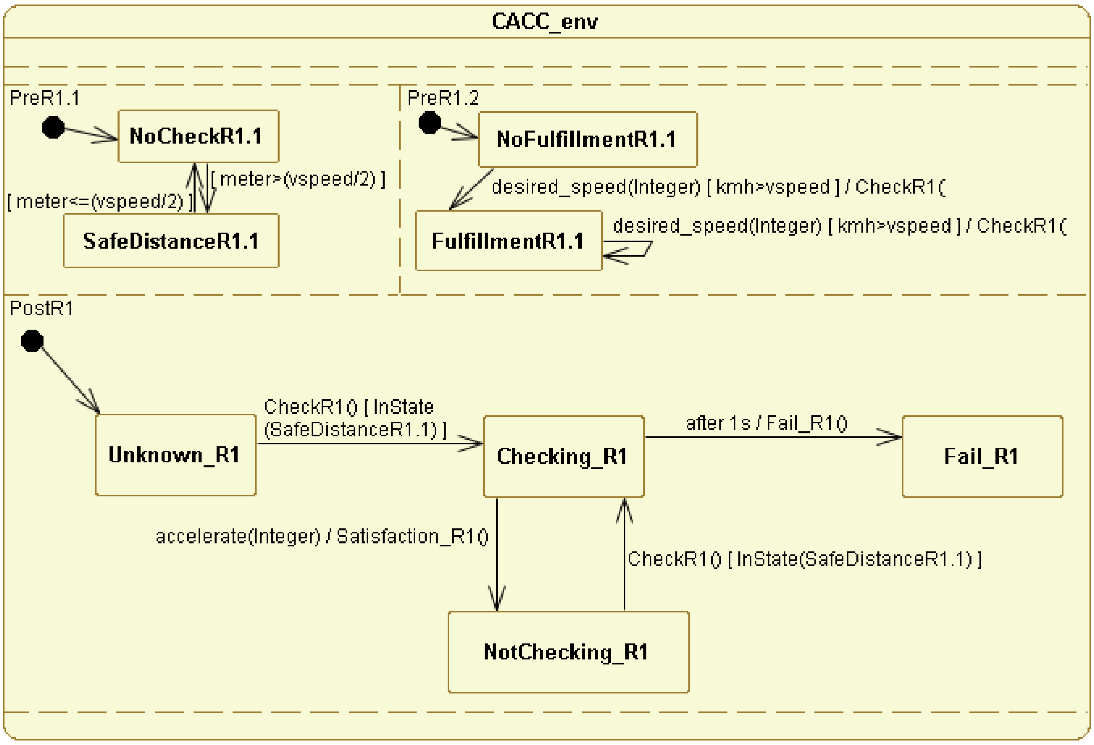
The requirement

R1 The CACC should accelerate the car if the desired_speed is higher than the current_speed, the CACC is activated and the distance to the car(s) ahead is safe.

can be modeled with the state machine in Fig. 11.13. It is an instance of the pattern depicted in Fig. 11.6 on Page 161. The precondition in the instance consists of 2 parts: The first part enters the state SafeDistanceR1.1 if the distance is safe (meter>(speed/2)), and it leaves this state if the distance is no longer safe (meter>(speed/2)). The terms meter and speed refer to attributes of the environment class. The state SafeDistanceR1.1 in the instance corresponds the state Check_i.1. The second part triggers the postcondition state machine with CheckR1 if the signal desired_speed is received and its parameter kmh is higher than the current speed of the car (attribute vspeed). We choose the SUT output signal desired_speed as a trigger because it indicates that CACC is activated and carries the desired speed. In the postcondition of the instance, the state Unknown_R1 corresponds to the state ANY in the pattern, Checking_R1 corresponds to CheckingRi in the pattern, NotChecking_R1 corresponds to StopCheckingRi, and Fail_R1 corresponds to FailRi in the pattern.

The Input-Event-Generator is presented in Section 11.2. Here, we omit the Interface Abstraction Layer in the environment model as well as the adaptors and the Hardware Abstraction Layer with the Drivers in the SUT model in order to focus on the main aspects. Additionally, the handling of the speed and position transmitted by a wireless interfaces are omitted.

Figure 11.13.: *State Machine to check Requirement R1*


CACC_env
PreR1.1
NoCheckR1.1
[meter>(vspeed/2)]
[meter<=(vspeed/2)]
SafeDistanceR1.1
PreR1.2
NoFulfillmentR1.1
desired_speed(Integer) [kmh>vspeed] / CheckR1(
FulfillmentR1.1
desired_speed(Integer) [kmh>vspeed] / CheckR1(
PostR1
Unknown_R1
CheckR1() [InState (SafeDistanceR1.1)]
Checking_R1
after 1 s / Fail_R1()
Fail_R1
accelerate(Integer) / Satisfaction_R1()
CheckR1() [InState(SafeDistanceR1.1)]
NotChecking_R1

The presented state machines for the Input-Event-Generators and Requirement R1 can be imported into Teager . Its textual state machine representation for Teager is shown in Listing 11.1.

```
name = CACC_env;

structure =
    RegionIII(STATEIII!(Region1(State_0!)&Region2(State_0!)&Region3(State_0!)&
    Region4(accelerate!|decelerate)&Region5(NormalDriving!|Automatic)&
    Region6(GenPos!)&Region7(NoCheckR1.1!|SafeDistanceR1.1)&
    Region8(NoFulfillmentR1.1!|FulfillmentR1.1)));

public brake(int pos);
public accelerate(int pos);
public desired_speed(int kmh);
public warn_driver;
public CACC_state(boolean state);
public ownPosSpeed(int long, int lat, int kmh);

private CheckR1();

timer TimeEvent_0(after 1 s);

external brake_pedal(int pos);
external accelerate_pedal(int pos);
external set_speed;
external increase_speed;
external decrease_speed;
external deactivate;
external resume;
external distance(int m);
external ahead(int long, int lat, int kmh);
external current_speed(int kmh);
external gps_position(int long, int lat);
external time_event;

```

```
int vspeed = 0;
int dspeed = 0;
int apos = 0;
int dpos = 0;
int apos_driver = 0;
int meter = 50;
int own_lat = 6023;
int ahead_lat = 6073;
int own_long = 30;
int ahead_long = 30;
int ahead_speed = 10;
int scale_factor = 1;

decelerate -> accelerate[dpos==0] / apos=accelerate.pos; -> accelerate;
accelerate -> accelerate[dpos==0] / apos=accelerate.pos; -> accelerate;
accelerate -> brake[apos==0] / dpos=brake.pos; -> decelerate;
decelerate -> brake[apos==0] / vspeed=vspeed-dpos; -> decelerate;
accelerate -> *[apos*scale_factor>vspeed] / vspeed=vspeed+4; -> accelerate;
accelerate -> *[(apos*scale_factor<vspeed)&(vspeed>=2)] / vspeed=vspeed-2; ->
    accelerate;
decelerate -> *[dpos > vspeed] / vspeed=vspeed-dpos; -> decelerate;

NormalDriving -> %20 / send(resume); -> Automatic;
NormalDriving -> %20 / send(set_speed); -> Automatic;
NormalDriving -> %40 / send(accelerate_pedal(apos_driver));
    send(accelerate(apos_driver)); -> NormalDriving;
NormalDriving -> %20 / send(brake_pedal(4)); send(brake(4));-> NormalDriving;
Automatic -> %10 / send(deactivate); -> NormalDriving;
Automatic -> %10 / send(increase_speed); -> Automatic;
Automatic -> %10 / send(decrease_speed); -> Automatic;
Automatic -> %10 / send(accelerate_pedal(apos_driver)); -> Automatic;
Automatic -> %10 / send(brake_pedal(1)); dspeed=0;-> NormalDriving;
Automatic -> %30 / send(distance(meter)); -> Automatic;
Automatic -> %10[meter>10] / meter=meter-10; -> Automatic;
Automatic -> %10 / meter=meter+10; -> Automatic;

GenPos -> %52 / send(ahead(ahead_long,ahead_lat,ahead_speed));
    send(gps_position(own_long,own_lat)); send(current_speed(vspeed));
    send(time_event); -> GenPos;
GenPos -> %04 / ahead_speed=ahead_speed+10; -> GenPos;
GenPos -> %04 / ahead_speed=ahead_speed-10; -> GenPos;
GenPos -> %04 / own_lat=own_lat+10; -> GenPos;
GenPos -> %04 / own_lat=own_lat-10; -> GenPos;
GenPos -> %04 / own_long=own_long+10; -> GenPos;
GenPos -> %04 / own_long=own_long-10; -> GenPos;
GenPos -> %04 / apos_driver=apos_driver+10; -> GenPos;
GenPos -> %04[apos_driver>=10] / apos_driver=apos_driver-10; -> GenPos;
GenPos -> %04 / ahead_long=ahead_long+10; -> GenPos;
GenPos -> %04 / ahead_long=ahead_long-10; -> GenPos;
GenPos -> %04 / ahead_lat=ahead_lat+10; -> GenPos;
GenPos -> %04 / ahead_lat=ahead_lat-10; -> GenPos;

NoCheckR1.1 -> *[meter>(vspeed/2)] / -> SafeDistanceR1.1;
SafeDistanceR1.1 -> *[meter<=(vspeed/2)] / -> NoCheckR1.1;

NoFulfillmentR1.1 -> desired_speed[desired_speed.kmh>vspeed] / send(CheckR1()); ->
    FulfillmentR1.1;
FulfillmentR1.1 -> desired_speed[desired_speed.kmh>vspeed] / send(CheckR1()); ->
    FulfillmentR1.1;

Unknown_R1 -> CheckR1[InState(SafeDistanceR1.1)] / setTimer(TimeEvent_0); ->
    Checking_R1;
NotChecking_R1; -> CheckR1[InState(SafeDistanceR1.1)] / setTimer(TimeEvent_0); ->
    Checking_R1;
Checking_R1 -> accelerate / send(SatisfactionR1); -> NotChecking_R1;
Checking_R1 -> TimeEvent_0 / send(FailR1); -> Fail_R1;
```

Listing 11.1: *Environment State Machine for Testing with* Teager

The textual state machine describes exactly the same behavior as the state machines in the environment class diagram (see Figures 11.3 on Page 159 and 11.13 on the previous page). In line 1 of Listing 11.1, the name of the class is given. Line 3 describes the structure of the state machine. Regions are described by the name of the region followed by brackets with the states inside. If a state is a composite state, it can contain several regions separated by '&'. The initial state of each region is marked by a following exclamation mark. Lines 5-10 contain the public operation declarations of the class (see Fig. 11.12). In Line 12 the internal operation CheckR1 used in the state machine for R1 is declared. Line 14 declares the time event after 1s used in the state machine for R1. Lines 16-27 represent the operations of the used interface (see Fig. 11.12). In lines 29-40, the attributes of the class in Fig. 11.12 are specified. In lines 42-87, all transitions are described in the form 'source state', followed by '->', followed by the trigger event, followed by an optional guard in square brackets, followed by '/', followed by the effect, followed by '->', and followed by the target state. Within the effect, Java notation is used for calculations and the keyword 'send' is used for sending events. For parameters of an operation, the operation name followed by a dot and the parameter name is used. The state machines are modeled in the following lines:

- Lines 42-28 model the state machine of region InputEventGeneratorCarSpeed of Fig. 11.3.
- Lines 51-62 model the state machine of region InputEventGeneratorDriver of Fig. 11.3
- Lines 64-76 model the state machine of region InputEventGeneratorPos of Fig. 11.3
- Lines 78-79 model the state machine of region PreR1.1 of Fig. 11.13
- Lines 81-82 model the state machine of region PreR1.2 of Fig. 11.13
- Lines 84-87 model the state machine of region PostR1 of Fig. 11.13

All in all, to completely model the CACC problem, 3 input event generators, some adaptors (omitted), and one state machines for each requirement have to be set up. All the requirements are instances of patterns:

- R1: Multiple States, Single Event – Output Event (see above)
- R2: Multiple States, Single Event – Output Event (similar to R1)
- R3: Single Event – In State (activated)
  (Note that the changes in the internal lexical domain cannot be checked directly.)
- R4: Single Event – In State (activated)
  (Note that the changes in the internal lexical domain cannot be checked directly.)
- R3+R4: Multiple States, Single Event – Output Event (desired_speed)
  (Note: A second instance of the pattern is necessary for one requirement because of second output events.)
- R5: Multiple States, Single Event – Output Event (desired_speed)
- R6: Multiple States, Single Event – Output Event (desired_speed)
- R7: Multiple States, Single Event – Output Event (desired_speed)
  (with an empty set of states)
- R8: Multiple States, Single Event – Output Event (warn_driver)
- R8: Multiple States, Single Event – Output Event (CACC_state)
  (Note: A second instance of the pattern is necessary for one requirement because of second output events.)

- R9: Multiple States, Single Event – Output Event (ownPosSpeed)
(with an empty set of states)

11.5. Related Work

To our knowledge, no method for testing requirements expressed as UML state machines has been published.

A detailed overview of the fundamental literature for classical formal testing can be found in Brinksma's and Tretmans' annotated bibliography (Brinksma & Tretmans, 2001). De Nicola and Hennessy (Rocco De Nicola and M. C. B. Hennessy, 1984) introduce a formal theory of testing on which Brinksma (Brinksma, 1988) and Tretmans (Tretmans, 1996) build approaches to derive test cases from a formal specification. In contrast to our work, these approaches assume that a testing process communicates synchronously with the system under test. The developed tool TorX (`fmt.cs.utwente.nl/tools/torx`) also allows conformance testing of reactive systems.

Belli at al. (see (Belli & Hollmann, 2007) and the work cited there) base their testing methodology on a variant of state machines. In contrast to our approach, they do not test against requirements, but against a fault model that has to be set up explicitly. Moreover, they do not execute the state machines directly, but represent them as event sequence graphs.

Auguston et al. (Auguston, Michael, & Shing, 2005) use environment models for test case generation. In contrast to our approach, they do not use state machines, but attributed event grammars.

While these works have their merits, we think that the combination of environment models and UML state machines for testing is a particularly attractive one.

Besides the work from academia, an increasing number of CASE Tool manufacturers offer components for model based testing. I-Logix Rhapsody, for example, offers two add-on products for testing state machines, *Test Conduct* and *Testing and Validation* (`www.telelogic.com`). Simulink *Verification and Validation* generates test cases in Simulink and Stateflow, and measures test coverage for Statecharts (`www.mathworks.com`). Conformiq Software Ltd. offers a *Test Generator* which accepts "extended UML state charts" as the model of the system under test for dynamic testing (`www.conformiq.com`). *AsmL 2* by Microsoft provides an executable specification language based on the theory of Abstract State Machines (`research.microsoft.com/fse/asml`). The *AsmL 2* test tool supports parameter generation and test sequence generation based on interface automata.

11.6. Conclusions and Future Work

We have developed a novel approach to testing reactive and embedded systems, based on environment models and using UML state machines. To evaluate our approach, we used the Teager tool suite (Santen & Seifert, 2006; Seifert, 2009). It allows its users to generate and execute test cases or to directly stimulate the SUT. Teager logs the stimuli it sends to the SUT and the reactions of the SUT. During execution, these reactions are compared to the precalculated possible correct reactions to evaluate the test execution process (Seifert, 2007). Using Jackson's terminology, we have defined uniform architectures and procedures for on-the-fly as well as batch testing that have the following characteristics:

- Requirements, facts, and assumptions are modeled explicitly.
- We have defined patterns for the different state machines: For requirements, a parallel state machine is set up for each precondition. When all preconditions are fulfilled, the

postcondition is checked. Input generators and adaptors also consist of parallel state machines, one for each item of the environment that generates stimuli or receives observations, respectively.

- Once these models have been set up manually (but systematically), the tests are performed automatically, using the tool TEAGER .

Currently our approach is limited to reactive systems that can be modeled with state machines. Systems that are based on complex data types cannot be tested with this approach. It is an interesting task to extend TEAGER to be able to handle complex data types.

Our approach has the following advantages:

- When requirements change, in the test case generator only the state machine describing those requirements must be changed.
- Modeling the facts and assumptions about the environment supports the validation of requirements.
- States can be left out in the environment model if the machine implements features that are not part of the requirements. The same environment model can be (re-)used for a CACC that uses GPS position and a CACC that uses only the radar sensor.
- Modeling the environment adds diversity to the development process and thus helps to avoid that the same mistake occurs for test development and SUT development. This is because the test developers, who model the environment, must think in terms of the environment rather than the SUT behavior.
- In the environment model, a reasonable test case selection strategy can be defined, so that no inadequate test cases are generated. Atypical behavior can be identified and tested using a dedicated environment model.

Chapter 12

Relation to Standards

The development of safe and secure systems is performed differently and certification is done according to different standards with different approaches. The standard ISO/IEC 61508 (International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC), 2000) is applied for many safety-critical systems. It focuses on processes. In contrast, the Common Criteria (ISO/IEC 15408, CC, International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC) (2009a)), as applied for many security-critical systems, focuses on documents. The ISO/IEC 61508 classifies systems into different Safety Integrity Levels (SIL 1 to SIL 4). The SIL describes the necessary risk reduction by reducing the failure rate of the functionality. To achieve a higher SIL additional safety mechanisms must be added or the design must be changed. The Common Criteria (CC) define seven Evaluation Assurance Levels (EAL). A higher EAL provides a higher level of trust. Since the CC is document-oriented, the same system can, e.g. be certified EAL 3 and later EAL4 without any change of the system. The terminology used in the two standards is completely different. The differences will be discussed in the sections below. This chapter is based on joint work with Maritta Heisel and Holger Schmidt. Results have been presented on the SQS Conference 2008 in Düsseldorf in our talk "Combining safety and security in a model-based development process".

With the methods described in Chapters 4 - 11 the development of safe and secure systems can be integrated seamlessly. This allows certification according to Common Criteria and ISO/IEC 61508 for the same product without doubling the effort. This chapter provides information about the relation between our approach (see Chapter 3) and the Common Criteria in Section 12.1 and ISO/IEC 61508 in Section 12.2. No dedicated section discussing related work is given since the whole chapter only describes related work. In Section 12.3, we give a summary of this chapter and point out directions for future work.

12.1. Common Criteria

Within the Common Criteria (International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC), 2009a), the machine is called *Target Of Evaluation (TOE)*, the requirements correspond to the *Security Objectives* and the specification is stated as *TOE Security Functionality (TSF)*. The TSF describe the combined functionality of all hardware, software, and firmware of a TOE that must be relied upon for the correct enforcement of the *Security Functional Requirements (SFRs)*. For the SFRs, the Common Criteria provides textual patterns.

The Common Criteria defines a set of *Security Assurance Components* that have to be considered for a chosen *Evaluation Assurance Level (EAL)*. For these components, developer activities, content of corresponding components, and actions for an evaluator are defined. The Common Criteria defines security assurance components for the following *Assurance classes*:

- Protection Profile Evaluation (APE)
- Security Target Evaluation (ASE)
- Development (ADV)
- Life-Cycle support (ALC)
- Tests (ATE)
- Vulnerability Assessment (AVA)

The security assurance components for *Protection Profile* evaluation and *Security Target* evaluation are similar. They cover the requirements engineering phases - the Protection Profile from customer side, and the Security Target from supplier side. Creating these documents can be supported by the methods described in Chapters 4, 5, and 6. The Common Criteria requires a description of the threats. Not only the functionality of the TOE, but also the functionality provided by the environment is described using SFRs. Threats and SFRs for the environment constitute the domain knowledge. Examples for domain knowledge of the CACC are physical properties about acceleration, breaking, and measurement of the distance (relevant for safety). Other examples are the assumed intention, knowledge and equipment of an attacker. The intention of the attacker could be to change the speed of the car, in order to produce a rear-end collision. The requirements describe how the environment should behave when the system is in action. The security requirements correspond to the objectives that must be specified in Common Criteria documents. An example for a security objective is to only accept position, acceleration and speed data from another trusted cooperative adaptive cruise control. An overall safety requirement may be to keep a certain distance to the car ahead while being activated. Using Table 5.1 on Page 79, Security Functional Requirements for the TOE and the environment can be easily derived from the security requirements (that correspond to the Security Objectives). For example, the CACC should check if the car ahead has a trusted CACC using an authentication mechanism. It should prevent replay attacks by using random numbers for authentication and a session key for checking the signature of each message. This functionality fulfills the security functional requirements FPD_UIT.1, FTP_ITC.1 and FDP_IFC.1 of the CC (integrity preserving data transmission, see Section 5.5).

The activities within the development process defined by the Common Criteria is structured by a set of *assurance classes* and each assurance classed is structured by a set of *assurance components*. For the development assurance class, the following assurance components are defined:

- Functional specification (FSC)
- Security architecture description (ARC)
- TOE design (TDS, Subsystems and Modules)
- Supplementary material on TSF internals (INT)
- Security policy model (SPM)
- Mapping of the implementation representation of the TSF (IMP)

The functional specification can be supported by the method described in Chapter 7. The security architecture description and the TOE design can be supported by the method described in Chapters 8 and 9. It can be applied for hardware and software architectures. The Common Criteria requires to describe the purpose, the associated security requirements, the method of use, as well as all parameters and error messages of the interfaces to security functions. For each component, the purpose, the behavior, and the interaction with other modules must be described (explicitly required in the assurance component ADV_TDS). The description effort can be substantially decreased if described and verified modules are reused. Such a reuse is supported by the structured requirements analysis presented in Chapters 4, 5, and 6. Supplementary material on TSF internals and a security policy model are very specific for the Common Criteria and only required for high EALs (≥ 5). The mapping of the implementation representation of the TSF can be created with low effort if the method described in Section 10.1 is used.

The life-cycle support in the Common Criteria includes configuration management, product delivery, development security, flaw remediation, a life-cycle definition and the documentation of tools and techniques. These aspects are not covered by this thesis.

For testing, the Common Criteria requires state-of-the-art testing with a documentation that shows that the security aspects are covered and additionally penetration testing based on the vulnerability assessment. These aspects are not covered by this thesis. Nevertheless, for component tests (required in assurance component ATE_DPT of the Common Criteria and 61508 Part 3, Chapter 7.4.7) the specified behavior must be tested. If the behavior is described using semi-formal methods such as UML, the specification can be used for test case generation. For the acceptance test, the requirements and the environment description are necessary. This is particularly important since the requirements can only be fulfilled in the assumed environment. For example, if we can assume a physically protected server, penetration tests that require physical access to the machine are not appropriate.

12.2. IEC 61508

Within the ISO/IEC 61508 (International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC), 2000), the machine is called *Electrical/Electronic/Programmable Electric System (E/E/PES)*, the requirements correspond to *overall safety requirements* in the ISO/IEC 61508 and the specifications correspond to *E/E/PES safety requirements* in the ISO/IEC 61508.

The activities required by ISO/IEC 61508, Part 1, are covered by this thesis as described in the following list:

- The ISO/IEC 61508 requires a description of the *Concepts (Section 7.2)* and a description of the *Overall scope (Section 7.3)*. Both descriptions are part of the domain knowledge and machine definition as shown in Chapter 4.

- The *hazard and risk analysis (Section 7.4)* acts as an input of the process described in Chapter 3.

- Creating the *overall safety requirements (Section 7.5)* and *overall safety requirements allocation (Section 7.6)* can be supported by the methods described in Chapters 4, 5, and 6. The requirements can be classified according to requirements patterns. Patterns for safety requirements consider the reliability of the safety function. For example, the reliability of the safety function for the CACC allows at most 10^{-7} failures per hour (determined by the risk analysis corresponding to a SIL).

- *Overall operation and maintenance planning (Section 7.7)* is not covered by this thesis.

- Parts of the *overall safety validation planning (Section 7.8)* and parts of the *verification* are covered by Phase T3 of ADIT (see Chapter 3/ Appendix A.16).

- *Overall installation and commissioning planning (Section 7.9)* is not covered by this thesis.

- *E/E/PE system safety requirements specification (Section 7.10)* and *E/E/PE safety-related systems: realization (Section 7.11)* are covered by the methods presented in Chapter 7.

- Input for the documentation of *other risk reduction measures (Section 7.12)* is provided by the methods described in Chapter 6. The specification of the system to be built (called machine) can be derived considering the specified domain knowledge by choosing mechanisms. This specification describes the behavior of the machine at its external interfaces. Moreover, if the plausibility check of the distance sensor detects a failure or the self-test of the CACC hardware fails, the CACC should be deactivated.

- *Overall installation and commissioning (Section 7.13)* is not covered by this thesis.

- For the *Overall safety validation (Section 7.14)* we only provide support for reactive systems that can be described by a state machine in Chapter 11.

- *Overall operation, maintenance and repair (Section 7.15)*, *Overall modification and retrofit (Section 7.16)*, and *Decommissioning or disposal (Section 7.17)* are not covered by this thesis.

- For *Verification (Section 7.18)* we only provide support for reactive systems that can be described by state machine in Chapter 11.

In Part 2 of the ISO/IEC 61508, activities with a set of requirements are defined for electrical/electronic/programmable electronic (E/E/PE)) safety-related systems. In Part 3 of the ISO/IEC 61508, activities with a set of requirements are defined for software. The activities are covered by this thesis as follows:

- The *Safety lifecycle requirements (Part 2, Section 7.1 and Part 3, Section 7.1)* are not covered by this thesis.

- *E/E/PE system design requirements (Part 2, Section 7.2)*, *Software Safety Requirements Specification (Part 3, Section 7.2)*, *E/E/PE system design and development (Part 2, Section 7.4)*, *Software design and development (architecture, tools, programming languages, detailed design, implementation, testing, Part 3, Section 7.4)*, *E/E/PE system integration (Part 2, Section 7.5)*, and *Programmable electronics integration (hardware and software, Part 3, Section 7.5)* are addressed in Chapters 8 and 9. They are not separated between hardware and software to allow late hardware and software partitioning. Nevertheless, the presented methods allow the developer to use stereotypes to show which parts are hardware and which parts are software. Hardware and software integration is supported by explicit definition and documentation of interfaces as shown in Chapter 8. The description effort can be substantially decreased if described and verified modules are reused. Such a reuse is supported by the structured requirements analysis presented in Chapters 4, 5, and 6. For the CACC, components performing plausibility checks on sensor values and checks of the standard electronic control unit can be instantiated to achieve the requested detection coverage. Chapter 9 shows how to split requirements to systematically derive software requirements from system requirements. The ISO/IEC 61508 recommends a module-oriented design with, e.g., a software module size limit, information hiding/encapsulation, a parameter number limit, fully defined interfaces, and a description of all interfaces. These descriptions are a result of the methods described in Chapter 8.

  The methods presented in Chapter 10 shows how to implement the specified software components according to their specification in the module description. The presented implementation methods allows static verification and reduces unpredictable behavior as required for safety systems. Implementation requirement of the standard are not repeated in Chapter 10, but additional requirements on object-oriented software are presented.

- *E/E/PE system safety validation planning (Part 2, Section 7.3)*, *Validation Plan for Software Aspects of System Safety (Part 3, Section 7.3)*, *E/E/PE system operation and maintenance procedures (Part 2, Section 7.6)*, *Software operation and modification procedures (Part 3, Section 7.6)*, *E/E/PE system modification (Part 2, Section 7.8)*, and *Software modification (Part 3, Section 7.8)* are not covered by this thesis.
- For testing (see *E/E/PE system safety validation (Part 2, Section 7.7)*, *Software aspects of system safety validation (Part 3, Section 7.7)*, *E/E/PE system verification (Part 2, Section 7.9)*, and *Software verification (Part 3, Section 7.9)*), the 61508 requires state-of-the-art testing with a documentation that shows that the safety requirements are fulfilled. The approach for automatic test case generation described in Chapter 11 can be applied for systems with state-dependent behavior. For the acceptance tests, the requirements and the environment description are necessary. This is particularly important since the requirements can only be fulfilled in the assumed environment. If for special reasons a reliable power supply can be assumed, we need no fault injection tests for the power supply.

Part 4 of the ISO/IEC 61508 contains definitions and abbreviations. Part 5 of the ISO/IEC 61508 is an informative part that presents different approaches for the risk assessment and hazard analysis that can be used as an input for the requirement engineering methods presented in Chapter 4. Part 6 of the ISO/IEC 61508 is also an informative part with guidelines for the application of parts 2 and 3, techniques for evaluating probabilities of hardware failure, calculation of diagnostic coverage, common cause quantification, and it gives example applications of ISO/IEC 61508 software safety integrity tables.

In Annex A of Part 7 of the ISO/IEC 61508, mechanisms for the control of random hardware failures are listed and explained. These mechanisms extend the list of mechanisms given in Chapter 6. The informative guidance for the development of safety-related object oriented-software in Annex G of Part 7 is considered in Chapter 10. Following informative annexes of ISO/IEC 61508, Part 7 are not in the scope of this thesis:

- *Techniques and measures for E/E/PE safety related systems (Annex B and C)*
- *The probabilistic approach to determining software safety integrity for pre-developed software (Annex D)*
- *Overview of techniques and measures for design of ASICs (Annex E)*
- *Definitions of properties of software lifecycle phases (Annex F)*

12.3. Conclusion

In this chapter we have shown, how the method presented in Chapters 3 with the detailed descriptions in Chapter 4 to 11 help in generating a documentation suitable for safety and security certification. We have presented a mapping between the methods presented in this thesis and the security standard Common Criteria and the Safety standard ISO/IEC 61508. Using these mappings, the following points can be achieved. These advantages are neither provided by the Common Criteria nor by the ISO/IEC 61508 alone

- Synergies between standards can be used. The method described in this thesis supports the generation of documentation necessary for certification according to both standards, the Common Criteria and the ISO/IEC 61508.

- Patterns for requirements support a systematic search for missing requirements and help to find contradictions as early as possible.
- The patterns can be associated to a set of components. When a requirement pattern is instantiated, a component that fulfills the requirements can be chosen.
- The acceptance test can be based on the environment model necessary for both standards.

In the future, we plan to achieve a tighter integration of ADIT and the standards. We plan to create guidelines describing the generation of documentation for certification according standards, and maybe also a tool that exports this documentation from the model. We will consider additional standards, e.g. the ISO 26262 that is a specific adaption of the standard ISO/IEC 61508 for automotive systems.

Chapter 13

Case Study

In addition to the CACC case study that served as a running example in the previous chapters, we present a second case study in this chapter. In this case study, a *patient care system* is developed. It displays the vital signs of patients to physicians and nurses, and controls an infusion flow according to previously configured rules. In this setting, the display data and the configuration rules are transmitted over an insecure wireless network. To this case study, we apply the procedures referenced in Chapter 3. Problem elicitation and description are presented in Section 13.1. The prpartoblem decomposition is presented in Section 13.2. Section 13.3 contains the abstract machine specification including the specification of the dependability requirements. Technical context diagram, operations and data specification, and machine life-cycle are discussed in Sections 13.4, 13.5, and 13.6. The architecture of the machine is presented in Sections 13.7 (initial), 13.8 (implementable), and 13.9 (restructured). Communication behavior within this architecture are discussed in Sections 13.10 (inter-component) and 13.11 (intra-component). Section 13.12 discusses the complete behavior specification. The approaches for implementation, unit tests, component tests and system test are state-of-the-art and therefore not considered in this case study. The behavior of the patient care system is not based on the state of machine and environment, but on the internal data. Therefore, the approach presented in Chapter 11 is not applicable.

13.1. Step A1 – Problem Elicitation and Description

Figure 13.1 shows the context diagram of the PatientCareSystem (PCS) case study in UML notation with stereotypes defined in the UML profile UML4PF defined in Chapter 4.

The machine (stereotyped with ≪machine≫) in Fig. 13.1 is represented by the class PatientCareSystem. The context diagram in Fig. 13.1 shows the biddable domains Patient and PhysiciansAndNurses, and the causal domains O2Sensor, HeartbeatSensor, InfusionPump, and Terminal. The domain knowledge (facts and assumptions) in this case study are the following:

F1 The O2Sensor measures the O_2 (oxygen) saturation in the blood of the patient and the pulse of the patient. This physical information (indicated by the stereotype ≪physical≫) is forwarded as electrical signals (indicated by the stereotype ≪electrical≫) to the machine.

F2 The HeartbeatSensor measures the heart frequency. This physical information (indicated by the stereotype ≪physical≫) is forwarded as electrical signals (indicated by the stereotype ≪electrical≫) to the machine.

F3 The InfusionPump provides the requested amount of medicine to the patient by controlling the flow of the infusion. The flow is requested using an electrical signal (indicated by the stereotype ≪electrical≫).

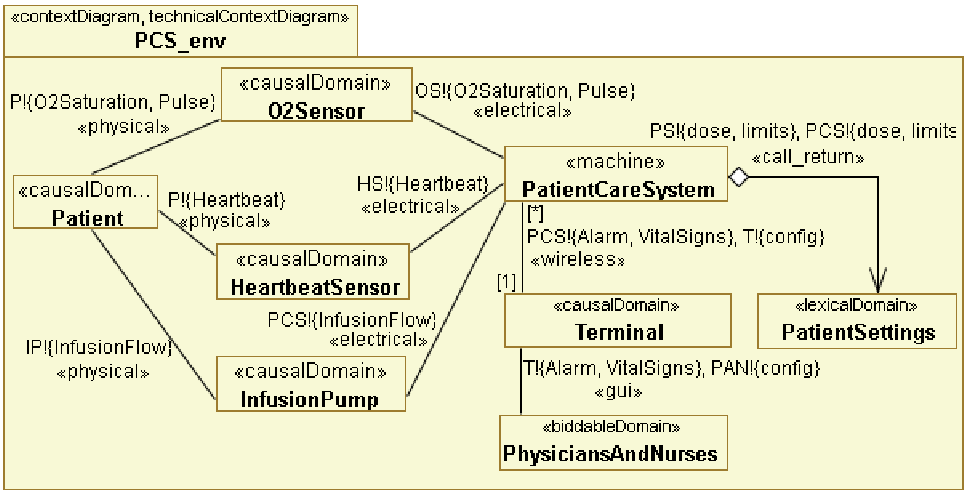


Figure 13.1.: *Context Diagram of Patient Care System*

F4 The Terminal provides a graphical user interface (stereotype ≪gui≫) for nurses and physicians. It displays the alarms and vital signs from the patient care systems and can be used to configure the patient care systems.

F5 Dose and limits for the alarm are stored in the internal PatientSettings of the patient case system.

A1 Nurses and physicians perceive the alarms.

A2 Nurses and physicians configure the patient care systems correctly and appropriately.

The functional requirements are the following:

R1 The vital signs should be displayed, and an alarm should be raised if the vital signs exceed the limits.

R2 Physicians and nurses can change the configuration.

R3 The infusion flow is controlled according to the configured doses for the current vital signs.

The glossary content (definitions, designations, domains, phenomena) can be directly derived from the facts and assumptions.
We validated manually that the following conditions are true:

- The domains and phenomena of the context diagram are consistent with R and D.
- Phenomena controlled by a biddable domain have counterpart phenomena located between machine and causal, lexical, or designed domains. To perform this check successfully, the following relationships are defined:
  - VitalSigns is a combination of Heartbeat, Pulse, and O2Saturation.
  - Alarm can be calculated from limits, Heartbeat, Pulse, and O2Saturation.
  - config is a combination of limits and dose.

All automatic validations have been performed.

13.2. Step A2 – Problem Decomposition

Fig. 13.2 depicts the problem diagram for requirement R1. It shows that the Terminal is constrained based on the information on the domains Patient and PatientSettings. The interface in the problem diagram with P!{VitalSigns} combines the interfaces in the context diagram with P!{Heartbeat} and P!{O2Saturation, Pulse}.

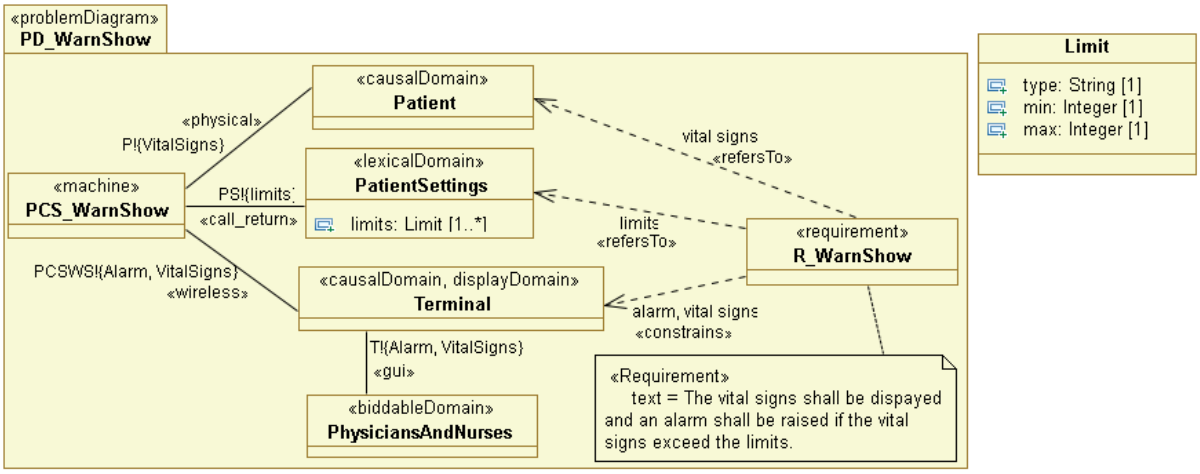


Figure 13.2.: *Problem Diagram WarnShow of Patient Care System*

Fig. 13.3 depicts the problem diagram for requirement R2. It shows that the PatientSettings are constrained based on the commands of the domain PhysiciansAndNurses. The connection domain Terminal of the context diagram is left out.

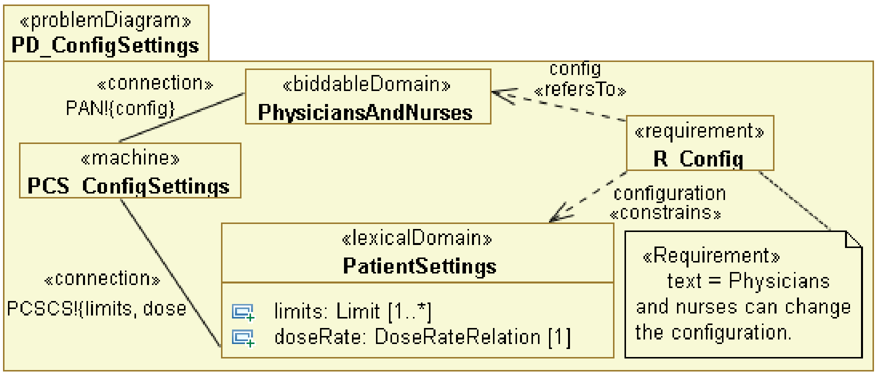


Figure 13.3.: *Problem Diagram ConfigSettings of Patient Care System*

Fig. 13.4 depicts the problem diagram for requirement R3. It shows that the InfusionPump is constrained based on the information on the domains Patient and PatientSettings. The interface in the problem diagram with P!{VitalSigns} combines the interfaces in the context diagram with P!{Heartbeat} and P!{O2Saturation, Pulse}.

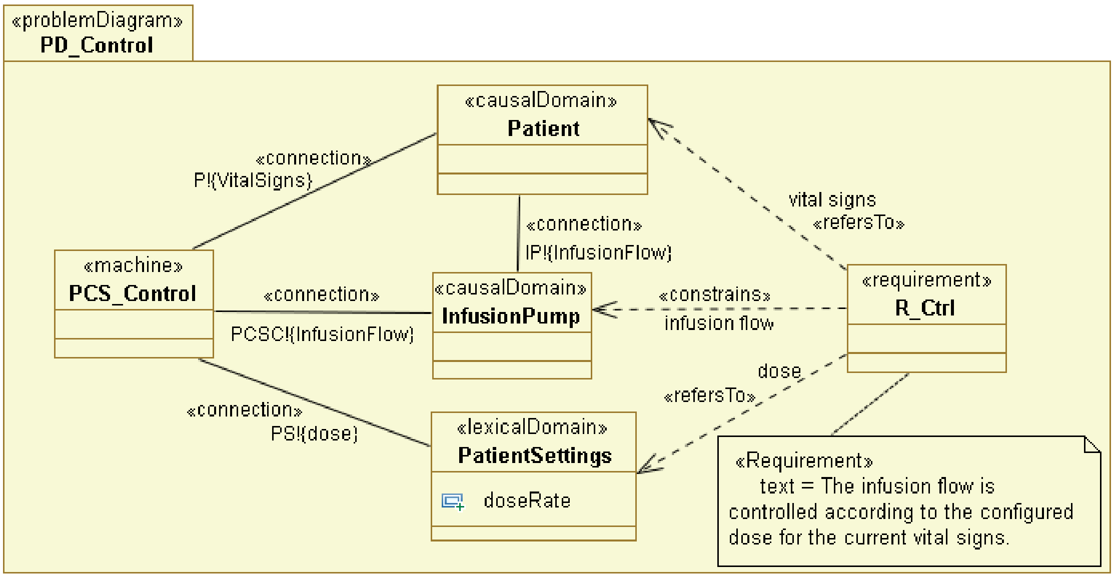


Figure 13.4.: *Problem Diagram* Control *of Patient Care System*

All subproblems are summarized in Table 13.1.

No	Requirement	≪refersTo≫	≪constrains≫
R1	The vital signs should be displayed, and an alarm should be raised if the vital signs exceed the limits.	Patient, PatientSettings	Terminal
R2	Physicians and nurses can change the configuration.	PhysiciansAndNurses	PatientSettings
R3	The infusion flow is controlled according to the configured doses for the current vital signs.	Patient, PatientSettings	InfusionPump

Table 13.1.: *Functional Requirements of Patient Care System*

Dependability requirements are associated with functional requirements, which we express using the stereotype ≪complements≫. For the functional requirements listed in Tab. 13.1, we initially identified some dependability requirements, as shown in Tab. 13.2 in $DR1$ - $DR4$ are expressed as proposed in Chapter 5. The required integrity ($DR1$ and $DR2$) supports the safety of the system, the required confidentiality ($DR3$) is necessary for privacy reasons, and the reliability required ($DR4$) is necessary for safety. We decide on generic mechanisms that represent solutions of these requirements.

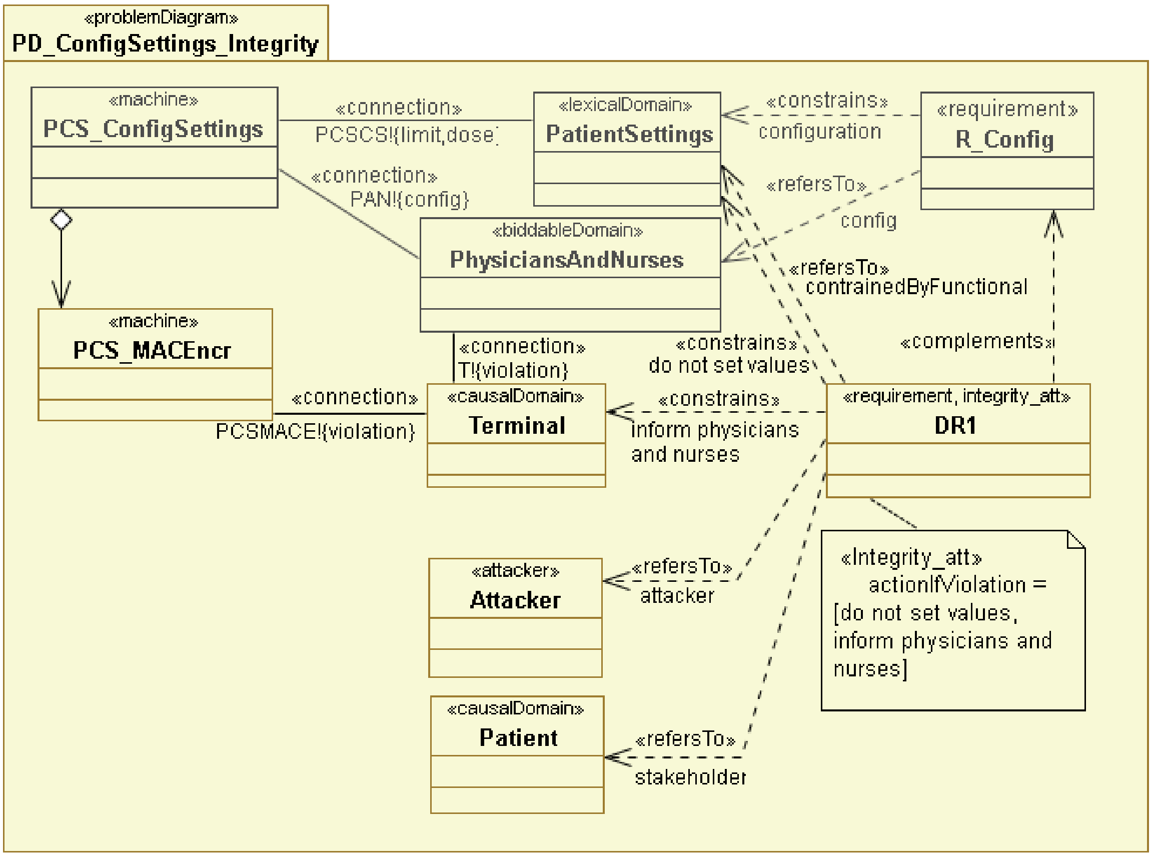


Figure 13.5.: *Problem Diagram ConfigSettings with Integrity Requirement*

Figure 13.5 exemplary shows how the problem diagram for the functional requirement depicted in Fig. 13.3 on Page 181 (gray boxes) is complemented with the integrity requirement considered an attacker. The requirement *DR*1 complements the *R*2 (R_Config) and the machine implementing the requirement PCS_MACEncr is part of the machine PCS_ConfigSettings in the problem diagram of the functional requirement. As shown in Tab. 13.1, the requirement refers to the Patient as the stakeholder and to the Attacker. It constrains the PatientSettings to avoid that values are set and the Terminal to inform the physicians and nurses.

To implement these mechanisms, additional domains have to be introduced, and additional requirements have to be fulfilled (see Fig. 13.6). We choose the security mechanism MAC (Message Authentication Code) for integrity and symmetric encryption for confidentiality. For the mechanisms MAC and encryption, a Shared Key known by the Terminal and by the PatientCareSystem is necessary. As required in Tab. 13.2, *DR*5, this Shared Key must be distributed to the Terminal and to the PCS. The integrity and confidentiality of the Shared Key must be preserved. This will be implemented using a key exchange protocol. For the key exchange, additional secrets (KE keys) are necessary. PhysiciansAndNurses must be able to set these secret in a protected environment (*R*4). The protected environment has to ensure that no attacker can read or modify the KE keys. To achieve the reliability requirement *DR*4 considering random faults, redundancy will be used in the architecture. For this mechanism, the input signal must be correct: The integrity of PatientSettings is ensured by using the checksum mechanism (*DR*6). To ensure that the correct values are set, the functional requirement is implemented in a way, that after setting the values, the new values are re-read from the PatientCareSystem. Redundant information in O2Saturation and Heartbeat help to detect sensor faults (*DR*7) and for the output, redundant FeedbackFlowSensors are used to detect a wrong infusion flow (*DR*8).

No	Dependability Requirement	≪compl-ements≫	≪refersTo≫	≪con-strains≫/ Mechanism
DR1	PatientSettings should be protected from modification for Patient against Attacker OR PatientSettings should be set and PhysiciansAndNurses should be informed.	R2	PatientSettings is asset, Terminal and WLAN know asset, Patient is stakeholder, against Attacker	Terminal PatientSettings/ MAC part of SSL protocol
DR2	Alarm and Vital Signs should be protected from modification for Patient against Attacker or PhysiciansAndNurses should be informed.	R1	Alarm and Vital Signs are assets, Terminal and WLAN know asset, Patient is stakeholder, against Attacker	Terminal/ MAC part of SSL protocol
DR3	PatientSettings, Alarm, and Vital Signs should be protected from disclosure for Patient against Attacker.	R1, R2	PatientSettings, Alarm, and Vital Signs are assets, Patient is stakeholder, against Attacker	WLAN/ encryption part of SSL protocol
DR4	The service (described in R3) with influence on InfusionPump should be reliable with a probability of $10^-8/h$.	R3	O2Sensor, HeartbeatSensor, PatientSettings, FlowFeedBackSensor	InfusionPump/ Redundancy of PatientCareSystem and input
DR5	The Shared Keys should be distributed to Terminal and PCS (for Patient) and Attacker should not be able to access Shared Keys.	R1, R2	Shared Keys are assets, Patient is stakeholder, against Attacker	WLAN/ key exchange of SSL (KE)
DR6	With a probability of $10^{-8}/h$, one of the following things should happen: the data of PatientSettings must be either correct, or PhysiciansAndNurses should be informed.	R3	PatientSettings	Terminal/ Checksum
DR7	With a probability of $10^{-8}/h$, one of the following things should happen: the value of O2Sensor must be either correct, or PhysiciansAndNurses should be informed.	R3	O2Sensor	Terminal/ Plausibility check with Heartbeat
DR8	The service with influence on the InfusionPump must be reliable with a probability of $10^{-8}/h$.	R3	FeedbackFlowSensor	Terminal/ Redundancy
DR9	With a probability of $10^{-8}/h$, one of the following things should happen: the value of FeedbackFlowSensor must be either correct, or PhysiciansAndNurses should be informed.	R3	FeedbackFlowSensor	Terminal/ Plausibility check with redundant FeedbackFlowSensor
R4	PhysiciansAndNurses should be able to set the KE Key in the PatientCareSystem.	-	PhysiciansAndNurses	KEKeyP / set in protected environment

Table 13.2.: *Dependability Requirements of Patient Care System*

The dependability requirements $DR1$ - $DR8$ have necessary conditions that can be fulfilled by other requirement or assumptions. We decided to assume the conditions stated in Tab. 13.3. Nevertheless, we describe the assumptions in the same way as we describe requirements.

The KE keys are distributed manually as described in Tab. 13.3, $DA1$. The corresponding associations is necessary for $R1$ and $R2$ and therefore we state that they complements $R1$ and $R2$. The assumption refers to KE Keys since they have to be distributed securely considering der Patient as a stakeholder and the described Attacker. No domain is constrained by this assumption since it is realized by a manual import in a protected area.

Integrity and confidentiality of the Infusion Flow and the PatientCareSystem (domains constrained by the assumption) have to be achieved. We decided that integrity and confidentiality of these domains for the Patient (stakeholder) and against the Attacker are ensured by

- physical protection (e.g., by reducing electromagnetic field (EMF) radiation and by protection against EMF radiation), and

No	Dependability Assumption	≪compl-ements≫	≪refersTo≫	≪constrains≫/ Mechanism
DA1	The KE keys are distributed to Terminal and PCS for Patient, and Attacker should not be able to access Shared Keys.	R1, R2	KE keys are assets, Patient is stakeholder, against Attacker	*none*/ manual import in physically protected area
DA2	Infusion Flow and PatientCareSystem are protected from modification for Patient against Attacker or Patient is informed.	R1, R2, R3	Infusion Flow and PatientCareSystem are assets, Patient is stakeholder, against Attacker	Infusion Pump, PatientCare-System/ physical protection (e.g., EMF) and protection by Patient
DA3	Infusion Flow and PatientCareSystem are protected from disclosure for Patient against Attacker.	R1, R2, R3	Infusion Flow and PatientCareSystem are assets, Patient is stakeholder, against Attacker	Infusion Pump, PatientCare-System/ physical protection (e.g., EMF) and protection by Patient
DA4	Terminal is protected from modification for Patient against Attacker or PhysiciansAndNurses are informed.	R1, R2	Terminal is asset, Patient is stakeholder, against Attacker	Terminal/ physical protection (e.g., EMF) and protection by PhysiciansAnd-Nurses
DA5	Terminal is protected from disclosure for Patient against Attacker.	R1, R2	Terminal is asset, Patient is stakeholder, against Attacker	Terminal/ physical protection (e.g., EMF) and protection by PhysiciansAnd-Nurses

Table 13.3.: *Domain Knowledge of Patient Care System*

- and protection by Patient (e.g., by preventing someone else from accessing the PatientCareSystem)

The assumptions are described by Tab. 13.3, *DA*2 and *DA*3. They are necessary for *R*1, *R*2, and *R*3.

Integrity and confidentiality of the Terminal (domains constrained by the assumption) have to be achieved. We decided that integrity and confidentiality of these domains for the Patient (stakeholder) and against the Attacker are ensured by

- physical protection (e.g., by reducing electromagnetic field (EMF) radiation and by protection against EMF radiation), and
- protection by PhysiciansAndNurses (e.g., by preventing someone else from accessing the Terminal)

The assumptions are described by Tab. 13.3, *DA*4 and *DA*5. They are necessary for *R*1 and *R*2.

The context diagram is extended with the domains necessary to fulfill the dependability requirements. This extension is depicted in a separate diagram. This diagram provides additional information (i.e. connections) about some domains in the context diagram and therefore a domain knowledge diagram (see Fig. 13.6). It adds redundant sensors measuring the infusion flow (FlowFeedBackSensors), additional phenomena (init, resp, xchd, see Fig. 7.7 on Page 117) for the key exchange between Terminal and PatientCareSystem, the SessionKeys, the key exchange keys

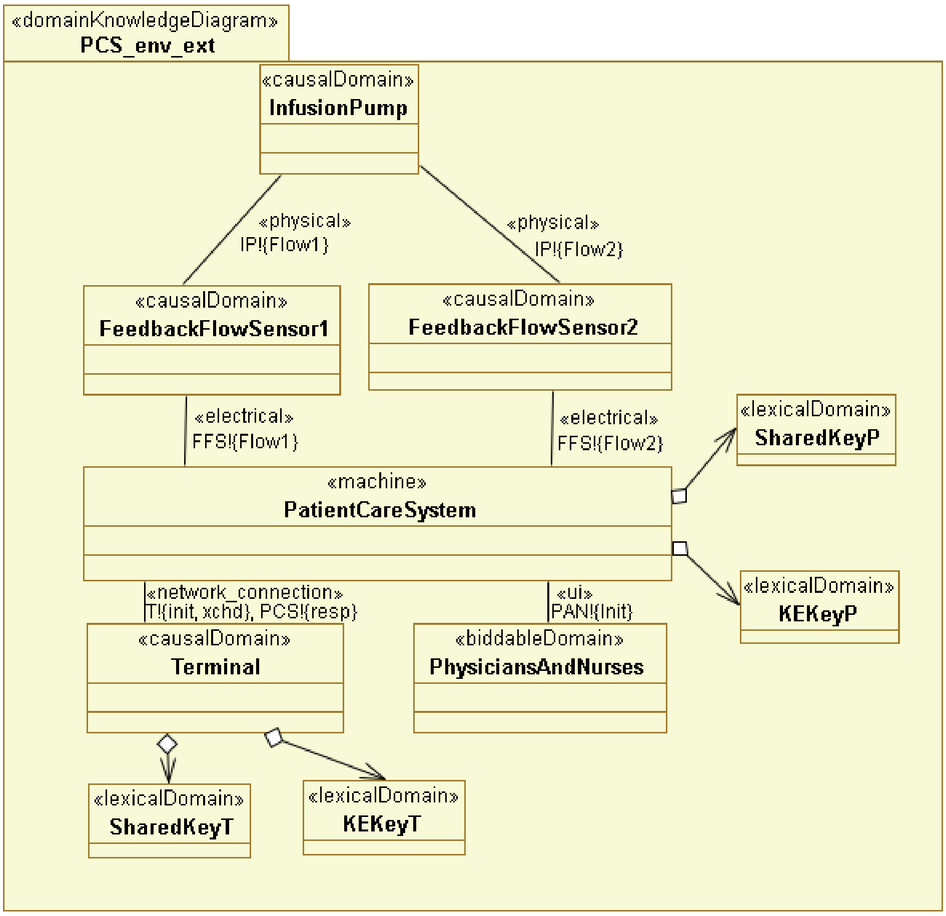


Figure 13.6.: *Domain Knowledge Diagram with Dependability Extension of Context Diagram*

(KEKeys), and an additional interface to PhysiciansAndNurses for an initial import of the KEKeys. We use no additional redundant sensor for measuring the O2 concentration and the heartbeat since these both sensors provide redundant information that can be used for plausibility checks and the patient should not be connected with additional sensors.

All functional and dependability requirements are implemented by a submachine. The glossary contains the following submachine names:

- **PCS_Show** submachine that implements requirement R1.
- **PCS_Config** submachine that implements requirement R2.
- **PCS_Control** submachine that implements requirement R3.
- **PCS_MACEncr** submachine that implements requirements DR1, DR2, and DR3.
- **PCS_KeyEx** submachine that implements requirement DR5.
- **PCS_PSInteg** submachine that implements requirement DR6.
- **PCS_InInteg** submachine that implements requirement DR7.
- **PCS_OutInteg** submachine that implements requirement DR9.
- **PCS_InitKey** submachine that implements requirement R4.

Requirement DR8 has to be realized by the overall machine using redundancy mechanisms.

The following new phenomena and domains have been introduced:

- **Init** phenomenon to initialize the KEKey.
- **init** phenomenon in initiate the key exchange protocol.
- **resp** phenomenon in answer the key exchange initialization.
- **xchd** phenomenon to finalize the key exchange protocol.

We performed all automatic checks for this step.

13.3. Step A3 – Abstract Machine Specification

The sequence diagram in Fig. 13.7 depicts the behavior for subproblem *WarnShow*. The VitalSigns are continuously forwarded to the PhysiciansAndNurses using the Terminal. If the VitalSigns exceed the limits, a warning is raised. To express the specification, the connection domains HeartbeatSensor and O2Sensor has to be depicted and the message VitalSigns need to be replaced by the messages Heartbeat and O2Saturation.

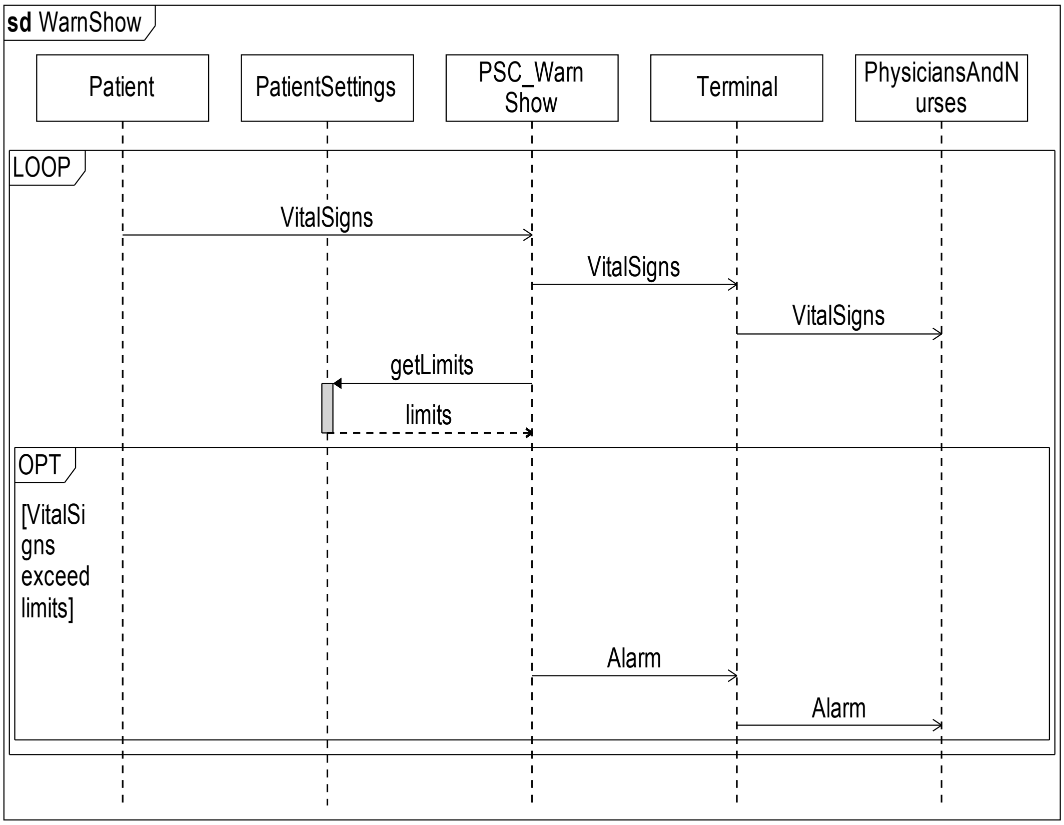


Figure 13.7.: *Sequence Diagram for Subproblem* WarnShow

The sequence diagram in Fig. 13.8 depicts the behavior for subproblem *ConfigSettings*. The machine stored the configuration with limits and dose for the patient in its PatientSettings.

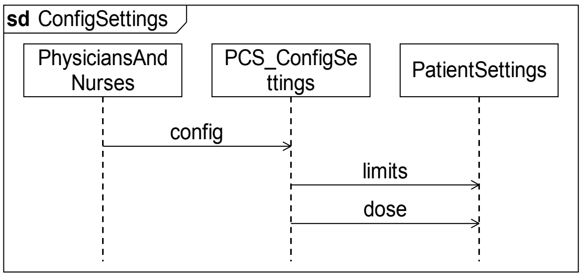


Figure 13.8.: *Sequence Diagram for Subproblem* ConfigSettings

The sequence diagram in Fig. 13.9 depicts the behavior for subproblem *Control*. The machine continuously sends an updated InfusionFlow to the InfusionPump. The flow is calculated based on the VitalSigns and the configured dose. To express the specification, the connection domains HeartbeatSensor and O2Sensor has to be depicted and the message VitalSigns needs to be replaced by the messages Heartbeat and O2Saturation.

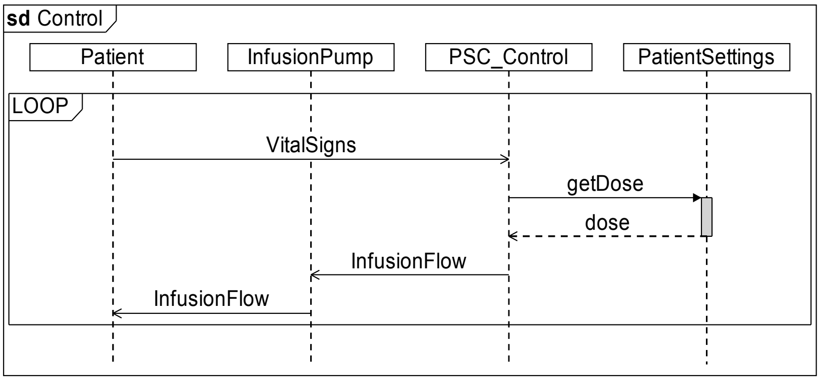


Figure 13.9.: *Sequence Diagram for Subproblem* Control

The result of applying the rules for creating a specification described in Section 7.3 to the context diagram of the patient care system shown in Fig. 13.1 on Page 180 is presented in Fig. 13.10. This UMLsec deployment diagram can be created following the command sequence depicted in Listing 13.1.

```
1 createDeploymentDiagram(’PCS_deployment’);
2 addSecureLinksStereotype(’PCS_deployment’,’default’);
3 createNodes(’PCS_deployment’);
4 createNestedClasses({’PatientSettings’});
5 createCommunicationPaths(’PCS_deployment’);
6 getNetworkConnections(); -- returns {’PCS!{Alarm,VitalSigns},T!{config}’}
7 setCommunicationPathType(’PCS_deployment’,’PCS!{Alarm, VitalSigns},
     T!{config}’,’encrypted’);
8 createDependencies(’PCS_deployment’);
```

Listing 13.1: *Generating a UMLsec Deployment Diagram*

First (line 1), the deployment diagram is created. In Line 2, to this deployment diagram the secure link stereotype with an attribute stating that adversary is a *default* adversary. In the next step (line 3), the nodes are created according to the domains in the context diagram. We decided that the PatientSettings should be shown as a nested class (line 4). In the next step (line 5), the communication paths are generated. The network connections without type are retrieved with getCommunicationPaths (line 6). For the returned connection

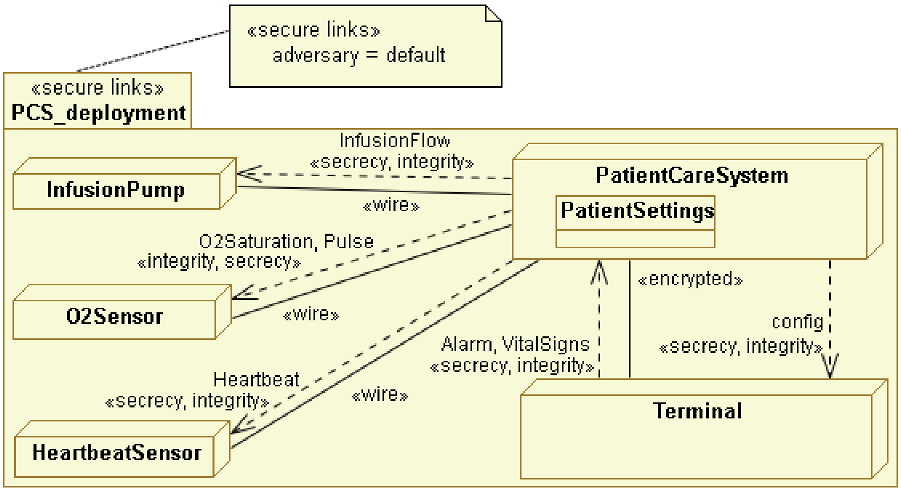


Figure 13.10.: *UMLsec Deployment Diagram Representing the Target State of Patient Care System*

(PCS!{Alarm,VitalSigns},T!{config}), we set the the communication path type to encryted (line 7) and at last the dependencies are created (line 8).

In the following, we show how to specify security mechanism by developing UMLsec diagrams based on security requirements. For each communication path contained in the UMLsec deployment diagram developed as shown in Fig. 13.10 on the preceding page that is not stereotyped ≪wire≫, we select an appropriate security mechanism according to the results of the problem analysis, e.g., MAC for integrity, symmetric encryption for security, and a protocol for key exchange (see Tab. 13.2 on Page 184). A security mechanism specification commonly consists of a structural and a behavioral description, which we specify based on the UMLsec ≪data security≫ stereotype.

We use the key exchange protocol (see Section 7.3) to realize the security requirement given in Table 13.2 on Page 184, *DR*4, of the patient care system. We use the protocol that secures data transmissions using MACs for the security requirements *DR*1 and *DR*2, and we use the protocol for symmetrically encrypted data transmissions for the security requirement *DR*3.

The model generation rule `createKeyExchangeProtocol(initiatorNodeName: String, responderNodeName: String, newPackage: String)` was presented in Hatebur et al. (2011) and can be found in Appendix D. The diagrams shown in Fig. 13.11 and 13.12 for the patient care system haven been created with `createKeyExchangeProtocol('Terminal', 'PatientCareSystem', 'PCS_KeyExchProt')`.

In the created model (see in Fig. 13.11), the tag secrecy of the ≪critical≫ class Terminal contains the secret s_, which represents an array of secrets to be exchanged in different rounds of this protocol. It also contains the private key inv(K_T) of the Terminal. Next to these assets, the integrity tag additionally contains the nonces N_ used for the protocol, the public key K_T of the Terminal, the public key K_CA of the certification authority, and the round iterator i. These tag values are reasonable because the security domain knowledge in Tab. 13.3, *DR*2 and *DR*3 states that the PatientCareSystem with its contained data is kept confidential and its integrity is preserved. The tag secrecy of the ≪critical≫ class PatientCareSystem contains the session keys k_ and the private key inv(K_P) of the PatientCareSystem. The integrity tag of the PatientCareSystem consists of assets similar to the ones of the same tag of the Terminal. Integrity and confidentiality of the data stored in the Terminal (private key inv(K_P), the public key K_R,

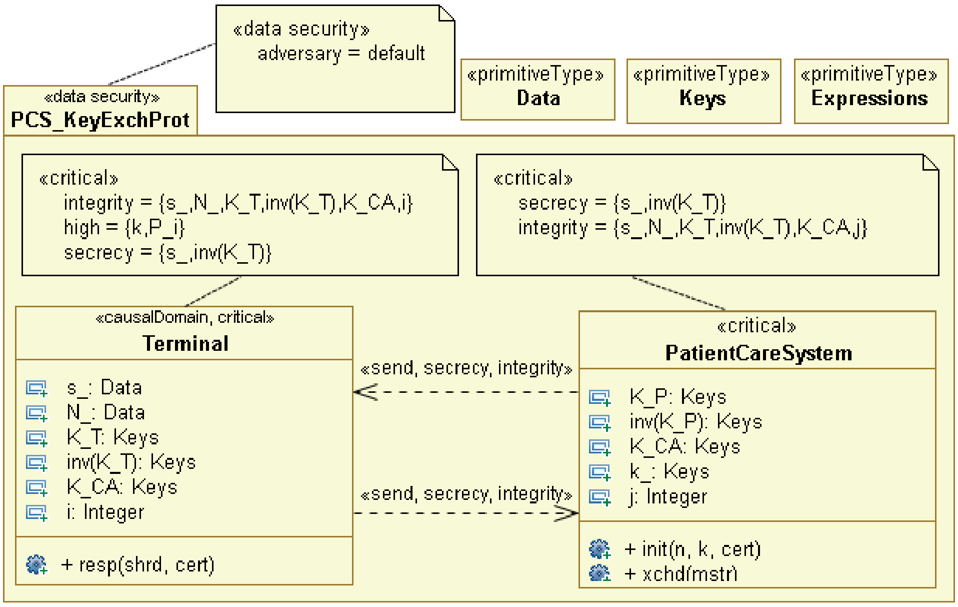


Figure 13.11.: *Class Diagram of Key Exchange Protocol for Patient Case System*

the public key K_CA of the certification authority, and the round iterator j) are covered by the domain knowledge in Tab. 13.3, *DR*4 and *DR*5.

The attributes correspond to the domains already defined in Step A2 - Problem Decomposition in the diagram in Fig. 13.6:

- The attribute s_ corresponds to the domain SharedKeyT.
- N_ represents the nonce generated by the Terminal in order to prevent replay attacks.
- The attributes K_T, inv(K_T) and inv(K_CA) are stored in the domain KEKeyT.
- The attributes K_P, and inv(K_P) are stored in the domain KEKeyP.
- The attribute k_ corresponds to the domain SharedKeyP.
- The attributes i and j are internal attributes of Terminal and PatientCareSystem.

The sequence diagram in Fig. 13.12 specifies three messages and two guards, and it considers the ith protocol run of the Terminal, and the jth protocol run of the PatientCareSystem. It is expressed in UML notation with side comments according to the description of Jürjens (2005). A translation of the guards in the line comments into combined fragments in UML2 is shown in Fig. 13.13. The sequence counters i and j are part of the Terminal and the PatientCareSystem, respectively. The init(. . .) message sent from the Terminal to the PatientCareSystem initiates the protocol. If the guard at the lifeline of the PatientCareSystem is true, i.e., the key K_T contained in the signature matches the one transmitted as clear text, then the PatientCareSystem sends the message resp(. . .) to the Terminal. If the guard at the lifeline of the Terminal is true, i.e., the certificate is actually for S and the Terminal returns the correct nonce is returned with the message xchd(. . .) to the PatientCareSystem. If the protocol is executed successfully, i.e., the two guards are evaluated to true, then both parties share the secret s_i.

The key exchange protocol only fulfills the corresponding security requirements if integrity, confidentiality, and authenticity of the keys are ensured. According to our pattern system for security requirements engineering (Hatebur & Heisel, 2010b), applying the key exchange

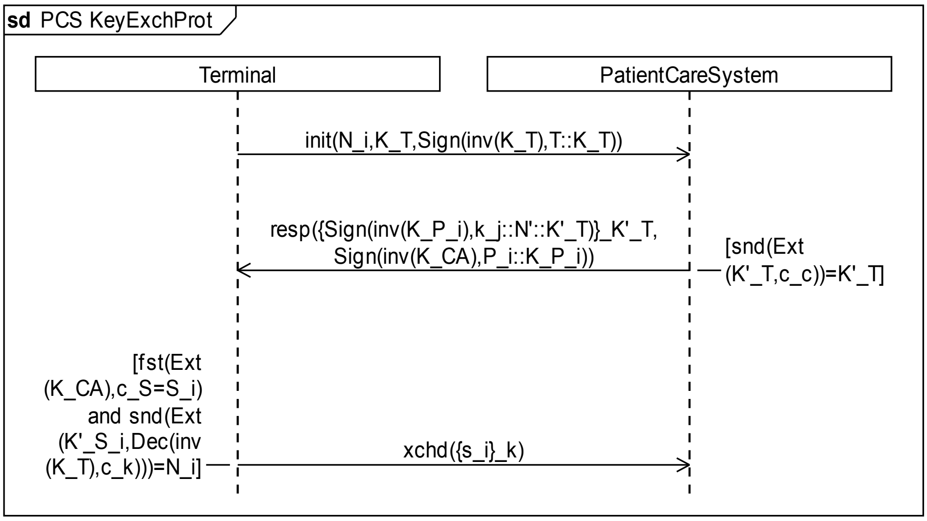


Figure 13.12.: *Sequence Diagram of Key Exchange Protocol for Patient Care System in UML 1.4 notation with UMLsec extension*

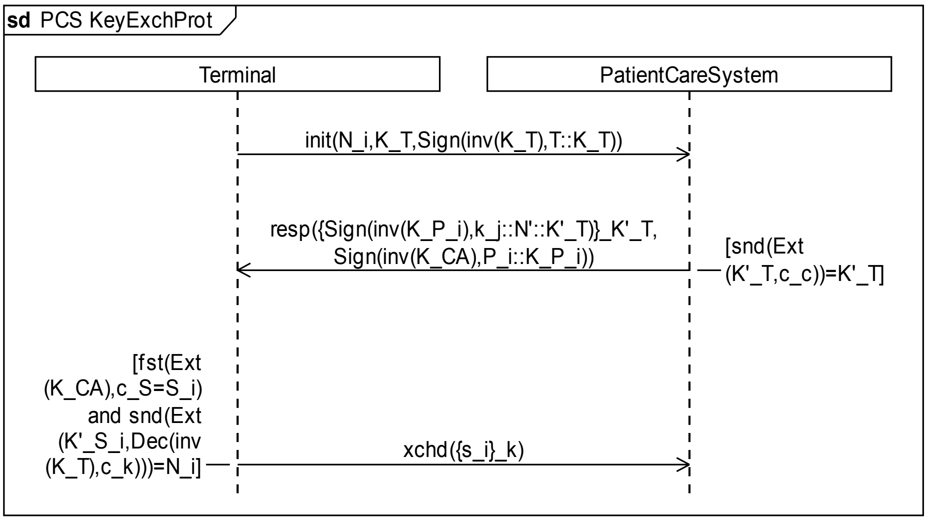


Figure 13.13.: *Sequence Diagram of Key Exchange Protocol for Patient Care System in UML 2 notation*

mechanism leads to dependent statements about integrity, confidentiality, and authenticity of the keys as stated in Tab. 13.3 on Page 185.

The specifications (given in Fig. 13.7 on Page 188, 13.8 on Page 188, 13.16 on Page 194, and 13.12 on the previous page) is created with messages that directly correspond to the phenomena of the context diagram. We found no contradictions in the abstract specification and the domain knowledge. We performed all automatic checks for this step to show that messages and phenomena are consistent and for each subproblem there exists at least one sequence diagram. For each subproblem there exists one normal case scenario. Exceptional cases are not considered for this case study.

13.4. Step A4 – Technical Context Diagram

The technical context diagram and the context diagram in Fig. 13.1 on Page 180 together with the domain knowledge diagram in Fig. 13.6 on Page 186 are the same. No new phenomena and domains have been introduced.

13.5. Step A5 – Operations and Data Specification

For each abstract submachine specification in Section 13.3, we developed the class model for the operation specification as depicted in Figures 13.14 on the facing page, 13.15 on the next page, and 13.16 on Page 194 and the operation specifications as presented in the following paragraphs. The procedure is described in Appendix A.

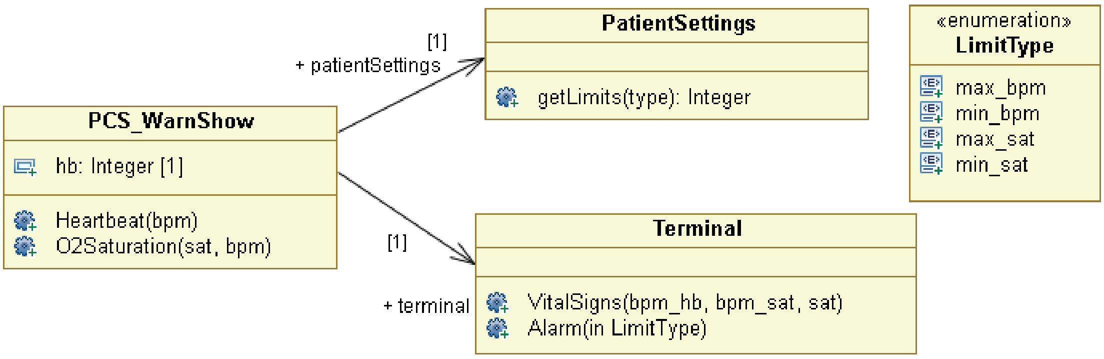


Figure 13.14.: *Class Diagram for Operation Specification for R1*

In the submachine PCS_WarnShow the Heartbeat value is stored when a new value is received in the parameter bpm (precondition is true and postcondition is hb=bpm). The precondition of PCS_WarnShow::O2Saturation is true. When a O2Saturation is received, the limits are retrieved using getLimits and depending on the configured limits warnings are sent to the terminal (see Listings 13.2).

```
terminal^VitalSigns(hb, bpm, sat) and
patientSettings.getLimits(LimitType::max_bpm)>hb implies
    terminal^Alarm(LimitType::max_bpm) and
patientSettings.getLimits(LimitType::max_bpm)>bpm implies
    terminal^Alarm(LimitType::max_bpm) and
patientSettings.getLimits(LimitType::min_bpm)<hb implies
    terminal^Alarm(LimitType::min_bpm) and
patientSettings.getLimits(LimitType::min_bpm)<bpm implies
    terminal^Alarm(LimitType::min_bpm) and
patientSettings.getLimits(LimitType::max_sat)>sat implies
    terminal^Alarm(LimitType::max_sat) and
patientSettings.getLimits(LimitType::min_sat)<sat implies
    terminal^Alarm(LimitType::min_sat)
```

Listing 13.2: *Postcondition for PCS_WarnShow::O2Saturation*

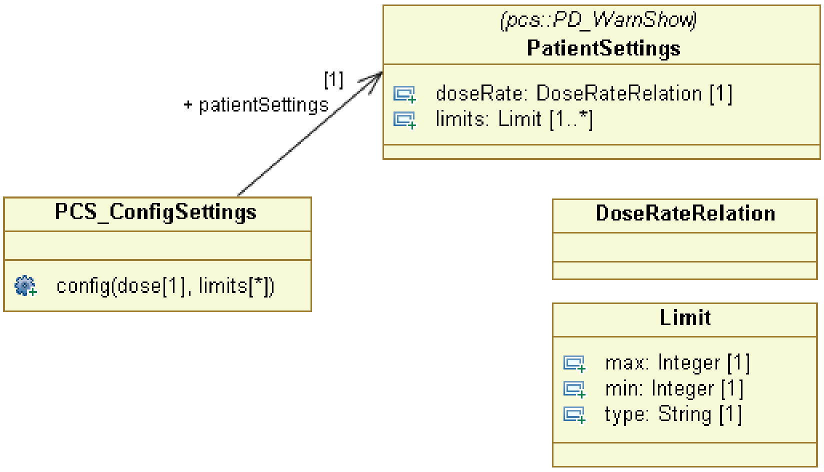


Figure 13.15.: *Class Diagram for Operation Specification for R2*

When the machine PCS_ConfigSettings receives a configuration message, the values are stored in the PatientSettings (precondition is true and postcondition is shown in Listings 13.3).

```
patientSettings.limits->includesAll(limits) and patientSettings.doseRate = dose
```

Listing 13.3: *Postcondition for PCS_ConfigSettings::config*

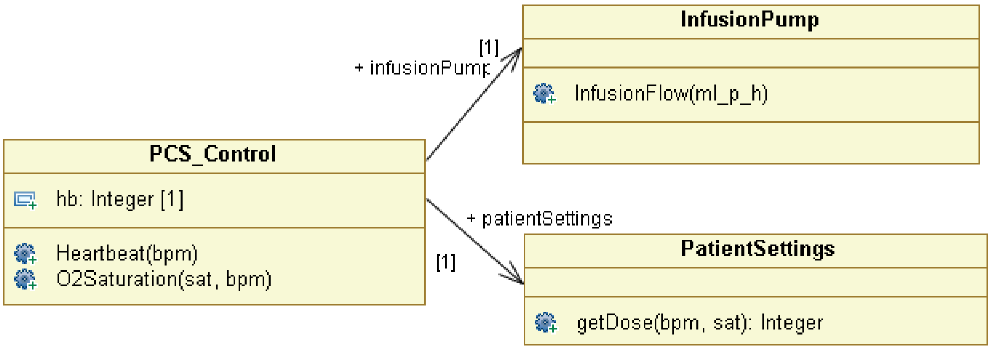


Figure 13.16.: *Class Diagram for Operation Specification for R3*

In the submachine PCS_Control the Heartbeat value is stored when a new value is received (precondition is true and postcondition is hb=bpm). When a O2Saturation is received, the configured dose is retrieved using getDose. This dose is sent to the InfusionPump (precondition is true and postcondition is shown in Listings 13.4).

```
infusionPump^InfusionFlow(patientSettings.getDose(hb,sat))
```

Listing 13.4: *Postcondition for PCS_Control::O2Saturation*

The behavioral description for the submachines PCS_MACEncr, PCS_PSInteg, PCS_InInteg, and PCS_OutInteg is intentionally omitted, because the basic idea is similar.

In the glossary, the following terms are defined:

- **hb** is an attribute in which the heartbeat from the HeatbeatSensor is stored.
- **getLimits** is an operation that returns the limit requested by the parameter.
- **getDose** is an operation that returns the configured dose for a given heartbeat value and a O2 saturation value.

We checked that

- exactly the operations (Heartbeat, O2Saturation, and config) occurring in the abstract specification in Step A3 - Abstract Machine Specification are described,
- for each described operation, a pre- and postcondition exist,
- the pre- and postconditions expressed in OCL are syntactically correct,
- the postcondition covers all cases exhibited in the abstract specification of Step Abstract Machine Specification,
- all parameters of operations are known by the caller and all parameters of sent messages must be known by the machine,
- all parameters are used in the pre- and/or postcondition,

- the operation specification is consistent with class model of the machine state, and

- all classes, associations, and attributes newly introduced in the class model are motivated by some operation specification.

13.6. Step A6 – Machine Life-Cycle

The relationship of the abstract specifications can be described by the following expression:

LifeCycle = InitKey; (KeyExchProt (ConfigSettings* ∥ WarnShow ∥ Control))*

The behavior of the submachines PCS_MACEncr, PCS_PSInteg, PCS_InInteg, and PCS_OutInteg are only extensions of the behavior described in ConfigSettings, WarnShow, and Control, as described in A2.

We checked that

- each sequence diagram of Step Abstract Machine Specification is contained in at least one life-cycle expression,

- for each biddable domain exactly one life-cycle exists,

- the life-cycles are consistent with the state predicates in Step Abstract Machine Specification,

- the life-cycles are consistent with the pre- and postconditions in Step Operations and Data Specification, and

- exactly one life-cycle exists for the machine domain, that combines all life-cycles.

13.7. Step D1a – Initial Architecture

In the initial software architecture shown in Fig. 13.17, exactly the machines in the subproblem diagrams are used as components. Additionally, the lexical domains PatientSettings, KEKeyP, and SharedKeyP become components of the initial architecture. All components are connected in the same way as the domains in the problem diagrams, e.g., PCS_WarnShow is connected with the PatientSettings, with the ports to the external domains HeartbeatSensor and O2Sensor, and via the component for encryption and MAC protection PCS_MACEncr with the port to the Terminal.

The connectors to external interface have the same types as the associations in the technical context diagram. All internal connectors represent software interfaces and have the stereotype ≪call_return≫.

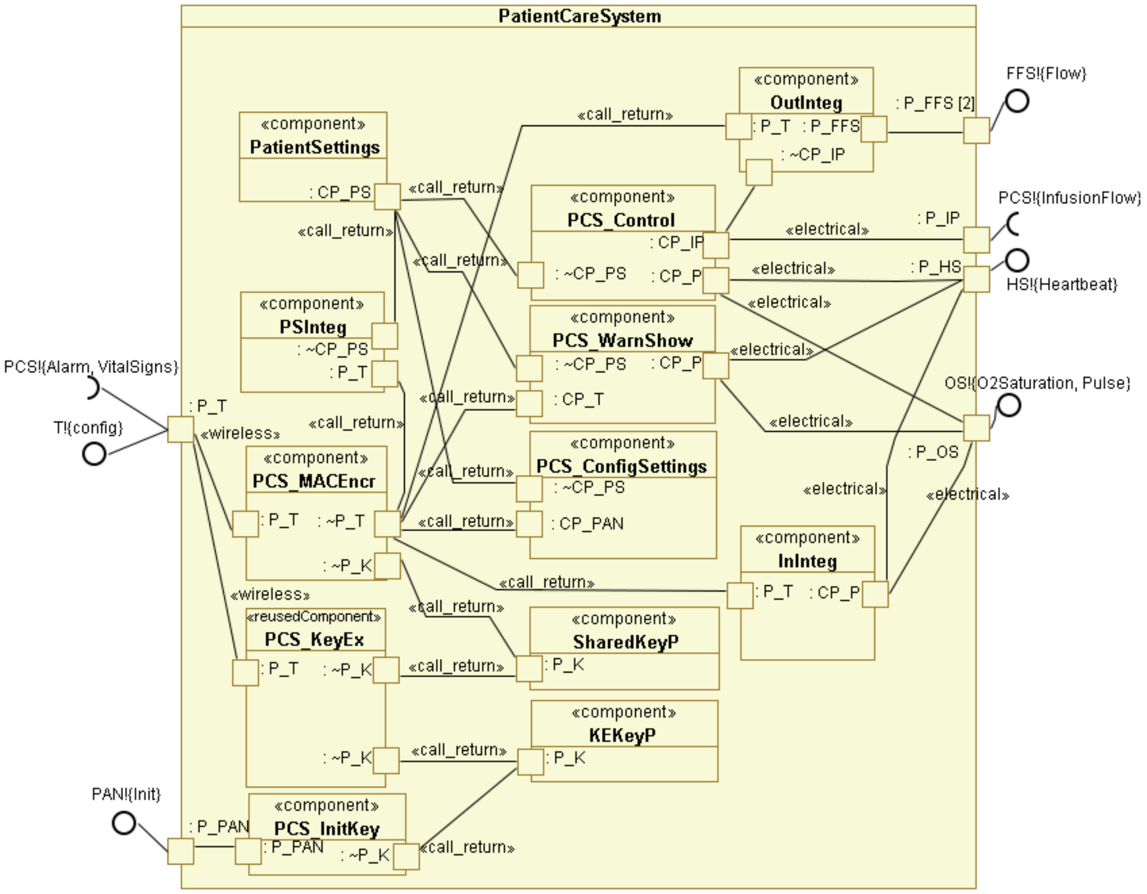


Figure 13.17.: *Initial Architecture for Patient Care System*

The external ports of the machine are defined based on the interfaces in the context diagram (see Fig. 13.1 on Page 180) and the domain knowledge diagram extending the context diagram (see Fig. 13.6 on Page 186). The ports of the components are defined based on the interfaces in the problem diagrams (see Tables 13.1 and 13.2). The external ports of the machine from the context diagram are depicted in Fig. 13.18. The port type P_P[1] for the patient information implements the same interface as the port types P_HS and P_OS for the sensors. All ports starting with ˜ are inverse port types: they use the implemented interfaces and implement the used interfaces of the port types with same name without the prefix ˜. The port types P_T for the terminal and P_IP for the infusion pump are also defined using the interfaces from the context diagram. The port type P_K is used for all cryptographic keys. The port type P_FFS for the feedback flow sensor and P_PAN for the direct interface to physicians and nurses in a protected environment are defined using the interfaces in the domain knowledge diagram extending the context diagram.

[1] For external ports, we use the prefix P_ followed by the abbreviation of the connected domain.

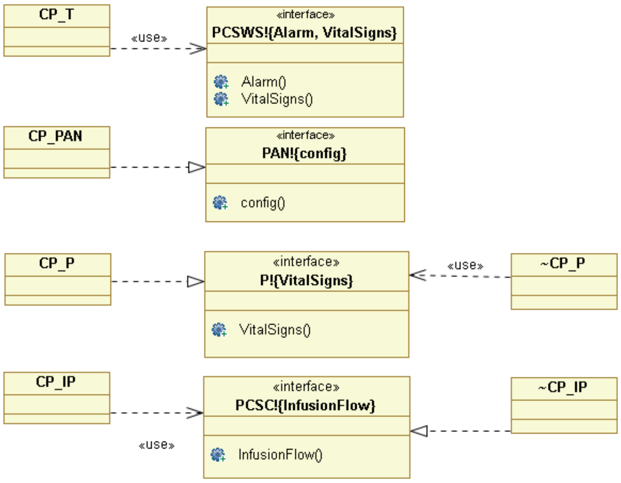


Figure 13.18.: *Port Types for Patient Care System*

The port type CP_PS for the patient settings is depicted in Fig. 13.19. It is derived from the problem digram using exactly the rules described in Appendix A.7.

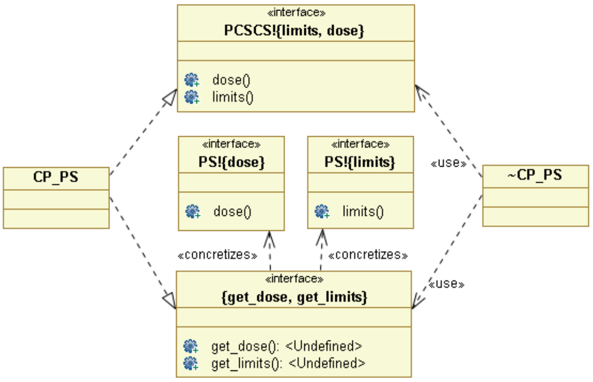


Figure 13.19.: *Component Port Type CP_PS for Patient Care System*

The port type CP_T for the terminal, CP_PAN for the direct interface to the physicians and nurses in a protected environment, CP_P for the interface to the patient, and CP_IP for the interface to the infusion pump are depicted in Fig. 13.20. They are defined using the interfaces of the problem diagrams in Tables 13.1 on Page 182 and 13.2 on Page 184.
No new entry is added to the glossary because the component names are already included as subproblem machines.
We checked that

- for each provided or required interface of machine ports in the architecture, there exists a corresponding interface in the technical context diagram,
- there is one component for each submachine related to the given machine of the (technical) context diagram as well as for all relevant lexical domains found in the problem diagrams.
- the internal components have the stereotype ≪Component≫ or ≪ReusedComponent≫,
- the internal components are connected to each other and to the external ports according to the connections in the technical context diagram / subproblems, and

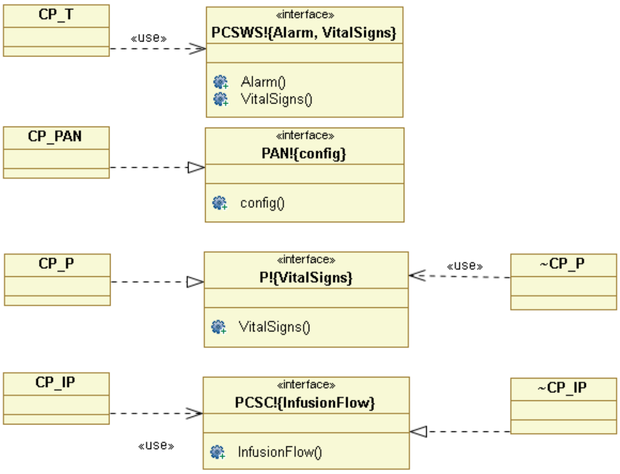


Figure 13.20.: *Further Component Port Types for Patient Care System*

- the stereotypes of the connectors are consistent to the associations of the technical context diagram.

13.8. Step D1b – Implementable Architecture

In the implementable architecture, we apply the pattern for redundancy to achieve the reliability requirement (see Requirement DR8)). This pattern consists of components for two channels and a kind of voter component for the actuator (ChooseActiveChannel). The channels supervise each other (using the port of type P_Lifeness) and are able to select the active channel (using the port of type P_ChooseChannel).

To instantiate this pattern, we connected the output port of each channel that controls the infusion pump (P_IP) to the component ChooseActiveChannel. All other ports of the single channels are connected to the machine ports as in the initial architecture.

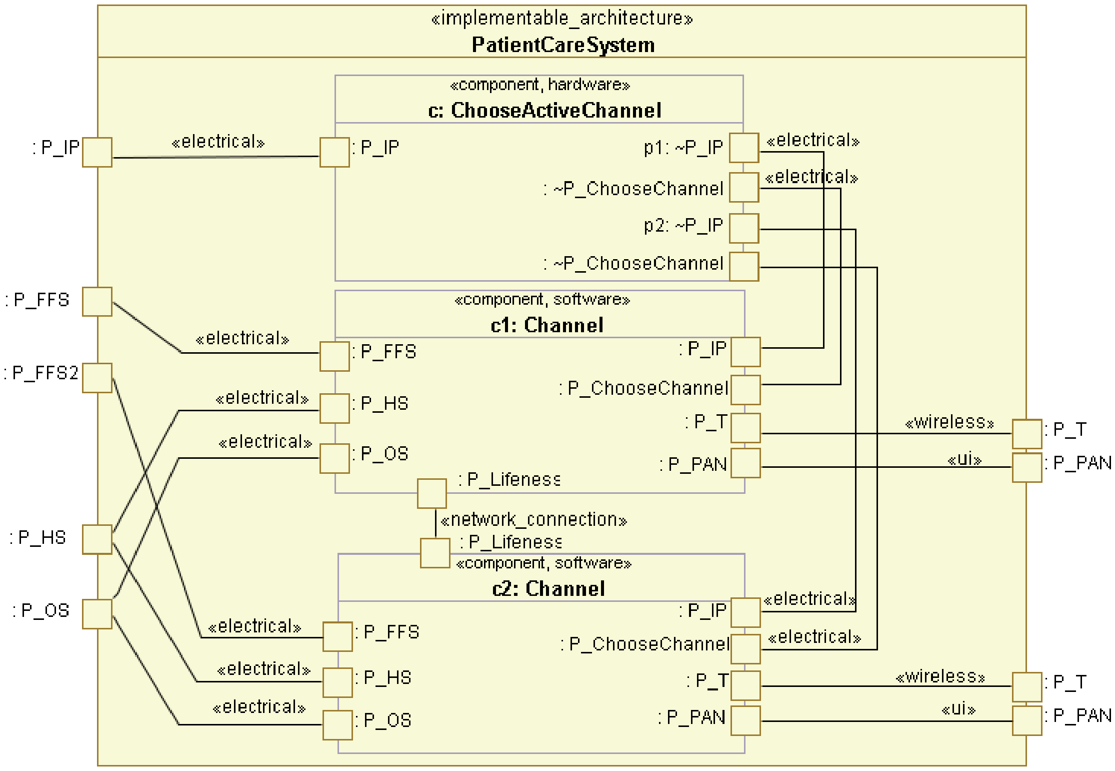


Figure 13.21.: *Implementable Architecture for Patient Care System*

To realize the redundancy concept, we need the port types P_ChooseChannel, its inverse type ∼P_ChooseChannel, and P_Lifeness on the highest architectural level (see Fig 13.22).

Additionally, each channel needs an interface to the hardware that can be used to switch of the channel in case of a critical fault (see CP_HW in Figures 13.22 and 13.23). This is used by the component PCS_RedundencyHandler that monitors the other channel, switches off the channel, and activates the correctly working channel. The other parts of the architecture of each channel (depicted in Fig. 13.23) can be directly derived from the initial architecture. The ports of type P_OS and P_HS are connected to several components. Therefore, we introduce the facade Sensor_Facade. The port of type P_T is connected to the components PCS_KeyEx and PCS_MACEncr. Therefore, we introduce the facade Terminal_Facade.
New components we add to the glossary are

- **PCS_RedundencyHandler**
- **Terminal_Facade**
- **Sensor_Facade**

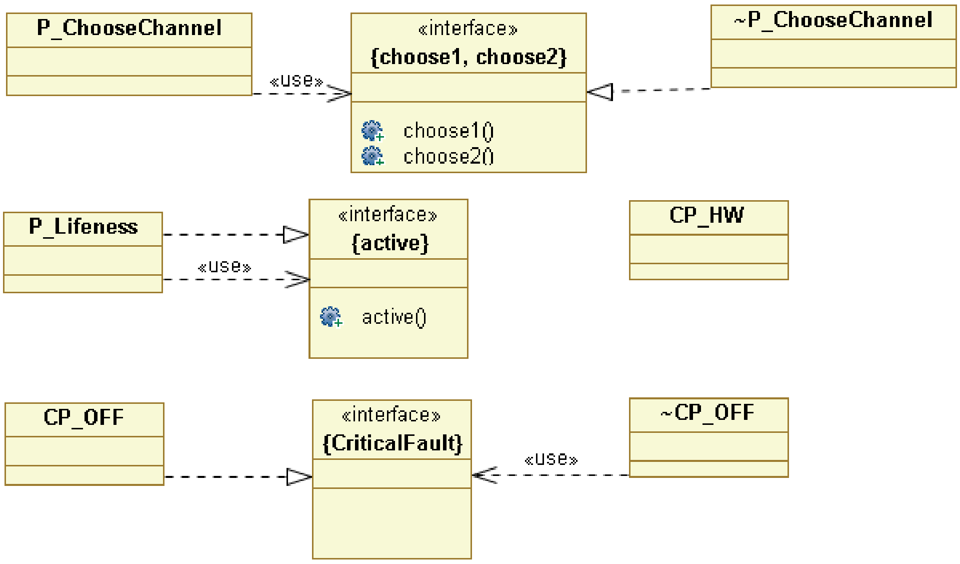


Figure 13.22.: *Port Types for Implementable Architecture of Patient Care System*

with the functionality as describe above.
We checked that

- the facade has port types that correspond to the external ports of the initial architecture,
- the facade has ports corresponding to the component ports. The port types are the inverse of the component ports,
- a concretizes dependency exists from the implementable architecture to the initial architecture (only in model, not depicted in this document), and
- the stereotypes of the connectors to the facade are the same as in the initial architecture.

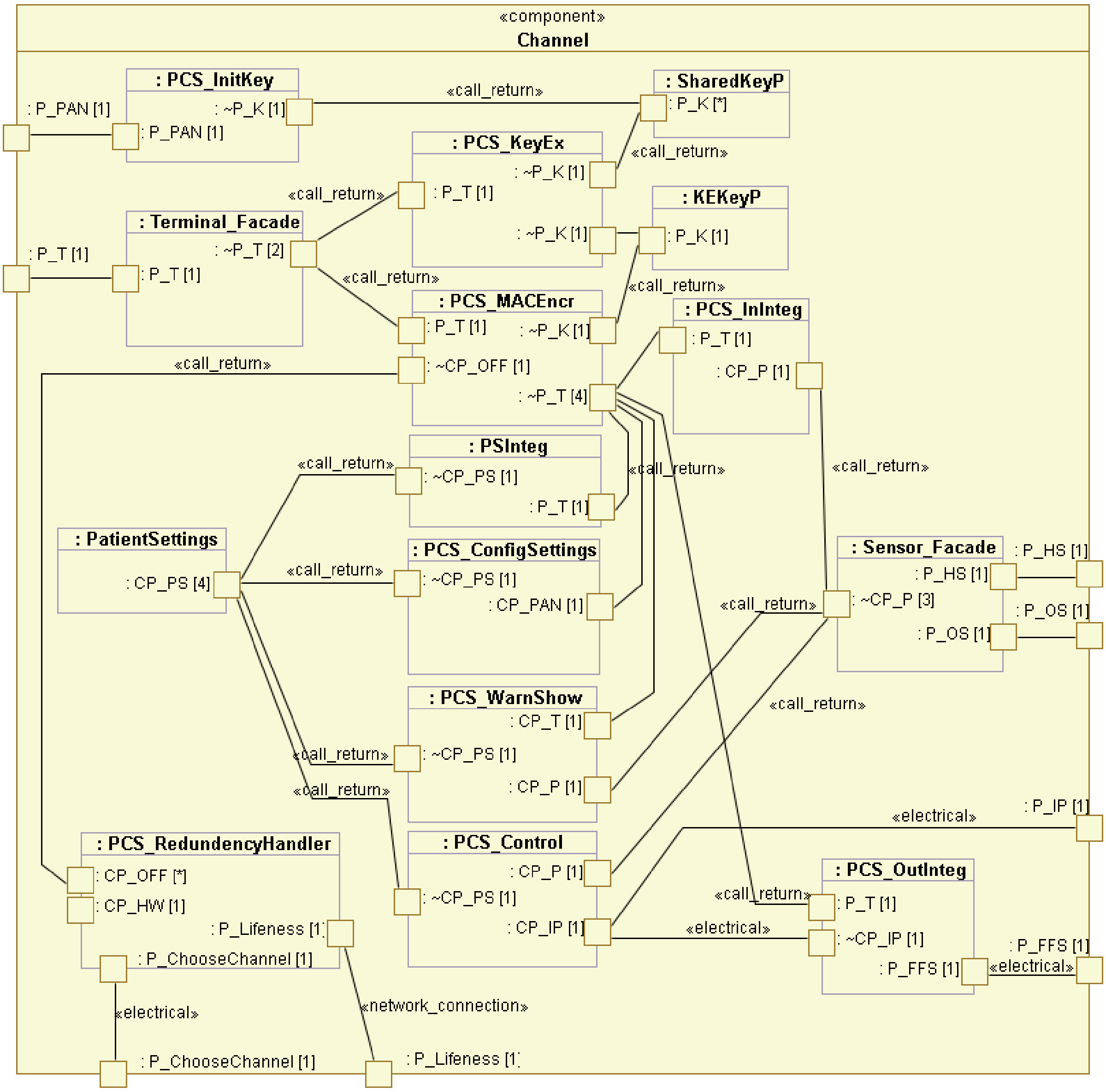


Figure 13.23.: *Implementable Architecture for one Channel of Patient Care System*

13.9. Step D1c – Re-structure Software Architecture

For each channel, we can create a layered architecture as shown for the CACC case study in Section 8.2. The following components could be part of the application layer:

- PCS_InitKey is in the application layer since it realizes one of the main functions of the machine (and bridges the interface abstraction layer).
- PCS_ConfigSettings is in the application layer since it realizes one of the main functions of the machine.
- PCS_WarnShow is in the application layer since it realizes one of the main functions of the machine.
- The control part of PCS_Control is in the application layer since it realizes one of the main functions of the machine.

The following components could be part of the interface abstraction layer (containing adapters):

- Terminal_Facade (functionality part) is an adapter since it converts the technical operations into logical operations).
- PCS_MACEncr is an adapter since it realizes complementary functionality for PCS_ConfigSettings, PCS_WarnShow, and PCS_Control.
- InInteg is an adapter since it realizes complementary functionality for PCS_WarnShow and PCS_Control.
- PSInteg is an adapter since it realizes complementary functionality for PCS_WarnShow and PCS_Control.
- Parts of OutInteg are in the interface abstraction layer since they realize complementary functionality for PCS_Control and generates warnings using PCS_MACEncr.
- Parts of the SensorFacade are in the interface abstraction layer since the values provided by the sensors has to be converted to the application and provided to several components.
- The parts of the RedundancyHandler not accessing the hardware directly are in the interface abstraction layer since they realize complementary functionality for PCS_Control.
- The parts of PCS_Control realizing the conversion are in the interface abstraction layer.

The following components could be part of the hardware abstraction layer:

- The parts of the SensorFacade that are used to directly access the hardware sensors.
- Parts of OutInteg are used to directly access the hardware sensor for feedback.
- SharedKeyP, KEKeyP, and PatientSettings are components storing the keys or settings and therefore access the hardware.
- The hardware-dependent part of the Terminal_Facade are in the hardware abstraction layer.
- The parts of PCS_Control accessing the hardware are in the hardware abstraction layer.
- The parts of RedundancyHandler accessing the hardware are in the hardware abstraction layer.

In the glossary, the following terms are defined:

- **InfusionFlow_HAL**: parts of PCS_Control accessing the hardware
- **Terminal_HAL**: hardware dependent part of the Terminal_Facade
- **FlowSensor_HAL**: parts of OutInteg for accessing the hardware
- **HB_HAL**: parts of SensorFacade for accessing the heartbeat sensor
- **O2_HAL**: parts of SensorFacade for accessing the O2 saturation sensor
- **Mem_ALU_HAL**: parts of RedundancyHandler for accessing the hardware for fault checking
- **Off_HAL**: parts of RedundancyHandler for switching off the channel

- **Sensor_Adapter**: conversion parts of SensorFacade
- **Terminal_Adapter**: conversion parts of Terminal_Facade

We checked that

- all components of the implementable architecture are contained in the layered architecture, and
- the connectors connected to the ports in the layered architecture have the same stereotypes or more specific ones than in the implementable architecture.

13.10. Step D2 – Inter-Component Interaction

Breaking down the sequence diagrams for the specifications expressed in Sections 13.3 and 13.5 to derive the interface behavior of all components for all operations is a routine task the regular development[2]. Since the redundancy mechanism is applied (see Section 13.2 and Chapter 9), the protocol for handling faults in a single channel has to be specified. The protocol presented in Figures 13.24 on Page 205 and 13.25 on Page 206 can be abstracted and used as a pattern.

In Fig. 13.24 on Page 205, first the normal behavior is described. The channel c1 is active and channel c2 is passive. Both channels receive the sensor values (O2Saturation, Pulse, clHeartbeat) and calculate the InfusionFlow. The component ChooseActiveChannel forwards the InfusionFlow of channel c1 to the infusion pump in the environment. Each channel also sends the message active to the other channel. If an error is detected in channel c1, e.g. a memory error (Mem error), channel c1 tries to send the command choose2 to the component ChooseActiveChannel, raises an alarm and stops working. If channel c2 receives no message active (due to an error), it also sends the command choose2 to the component ChooseActiveChannel and raises an alarm. It is then the active channel. The sensor values (O2Saturation, Pulse, clHeartbeat) are sent to both channels, but only channel c2 calculates the InfusionFlow and sends the message active to the other channel.

In Fig. 13.25 on Page 206, again, first the normal behavior is described. The channel c1 is active and channel c2 is passive. In this figure, the scenario with an detected error in channel c2 is described. In this case, channel c2 tries to raise an alarm. If channel c1 receives no message active, it also raises an alarm. It is then the active channel. The sensor values (O2Saturation, Pulse, clHeartbeat) are sent to both channels, but only channel c2 calculates the InfusionFlow and sends the message active to the other channel.

No new messages have been introduced. In the glossary the following state predicates have been defined:

- **active**: The channel is active and the output is forwarded to the infusion pump.
- **passive**: The channel is passive and its output is <u>not</u> forwarded to the infusion pump.
- **off**: The channel is switched off.

The following validations conditions have been checked for the redundancy protocol and have to be checked for the diagrams not shown in this thesis.

- The sequence diagrams must be consistent with the behavior described in Step Abstract Machine Specification and in Step Machine Life-Cycle.
- The sequence diagrams must realize the operations described in Step Operations and Data Specification.

[2] development of systems without dependability requirements

- All messages in the application interface classes of Step Machine Architecture must be used in some sequence diagram.
- The direction of messages must be consistent with the required and provided interfaces of Step Machine Architecture.
- The messages must connect components as connected in the software architecture of Step Machine Architecture
- It must be possible to relate any new state predicates to the state predicates of Step Abstract Machine Specification

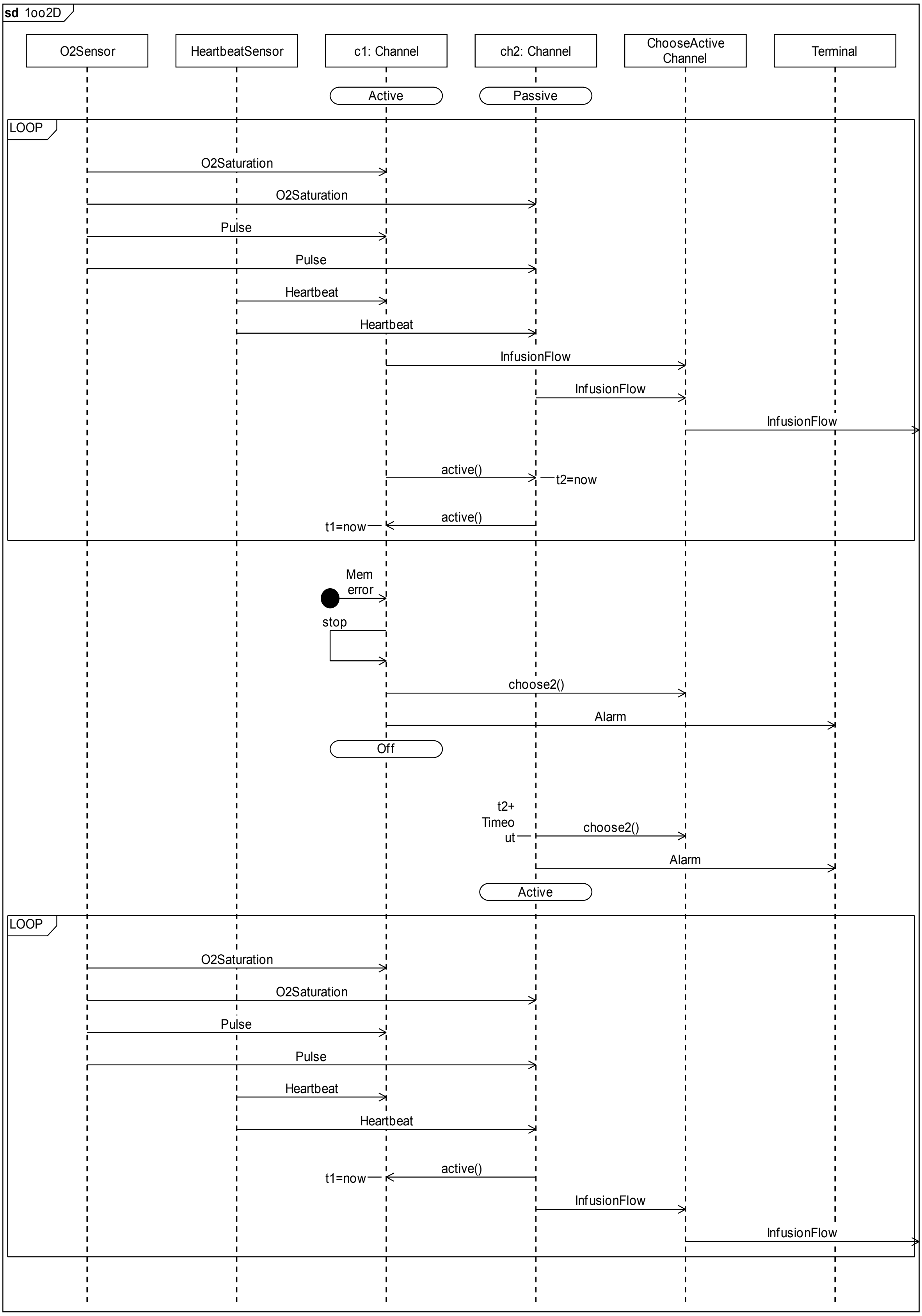


Figure 13.24.: *Redundancy Protocol 1*

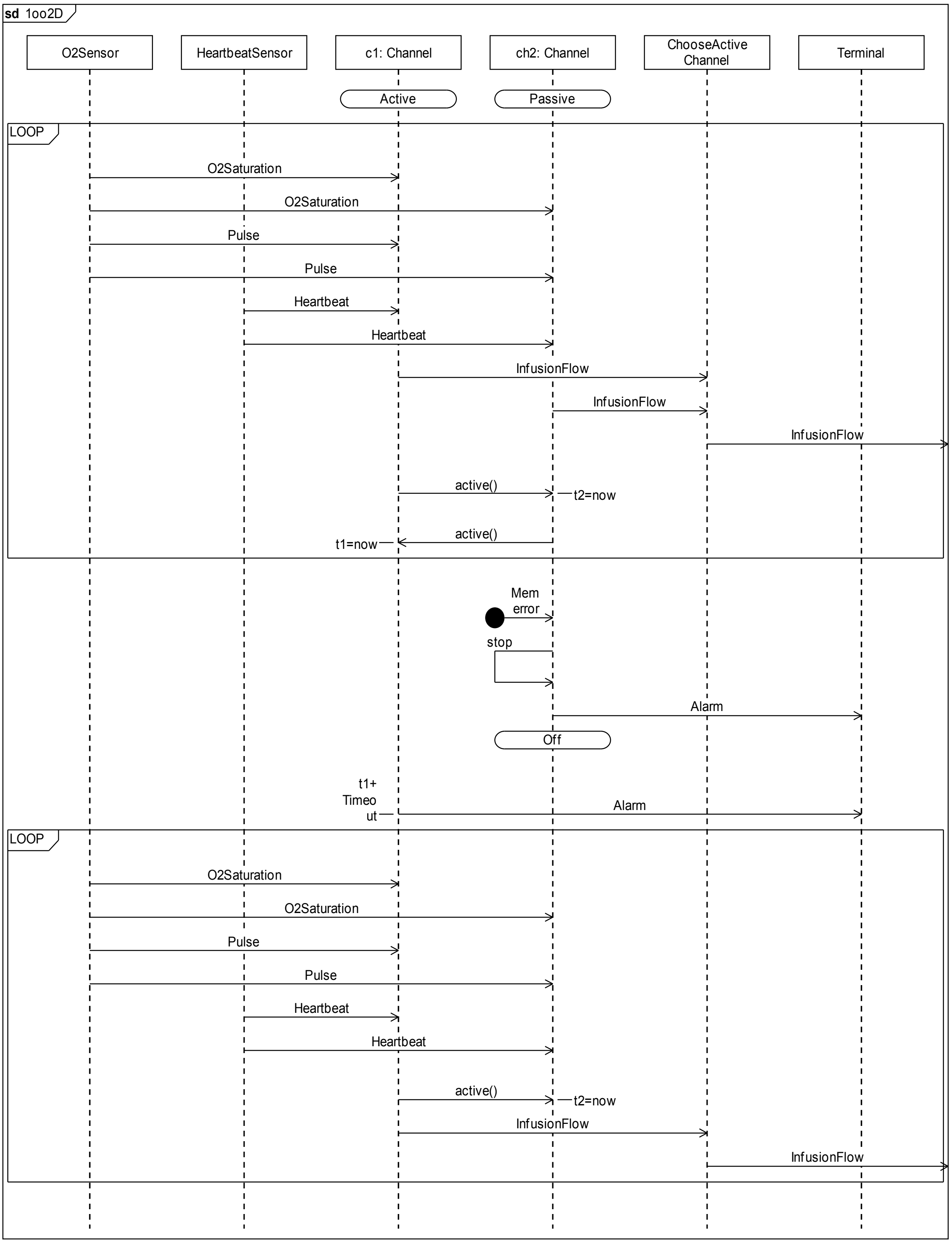


Figure 13.25.: *Redundancy Protocol 2*

13.11. Step D3 – Intra-Component Interaction

The components in this case study are quite simple. Therefore, they are not split into subcomponent and no interaction needs to be described.

13.12. Step D4 – Complete Component or Class Behavior

In this step, we check for each component if a state machine is necessary:

- To describe the behavior of the component ChooseActiveChannel, no state machine is necessary: By default the flow given via port p1 is forwarded to the output port of type P_IP. When choose1 was the last message received via one of the ports of type ∼P_ChooseChannel, the flow given via port p1 is forwarded to the output port of type P_IP. When choose2 was the last message received via one of the ports of type ∼P_ChooseChannel, the flow given via port p2 is forwarded to the output port of type P_IP.
- The component PCS_InitKey just sets the SharedKeyP when the environment authenticates itself. This component is stateless.
- The component SharedKeyP just represents a key (data type).
- The component PCS_KeyEx is a re-used component realizing the SSL protocol. Therefore, we do not specify the state machine.
- The component Terminal_Facade just forwards the messages depending on the type either to PCS_KeyEx or to PCS_MACEncr.
- The component PCS_MACEncr encrypts the following messages and creates Message Authentication Codes using the key of type KE_KeyP:
  - warnings from InInteg
  - warnings from OutInteg
  - warnings from PSInteg
  - warnings from PCS_WarnShow
  - vital signs from PCS_WarnShow
  - configurations to PCS_ConfigSettings

  In case of warnings from the integrity checking components, it may initiates a switch of the channel using the RedundancyHandler. The state machine describing this behavior is trivial and not necessary.
- The component PSInteg cyclically checks the integrity of the data of type PatientSettings using a Cyclic Redundancy Check (CRC, see (International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC), 2000)). In case of an error a warning is raised. The state machine describing this behavior is trivial and not necessary.
- The component PCS_ConfigSettings sets the data of type PatientSettings and updates its CRC. The state machine describing this behavior is trivial and not necessary.
- The component PCS_WarnShow forwards the vital signs and raises a warning if these vital signs exceed the configured limits. The state machine describing this behavior is trivial and not necessary.

- The component PCS_InInteg performs plausibility checks on the sensor data. In case of detected errors, a warning is raised. The state machine describing this behavior is trivial and not necessary.
- The component PCS_WarnShow forwards the vital signs and raises a waring if these vital signs exceed the configured limits. The state machine describing this behavior is trivial and not necessary.
- The component PCS_Control controls the infusion flow according to the configured rules and the current sensor data. The state machine describing this behavior is trivial and not necessary.
- The component PCS_OutInteg checks if the measured infusion flow corresponds to the desired flow. In case of detected errors, a warning is raised. The state machine describing this behavior is trivial and not necessary.
- The component RedundancyHandler continuously checks the hardware and may be informed about errors by other components. As long as no error is detected, it sends in regular intervals that this channel is still working correctly. If an error is detected, it tries to choose the other channel as the active one (controlling the infusion flow), tries to inform the physicians or nurses that service is necessary and shuts down the current channel. If the component RedundancyHandler receives no messages from the other channel that it is working correctly, it decides that this channel is the active one and informs the physicians or nurses that service is necessary. Since this is a re-used component, we give no state machine.

No new state predicates have been introduced.

13.13. Step I1 – Step T3

The patient care system can be implemented and tested as described in ADIT Steps I1 – Implementation and Unit Test, Step T1 – Component Tests, T2 – System Test, and T3 – Acceptance Test. The approaches for implementation, unit tests, component tests and system test are state-of-the-art and therefore not considered in this case study. The behavior of the patient care system is not based on the state of machine and environment, but on the internal data. Therefore, the approach presented in Chapter 11 is not applicable.

Chapter 14

Conclusions and Future Work

We give a summary of this thesis in Section 14.1 and present ideas for future work in Section 14.2.

14.1. Conclusions

In this thesis, we have presented several procedures to develop dependable systems. These procedures are integrated into ADIT, a process that covers Analysis, Design, Implementation and Testing. This process makes use of different kinds of patterns and ready-made components. In ADIT, we have provided a detailed description of the process steps with input, output, glossary, procedure and validation conditions (see Appendix A).

We have shown in Chapter 3 that such a structured heavyweight process (like ADIT) supports dependability even without extensions. The environment description helps to express dependability requirements, because dependability refers to domains in the environment. Patterns for requirements help to select solution approaches, re-use the corresponding behavioral descriptions and select appropriate dependability components. For dependable systems, often testing with a huge number of test cases is necessary. The environment models can also be used for test case generation.

Such a structured heavyweight process can only be handled with appropriate tool support, because even small changes in one output document lead to many other outputs to be updated. A tool should at least show all inconsistencies. If a single model is used for the different models that avoids redundant model elements, changes automatically propagate to other diagrams. We have extended a UML tool to support requirements engineering, the development of specifications, and the design of architectures. This has been achieved by defining stereotypes and validation conditions. A validation condition is, e.g., to check if a given sequence diagram is consistent with its problem diagram. These conditions can be checked automatically using our tool UML4PF described in Section 4.2. Users just have to press the "validator" button to check the validation conditions. If a condition does not hold, the user is provided with the natural-language description of the condition, and the wrong model element is pointed out to him or her. The possibility to identify inconsistencies leads to a reduced number of errors. A reduced number of errors is a necessary condition for dependable systems.

In Chapter 4, we have developed a UML profile to support the requirements engineering approach proposed by Jackson including problem diagrams, problem frames, and context diagram. With this profile, we enhanced the requirements engineering approach of Jackson by making use of UML concepts, such as multiplicities and aggregations. We also added the diagram type "technical context diagram" to prepare the architectural design in the analyses phase and the domain knowledge diagram to express domain knowledge. With our profile, a structured and seamlessly integrated development is possible. A structured and seamlessly integrated development process supports error reduction and therefore the dependability of the developed system.

Based on this UML profile for problem frames, in Chapter 5, we have developed dependability patterns that address confidentiality, integrity, availability, reliability and the security management.[1] These patterns are re-usable, because they refer to the environment description and are independent of solutions. The patterns are represented by a textual pattern with references to relevant domains, stereotypes that can be used to extend a problem diagram, and a corresponding predicate. For the dependability patterns expressed with UML classes and stereotypes, we developed integrity conditions, for example, that security requirements must explicitly address a potential attacker. The patterns and integrity conditions help the developers that they do not forget to describe important attributes of the requirements. Missing attributes are one possible source of errors in the development. This risk of such errors can be reduced by using our patterns and checking our integrity conditions.

Using these dependability patterns, in Chapter 6, we have developed a pattern system that can be used to identify missing requirements in a systematic way. The pattern system may also show possible conflicts between dependability requirements. Our pattern system is based on selecting generic mechanisms. Depending on the the selected mechanisms, other requirements that may conflict are given, additional domains are suggested, necessary conditions are given, and some related requirements are pointed out. Missing or conflicting requirements are another source of errors. This risk of such errors can be reduced by using our patterns and checking our integrity conditions.

When a set of requirements is selected, specifications (describing the behavior of the machine at the external interface) can be developed based on the requirements (see Chapter 7). The specifications can be expressed using sequence diagrams. We have defined constraints describing the consistency between problem diagrams and the specifications expressed with sequence diagrams. For security requirements, we have described a method to systematically derive the corresponding specifications. The method is formalized using model generation rules expressed in OCL. Our consistency check helps to avoid errors in the step to develop the specifications. This error avoidance supports the development of dependable systems.

When the behavior at the external interface is specified, the architecture of the machine can be designed. We have shown in Chapters 8 and 9 how software architectures can be derived in a systematic way from problem descriptions. First, we develop a simple initial architecture only based of the knowledge acquired in the requirements engineering phase. Using patterns, this architecture is refined to be implementable, and finally a layered architecture can be derived. We express the models with UML composite diagrams with stereotypes, and we developed a set of integrity conditions that allow one to identify inconsistencies in a diagram or between different diagrams. Since architectural design and requirements analysis are intertwined, design decisions drive the revision of the problem descriptions. Our method builds on established approaches to achieve dependability properties, such as encryption or redundancy. Its novelty is the fact that the different approaches are integrated and intertwined explicitly by an underlying methodology and a common notation.

To implement the architecture, we have presented in Chapter 10 how to implement components in Java, and we summarized rules for implementing secure and safe software. A well-structured software (as derived by this approach) supports the localization of functionality implemented to realized dependability requirements. The rules for safe and secure software realization help to avoid systematic faults.

Especially for dependability systems, testing is of great importance. In Chapter 11, we have proposed a new method for system validation by means of testing, which is based on environment models expressed as UML state machines. We extended a tool that can be used to generate and execute test cases and to execute UML state machines. In our method we propose to model requirements, facts, and assumptions explicitly by parallel state machines. We have defined patterns that help the developer to model requirements with state machines. Using

[1] Integrity, availability, reliability are addressed in the safety and security context.

the patterns, the environment model is structured in a way that parts can be easily re-used. Modeling the environment adds diversity to the development process and thus helps to avoid that the same mistake occurs for test development and SUT development.

Compliance to standards is an important property of many dependable systems. In Chapter 12, we have related the presented work to the Common Criteria for security aspects and to IEC/ISO 61508 for safety aspects. This helps to generate a documentation suitable for safety and security certification and use synergies.

We have illustrated the presented work with a Cooperative Adaptive Cruise Control case study and a Patient Care System Case study.

14.2. Future Work

The thesis covers the entire development process, and therefore it is not possible to address all important and interesting research topics in the field of dependability engineering. In the following sections, we outline the limitations of the methods presented in this thesis, sketch interesting research questions, and give first ideas for these questions.

14.2.1. Hazard, Threat, and Vulnerability Analyses

In this thesis, the risk assessment and hazard analyses necessary for systems with safety requirements and the threat and vulnerability analyses for systems with security requirements are not integrated. An agenda that outlines the integration has been presented in Section 3.1. Detailed validation conditions and procedures would help engineers in the task to perform these analysis steps. With such procedures and validation conditions, missing hazards or threats can be identified, and missing requirements that address the threat or hazard mitigation can be detected.

14.2.2. Problem Frame Approach

The problem frame approach is currently scarcely applied in industry, because

- many diagrams have to be created for a real project,
- no tools suitable for industrial applications exist, and
- the notation is not known by the majority of engineers.

The translation of all diagrams of the problem frames approach into the well-known UML notation is a first step to make the approach more applicable in industry. A UML-based tool that can generate the diagrams with minimal user interaction could help to make the approach more applicable.

14.2.3. Development of Further Requirements Patterns

Our patterns for dependability requirements are not complete. Patterns for maintainability requirements are missing, and we did not specify any patterns for privacy requirements and accountability requirements. With our set of patterns it is possible to describe the dependability requirements of many systems. We are currently working on patterns for privacy requirements.

14.2.4. Pattern System for Dependability Requirements

Our pattern system for dependability requirements is not complete. Especially, only a small set of mechanisms is suggested for each requirement. Nevertheless, a huge set of real-live problems can be expressed and analyzed using this pattern system. By an investigation of applied solutions, possible interactions, necessary requirements, domains to be considered, and related requirements can be determined. To investigate applied solutions, existing specifications (e.g., public Security Targets) and standards (e.g., ISO/IEC 61508) can be used as an input. Another interesting research question is the prioritization of dependability requirements. It is desirable to build a tool to support interactive identification of missing and interacting requirements.

14.2.5. Specification of Dependable Systems

The method for the development of specifications presented in this thesis is currently limited to functional requirements and security requirements expressed by sequence diagrams. Although sequence diagrams are well-accepted in industry, other notations such as UML communication diagrams can be generated in a similar way with different patterns. An interesting research question is to extend the method presented in Chapter 7 and to develop patterns for other dependability requirements that can be transformed into functional specifications.

14.2.6. Architecture of Dependable Systems

The architecture obtained by our method is not an optimized architecture, but a working one. An interesting research question for further investigation is to extend our approach to support the development of design alternatives and to support software evolution. Our approach is limited on structural descriptions of software architectures. It is also desirable to give rules for selecting appropriate design alternatives, evaluate the architectures for different dependability requirements, optimize the architectures, and describe additional patterns for architectural design.

14.2.7. Interface Behavior Description of Components

In this thesis no guidance and no patterns are provided for the interface description of components. According to the selected generic mechanism, a set of patterns for behavioral description between components can be provided. For example. the redundancy protocol in the PCS case study could be expressed as a pattern. It is interesting to investigate how to extend our method to also support automatically checking the coherence of behavioral descriptions with the structural descriptions.

14.2.8. Implementation of Dependable Systems

The implementation method for software components is presented in this thesis is limited to Java, but it can be extended easily to support other object-oriented languages. Only a subset of the rules for the implementation of safe and secure software is given in this thesis. It is interesting to elaborate a more comprehensive set of rules from different standards addressing all dependability requirement types.

14.2.9. Testing

Currently our method is limited to reactive systems that can be modeled with state machines. Systems that are based on complex data types cannot be tested. It is an interesting task to extend the tool and the method to be able to handle complex data types.

The method can be adapted to use model checking instead of testing. For example, the model and the requirements can be expressed with PROMELA (Process or Protocol Meta Language). They could be analyzed, e.g., with the SPIN model checker.

14.2.10. Standards for Dependable Systems

In this thesis, we only provide a mapping between the aspects covered in this thesis and the ISO/IEC 61508 and the Common Criteria. In the future, we plan to achieve a tighter integration of ADIT and these standards: We plan to create guidelines describing the generation of documentation for certification according to these standards, and possibly also a tool that exports this documentation from the model. We will consider additional standards, e.g. the ISO 26262, which is a specific adaption of the standard ISO/IEC 61508 for automotive systems, or the ISO 27001, which is a standard for security management systems in companies.

14.2.11. Validiation of the Process

The complete process has been evaluated by its application on two case studies, the Cooperative Cruise Control and the Patient Care System. Previous versions of the process and single aspects have been evaluated on different case studies presented in the cited papers. An interesting question is how the process scales to industrial applications. For an industrial application, the tool support needs to be improved and the process has to be integrated into the established processes in the given company.

Appendix A

ADIT steps

A.1. Step A1: Problem Elicitation and Description

A.1.1. Input, Output, Glossary, Validation

Input:	informal description of the task	natural language
Output:	requirements R	optative statements
	domain knowledge $D \equiv F \wedge A$	indicative statements
	context diagram of system to be built	extended UML notation
Glossary:	definitions, designations	natural language / formulas
	domains	natural language
	phenomena	natural language
Validation:	The domains and phenomena of the context diagram must be consistent with R and D.	check manually
	Phenomena controlled by a biddable domain must have counterpart phenomena located between machine and causal, lexical, or designed domains.	check manually
	Biddable domains cannot be directly connected to lexical domains.	check automatically
	A context diagram has at least one machine domain.	check automatically
	The machine domain must control at least one interface.	check automatically
	There must be exactly one context diagram.	check automatically
	Connection domains must have at least one observed and one controlled interface.	check automatically
	The elements allowed in a package with the stereotype ≪ContextDiagram≫ are: • classes with stereotype ≪Domain≫ or any of its sub-types • associations with stereotype ≪connection≫ or any of its sub-types • dependencies with stereotypes ≪controls≫, ≪observes≫, or ≪isPart≫	check automatically
	The stereotypes ≪CausalDomain≫, ≪DesignedDomain≫, ≪LexicalDomain≫, ≪DisplayDomain≫, or ≪Machine≫ are not allowed together with ≪BiddableDomain≫.	check automatically

A.1.2. Procedure

The procedure for this step is as follows:

1. Derive requirements from the given informal description.
2. Collect the necessary domain knowledge.
3. Describe the environment by setting up a context diagram using domains and phenomena. To identify domains and phenomena in the derived requirements, we can use the following heuristics:

   Nouns are good candidates for domains.

   Verbs are good candidates for phenomena.

All items of the output can be developed in parallel. When e.g., a new requirement is identified, it is possible that an additional phenomenon must be added to the context diagram or the domain knowledge must be described in more detail.

A.2. Step A2: Problem Decomposition

A.2.1. Input, Output, Glossary, Validation

Input:	all results of Step Problem Elicitation and Description	
Output:	set of subproblems	extended UML notation
Glossary:	machine domains of subproblems	natural language
	new phenomena and domains (if introduced)	natural language
	name of composed requirements	natural language
Validation:	A problem diagram has exactly one machine domain.	check automatically
	Requirements do not constrain a machine domain.	check automatically
	Requirements do not constrain biddable domains.	check automatically
	A problem diagram contains at least one requirement.	check automatically
	All subproblems can be derived from the context diagrams by means of decomposition operators.	check automatically
	Connection domains must have at least one observed and one controlled interface.	check automatically
	The machine domain must control at least one interface.	check automatically
	Dependency ≪contains≫ is only allowed between interfaces.	check automatically
	Dependency ≪restricts≫, ≪complements≫, or ≪similar≫ is only allowed between statements or any of their sub-types.	check automatically
	Dependency ≪constrains≫ and ≪refersTo≫ point from statements to domains.	check automatically
	Elements allowed in a package with the stereotype ≪ProblemDiagram≫ or ≪ProblemFrame≫ are: • classes with stereotype ≪Domain≫ or any of its sub-types • classes with stereotype ≪Statement≫ or any of its sub-types • associations with stereotype ≪connection≫ or any of its sub-types • dependencies with stereotypes ≪refersTo≫, ≪constrains≫, ≪controls≫, ≪observes≫, ≪isPart≫ or ≪complements≫	check automatically
	Dependency ≪concretizes≫ or ≪refines≫ is only allowed between (a) interfaces, (b) classes with stereotype ≪Domain≫ or any of its sub-types and classes with stereotype ≪ConnectionDomains≫ or any of its sub-types (c) classes with stereotype ≪ConnectionDomains≫ or any of its sub-types and associations with stereotype ≪connection≫ or any of its sub-types or interfaces	check automatically
	All requirements R are covered in some subproblem, i.e., a class with stereotype ≪Statement≫ has at least one ≪constrains≫ dependency.	check automatically
	The problem diagrams must be consistent with the context diagram • The submachines of the problem diagrams must be part of the machine(s) in the context diagram.	check automatically

A.2.2. Procedure

In a problem diagram, the knowledge for a sub-problem described by a set of requirements is represented. A problem diagram can be systematically derived from the context diagram by means of decomposition operators. To obtain these subproblems, proceed as follows:

1. Identify subsets of related requirements that could be solved by one subproblem.
2. Make use of decomposition operators (see Section 4.1.4) for deriving the subproblems as projections of the overall problem stated in the context diagram.

3. Set up a problem diagram for each subproblem.

A.3. Step A3: Abstract Machine Specification

A.3.1. Input, Output, Glossary, Validation

Input:	All results of Step Problem Decomposition	
	All results of Step Problem Elicitation and Description	
Output:	set of abstract submachine specifications $S_{abstract}$ represented by sequence diagrams	UML notation
Glossary:	state predicates	natural language/ formulas
validation:	$S_{abstract} \wedge D$ are non-contradictory, $S_{abstract} \wedge D \Rightarrow R$	check manually
	messages and phenomena are consistent	check automatically
	there exists at least one scenario for each subproblem	check automatically
	for each subproblem scenarios exist that consider normal and exceptional cases	check manually

A.3.2. Procedure

1. *Derive a specification for the machine*: For each subproblem, we check if the corresponding requirements are implementable and identify necessary domain knowledge. In Section 7.1, we stated three conditions when a requirement is not implementable and give guidance for deriving the specification.

2. For each subproblem, *draw sequence diagrams* that capture normal as well as exceptional cases to express the specification.

   a) For each domain which is directly connected to the machine in a problem diagram, a lifeline is drawn. Biddable domains are represented as actors.

   b) The phenomena from the environment to the machine are represented by asynchronous signals between lifelines.

   c) Phenomena to lexical domains are represented by synchronous signals.

   d) Add user feedback, where appropriate.

   e) Introduce state predicates serving as pre- and postcondition on the lifeline of the machine, where applicable.

   f) Introduce getter and setter messages with corresponding return messages.

   A problem diagram does not contain getter or setter phenomena. However, we need them in a sequence diagram. Therefore, we add them where appropriate. We propose the following naming convention for getters: 'get_' followed by the message name. The same applies to setters.

   g) Add timing constraints and lost/found messages, where appropriate.

A.4. Step A4: Technical Context Diagram

A.4.1. Input, Output, Glossary, Validation

Input:	Results of Step Problem Elicitation and Description	
	All results of Step Problem Decomposition	
Output:	Technical context diagram	extended UML notation
Glossary:	New phenomena and domains	natural language
	Technical description of interfaces	natural language
Validation:	New phenomena and domains are suitable to implement $S_{abstract}$	check manually
	The technical context diagram must be consistent to the problem diagrams	check automatically
	Each machine in the technical context diagram must be a machine in the context diagram	check automatically
	Elements allowed in a package with the stereotype ≪TechnicalContextDiagram≫ are: • classes with stereotype ≪Domain≫ or any of its sub-types • associations with stereotype ≪connection≫ or any of its sub-types • dependencies with stereotypes ≪controls≫, ≪observes≫, or ≪isPart≫	check automatically
	There is at least one machine domain in a technical context diagram	check automatically

A.4.2. Procedure

1. Collect the results from the subproblem decomposition (new domains and phenomena).

2. Introduce necessary connection domains.

3. Identify the technical interfaces that must be considered. Technical phenomena (such as TCP/IP packets, SMTP commands, etc.) should be used.

4. Describe the technical phenomena in detail or refer to an existing technical specification.

Rules for the technical context diagrams:

- The operators “introduce connection domain” and “introduce shared phenomena” (for that connection domain) are allowed.

- The operators “Refine phenomena” and “Combine (i.e., abstract) phenomena” are allowed.

- Domains with no direct connection to the machine are left out.

A.5. Step A5: Operations and Data Specification

A.5.1. Input, Output, Glossary, Validation

Input:	all results of Step Abstract Machine Specification	
Output:	class model for machine state	UML notation
	operation specification for each abstract submachine specification	operations with OCL
Glossary:	class names	natural language
	attribute names	natural language
	auxiliary function names	natural language
	association names and role names	natural language
	relation between state predicates in Step Abstract Machine Specification and pre-/postconditions	natural language/ OCL
Validation:	Exactly the operations occurring in the abstract specification in Step Abstract Machine Specification must be described	check automatically
	Operation specifications must be consistent with abstract specifications in Step Abstract Machine Specification	check manually
	For each described operation, a pre- and postcondition must exist	check automatically
	The pre- and postconditions expressed in OCL must be syntactically correct	check automatically
	The postcondition covers all cases exhibited in the abstract specification of Step Abstract Machine Specification	check manually
	All parameters of operations must be known by the caller and all parameters of sent messages must be known by the machine	check manually
	Parameters must be used in the pre- and/or postcondition	check manually
	Operation specification must be consistent with class model of the machine state	check automatically
	All classes, associations, and attributes newly introduced in the class model must be motivated by some operation specification	check manually

A.5.2. Procedure

1. Collect all operations contained in the abstract specification (arrow from the environment to the machine) and associated output events (arrows from the machine to the environment). Lexical domains are considered to be part of the machine.

2. For each operation, create exactly one operation schema.

   a) Enter the name of the operation in the section **Name**. The name must be equal to the name introduced in the abstract specification.

   b) Enter a short description of the operation in the section **Description** based on the textual task description.

   c) Draw a class diagram serving as class model:

      - For each lifeline except actors, a class is drawn.
      - Operations contained in the abstract specification (arrow from the environment to the machine) are operations of the submachine. Decide what user input is necessary to achieve the informally described effect of the operation and add parameters accordingly.
      - Output events (arrows from the machine to the environment) are operations of display domains. The names of the output events must be equal to the names introduced in the abstract specification. Possibly parameters must be added. Biddable domains do not occur in the class diagram.

   d) Specify a **Precondition**. The precondition of the corresponding sequence diagram must be taken into account.

   e) Specify a **Postcondition**.

   f) Update the class diagram:

- Collect newly introduced classes, attributes, associations, etc. and add necessary *dataTypes*, *enumerations* and auxiliary functions. Functions that correspond to API functions are added to a special class *APIProvided*.

g) Enter additional information about the operation, i.e., about auxiliary functions, conceptual remarks, etc., in the section **Remarks**.

3. Collect all used state predicates in the class models and define them.

A.6. Step A6: Machine Life-Cycle

A.6.1. Input, Output, Glossary, Validation

Input:	All results of Steps Abstract Machine Specification and Operations and Data Specification	
Output:	Relationship of abstract specifications	life-cycle expressions
Glossary:	new life-cycle expression names	natural language
Validation:	Each sequence diagram of Step Abstract Machine Specification is contained in at least one life-cycle expression	check manually
	For each biddable domain exactly one life-cycle exists	check manually
	The life-cycles must be consistent with the state predicates in Step Abstract Machine Specification	check manually
	The life-cycles must be consistent with the pre- and postconditions in Step Operations and Data Specification	check manually
	Exactly one life-cycle exists for the machine domain, that combines all life-cycles	check manually

A.6.2. Procedure

1. Set up exactly one life-cycle for each biddable domain:

   a) Collect all names of sequence diagrams, where the respective biddable domain invokes an operation.

   b) Combine life-cycle expressions for a biddable domain by means of life-cycle operators. Note that the pre-clause of the current operation must be implied by the post-clause of the previous operation.

   c) Check consistency between state predicates and each life-cycle.

2. Set up exactly one life-cycle for the machine domain:

   a) Collect all names of sequence diagrams, that specify internal operations.

   b) Combine life-cycles of biddable domains and internal operations by means of life-cycle operators.

   c) Check consistency between state predicates and machine life-cycle.

A.7. Step D1a: Initial Architecture

A.7.1. Input, Output, Glossary, Validation

Input:	Results of Step Problem Decomposition	
	Results of Step Technical Context Diagram	
Output:	Initial software architecture	UML composite structure diagram
	Specification of initial component interfaces	UML interface classes or references to APIs
Glossary:	component names	natural language
Validation:	For each provided or required interface of machine ports in the architecture, there exists a corresponding interface in the technical context diagram.	check manually
	There is one component for each submachine related to the given machine of the technical context diagram as well as for all relevant lexical domains found in the problem diagrams.	check manually
	The internal components have the stereotype ≪Component≫ or ≪ReusedComponent≫.	check automatically
	The internal components are connected to each other and to the external ports according to the connections in the technical context diagram /subproblems.	check manually
	The stereotypes of the connectors are consistent to the associations of the technical context diagram.	check manually

A.7.2. Procedure

Repeat this procedure for each machine in the technical context diagram:

- Select a machine in the technical context diagram. We create an initial architecture diagram for this machine.
  - Collect all domains that have a direct connection to this machine and are not part of the machine domain. For each of these domains we define a corresponding *external port*.
  - Define required and provided interfaces for the external ports according to the observed and controlled interfaces of the technical context diagram.
  - Collect all sub-machines as well as *relevant* lexical domains related to the machine in the technical context diagram. A lexical domain is considered to be relevant if it has a composition/aggregation relation to a machine domain.

    The sub-machines and lexical domains become *internal components*.
  - Add ports to the internal components.

    This is done according to the associations found in the problem diagrams. We create one port per association. These ports are considered as *component ports*.
  - Define required and provided interfaces for the component ports according to the observed and controlled interfaces of the corresponding problem diagram(s).

    We have identified the following cases for defining such required and provided interfaces:

    1. *An observed interface in a problem diagram corresponds to a provided interface of the component.*

       This case applies if the controlling domain is not lexical. The domain type of the observing component does not matter, i.e., it may or may not be lexical.

       Define an `interface implementation` from the component port to the interface.

2. *A controlled interface in a problem diagram corresponds to a required interface of the component.*

   This applies if the controlling component is not lexical. The domain type of the observing component does not matter, i.e., it may or may not be lexical.

   Define a ≪use≫-dependency from the component port to the interface.

3. *A controlled interface in a problem diagram corresponds to a required interface with operations with return values of the component.*

   This case applies if the component is lexical and the observing domain is not lexical.

   Create a new “getter”-interface

   To create the name of this interface, we use 'get_' followed by an operation name of the interface controlled by the lexical component. Add necessary return values, where applicable. This is repeated for each operation. The overall name of the “getter”-interface starts with '{' and ends with '}'.

   or

   add the elements to an already existing “getter”-interface for this component.

   Define a ≪concretizes≫-dependency from the “getter”-interface to the interface controlled by the lexical component.

   Define an `interface implementation` relation from the component port to the “getter”-interface.

4. *An observed interface in a problem diagram corresponds to a required interface with operations with return values of a component.*

   This case applies, if the component is not lexical and the controlling domain is lexical.

   Since lexical domains cannot call operations of other components, it is necessary to create a “getter”-interface or add the elements to an already existing “getter”-interface for this component.

   The name of this interface is created by using 'get_' followed by an operation name of the interface controlled by the lexical component. Add necessary return values, where applicable. This is repeated for each operation. The “getter”-interface name starts with '{' and ends with '}'.

   Define a ≪concretizes≫-dependency from the “getter”-interface to the interface controlled by the lexical component.

   Define a ≪use≫-dependency from the component port to the “getter”-interface.

- Connect the component ports to the component ports according to the connections found in the problem diagram.
- Connect component ports to component ports according to the connections found in the technical context diagram.

The “setter”-interfaces correspond to phenomena names. Therefore, it is not necessary to treat them in a special way.

A.8. Step D1b: Implementable Architecture

A.8.1. Input, Output, Glossary, Validation

Input:	Result of Step Initial Architecture	
	Results of Step Machine Life-Cycle	
Output:	Implementable software architecture	UML composite structure diagram
	Specification of components and their interfaces	UML interface classes or references to APIs
Glossary:	component names	natural language
Validation:	The facade has port types that correspond to the external ports of the initial architecture.	check manually
	The facade has ports corresponding to the component ports. The port types are the inverse of the component ports.	check manually
	A ≪concretizes≫ dependency exists from the implementable architecture to the initial architecture.	check manually
	The stereotypes of the connectors to the facade are the same as in the initial architecture.	check manually

A.8.2. Procedure

For each initial architecture do the following:

- If several internal components are connected to one or more external interface(s) in the initial architecture, a *facade* component (see the corresponding design pattern by Gamma et al. (1995b)) may be added. That component has one provided interface containing all operations of the corresponding component port and several used interfaces provided by the submachine components.
- If a facade has to ensure that the operations are called in a certain order, the facade component will contain an internal coordinator component. Such a coordinator is usually required if the component is connected to a biddable domain to assure that certain things are done in a specific order. A coordinator may also be necessary if certain interaction restrictions must be enforced. The software life-cycle for the respective biddable domain states how the coordinator has to be built.
- Add connectors between internal components and facade as well as between facade and external port.
- Apply patterns solving known problems. For example, for problems that automatically trigger actions, add a component responsible for handling timing.

A.9. Step D1c: Re-structure Software Architecture

A.9.1. Input, Output, Glossary, Validation

Input:	Results of Step Implementable Architecture	
	Results of Step Technical Context Diagram	
Output:	Software architecture of a specific architectural style	UML composite structure diagram
Glossary:	component names, e.g., of adapters, user interface	natural language
Validation:	All components of the implementable architecture must be contained in the layered architecture.	check manually
	The connectors connected to the ports in the layered architecture must have the same stereotypes or more specific ones than in the implementable architecture.	check manually

A.9.2. Procedure

In our layered architecture, we distinguish three different layers. The first and at the same time the highest layer is the *application layer*. One or more components of this layer implement the core functionality of the software. Their interfaces usually reflect the phenomena found in the context diagram. The application components are clearly separated from components responsible for handling auxiliary functionalities, such as translating signals from (hardware) drivers to signals of the application components and vice-versa. These components are called *adapters* and form the second, middle layer. The last and lowest layer consists of the *hardware abstraction layer* (HAL) and components realizing user interfaces. The signals processed at the interfaces are used to communicate with the environment and usually correspond to the phenomena found in the technical context diagram.

There is no universally valid approach on how to assign the different components to the different layers. Many design decisions have to be made by the developer. Therefore, many decisions rely on the personal experience of the developer. In the following, we illustrate our approach to tackle the arising design decisions.

To obtain a layered architecture we have to assign all components from the implementable architecture to one of the layers:

- Assign submachine components to the application layer.

  Add one port for each occurring port type of the submachine components.

- Facades related to causal domains: split facade into three parts

  1. A facade, which can be assigned to the application layer.

     Note that coordinator components for physical connections usually belong the the application layer.

     A facade in the application layer serves to keep the interfaces "narrow".

  2. An adapter, which can be assigned to the middle layer.

  3. A (re-used) component, which can be usually assigned to the lowest layer.

- Facades related to biddable domains: split into two parts

  1. A facade, which can be assigned to the application layer.

  2. A user interface, which can be assigned to the middle/lowest layer.

     Note that coordinator components for biddable domains should be part of the corresponding user interface component.

  However, before actually assigning a component part to the application or middle layer, we ask ourselves "Is this component really necessary?" For example, "Is this adapter really necessary?" The answer in this case is no should the adapter only implement a function call or pass on data without processing it further (i.e., it is considered as *trivial*).

- Connection stereotypes of the technical context diagram (referring to external interfaces) and the context diagram (referring to interfaces of the application layer) help in identifying new components, e.g., user interfaces or network drivers and corresponding adapters.

A.10. Step D2: Inter-Component Interaction

A.10.1. Input, Output, Glossary, Validation

Input:	results of Step Machine Architecture	
	results of Step Operations and Data Specification	
Output:	interface behavior of all components for all operations	UML sequence diagrams
Glossary:	new messages	
	new state predicates, if introduced	
Validation:	the sequence diagrams must be consistent with the behavior described in Step Abstract Machine Specification and in Step Machine Life-Cycle	check manually
	the sequence diagrams must realize the operations described in Step Operations and Data Specification	check manually
	all messages in the application interface classes of Step Machine Architecture must be used in some sequence diagram	check manually
	direction of messages must be consistent with the required and provided interfaces of Step Machine Architecture	check manually
	messages must connect components as connected in the software architecture of Step Machine Architecture	check manually
	it must be possible to relate any new state predicates to the state predicates of Step Abstract Machine Specification	check manually

A.10.2. Procedure

For each operation we identified in Step Operations and Data Specification:

1. Select the software components that realize the operation.
2. Collect the interface classes of the corresponding software components.
3. Create a sequence diagram that describes the communication flow between the software components such that the postcondition of the operation can be fulfilled.
4. Universally quantified expressions and set comprehensions contained in the postcondition of the operation are translated into loop combined fragments.
5. Re-use names of variables and parameters contained in the corresponding operation schema. If necessary, invent missing variables, parameters, and state predicates.
6. Use APIs of re-used software components and given domains (e.g., database management systems, web server).
7. Use synchronous messages only (exception: messages from or to the environment without feedback).

A.11. Step D3: Intra-Component Interaction

A.11.1. Input, Output, Glossary, Validation

Input:	all results of Step Inter-Component Interaction	
	class and operation model of Step Operations and Data Specification	
Output:	architectural description including interface descriptions of complex components	UML class diagram or composite structure diagram
	intra-component interaction	UML sequence diagrams
	description of internal functions	as appropriate
glossary:	new messages	
	new state predicates if introduced	
Validation:	sequence diagrams of one component must be consistent with the same interface behavior in Step Inter-Component Interaction	check manually
	it must be possible to relate any new state (predicates) to the state predicates of Step Inter-Component Interaction	check manually

A.11.2. Procedure

1. Set up a preliminary structure of complex components
   - For lexical domains take the class diagram of Step Operations and Data Specification into account.
   - If applicable use design patterns
2. For each sequence diagram of Step Inter-Component Interaction select the operations and the corresponding output messages of the component to be described and create a sequence diagram.
3. Set up a behavioral design for each operation; further messages and auxiliary functions can be invented.
4. Add newly introduced messages and auxiliary functions to the architectural description.

A.12. Step D4: Complete Component or Class Behavior

A.12.1. Input, Output, Glossary, Validation

Input:	results of Step Machine Architecture	
	all results of Steps Inter-Component Interaction and Intra-Component Interaction	
	results of Step Machine Life-Cycle	
Output:	complete internal behavior of the class or component (if it has more than two different states)	UML state machines
Glossary:	new state predicates if introduced	
Validation:	the state machines must describe the same behavior as in Step Inter-Component Interaction or Step Intra-Component Interaction	check manually
	the state machines must be consistent with the life-cycle model of Step Machine Life-Cycle	check manually

A.12.2. Procedure

For each component of the software architecture of Step Re-structure Software Architecture check whether a state machine is necessary.

- For re-used components no state machine must be created.
- Check the corresponding sequence diagrams together with the life-cycle expression if more than two states are to be expected.
- Check whether a refinement of the component exists in Step Inter-Component Interaction. If a refinement exists, the subsequent steps must be performed on the results of Step Intra-Component Interaction.

For each component of Step Re-structure Software Architecture where a state machine is necessary, perform the following steps according to the sequence diagrams:

- Messages to the component or class are triggers for a transition.
- Messages from the component or class are output signals or actions.
- Explicitly mentioned state predicates are transformed into states.
- Introduce new states where necessary.
- The states reached after a transition must be consistent with the life-cycle model.

A.13. Step I1: Implementation and Unit Test

A.13.1. Input, Output, Glossary, Validation

Input:	interface behavior of Steps Inter-Component Interaction and Intra-Component Interaction	
	interfaces and components of Step Machine Architecture	
	complete interface behavior of Step Complete Component or Class Behavior	
	operation model of Step Operations and Data Specification	
Output:	unit tests with expected results	source code
	implementation	source code
	test drivers and test stubs	source code
Glossary:	names and purpose test drivers/test stubs	comments in source code
	names and purpose test cases	comments in source code
Validation:	tests pass	check automatically

A.14. Step T1: Component Tests

A.14.1. Input, Output, Glossary, Validation

Input:	life-cycle of Step Machine Life-Cycle	
	interface behavior of Steps Inter-Component Interaction and Intra-Component Interaction	
	implementation of Step Implementation and Unit Test	
Output:	component tests with expected results	source code
	test drivers and test stubs	source code
Glossary:	names and purpose test drivers/test stubs	comments in source code
	names and purpose test cases	comments in source code
Validation:	tests pass	check automatically

A.14.2. Procedure

- Each sequence diagram of Step Inter-Component Interaction must be covered by test cases.
- To test internal operations, Step Intra-Component Interaction can be consulted.
- Use concrete values according to equivalence classes.
- JUnit can be used for testing components.
- The following procedure should be performed for all components:
  1. Instantiate all classes that implement the component.
  2. Create stubs for all components in the environment.
  3. Connect the component to stubs (maybe additional code for testing must be inserted).
  4. For each sequence diagram:
     - For each message the component sends to its environment, define the expectations. The method `verify(...)` can be used to specify how often a certain stub expects the messages.
     - If messages have return values, the expected return values must be defined.
     - For each message to the component, the corresponding operation must be called with concrete parameters.

- If the component returns a value, it must be checked with **assertEquals(...)**.

Alternative for components accessing a database

For each test case:

1. Set database to a defined state (using SQL-statements).
2. For each message to the component and expected database state:
   - Call operation with concrete parameters.
   - Check database state (using SQL-statements).
3. Reset database state (using SQL-statements).

A.15. Step T2: System Test

A.15.1. Input, Output, Glossary, Validation

Input:	life-cycle of Step Machine Life-Cycle	
	interface behavior of Step Abstract Machine Specification	
	implementation of Step Implementation and Unit Test	
Output:	"system tests" (machine tests, black-box) with expected results	source code
	test drivers and test stubs	source code
Glossary:	names and purpose test drivers/test stubs	comments in source code
	names and purpose test cases	comments in source code
Validation:	tests pass	check automatically

A.15.2. Procedure

- Each sequence diagram of Step Abstract Machine Specification must be covered by test cases.
- Use concrete values according to equivalence classes.
- JWebUnit can be used for testing web applications.
- The following procedure should be performed:
  1. For each message to the machine, the corresponding request (e.g. HTTP-POST or HTTP-GET) must be sent with concrete parameters.
  2. For each message from the machine to the environment, define the expectations.

A.16. Step T3: Acceptance Test

A.16.1. Input, Output, Glossary, Validation

Input:	life-cycle of Step Machine Life-Cycle	
	subproblems of Step Problem Decomposition with corresponding requirements of Step Problem Elicitation and Description	
	implementation of Step Implementation and Unit Test	
Output:	test description and test data for black-box tests	natural language
Glossary:	test cases	
Validation:	tests pass	check manually

A.16.2. Procedure

Remarks:

- Use checklists with expected results to describe test cases.
- Test data should come from the application domain.

1. Perform installation test.
2. Perform tests in intended environment.
3. Perform usability and performance tests.

Appendix B

CACC case study - additional figures

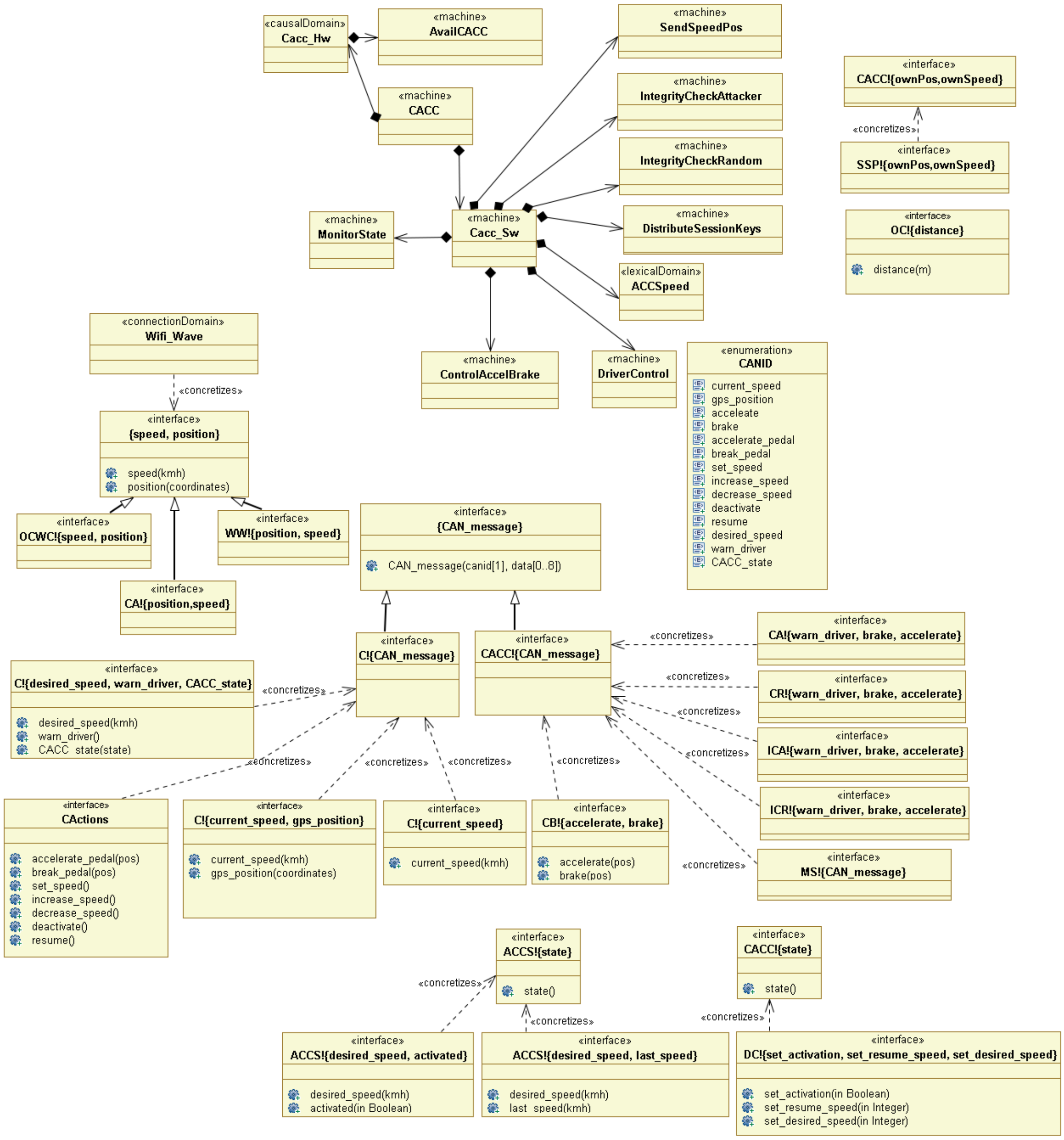


Figure B.1.: *Mapping of CACC Problem Diagram and Context Diagram*

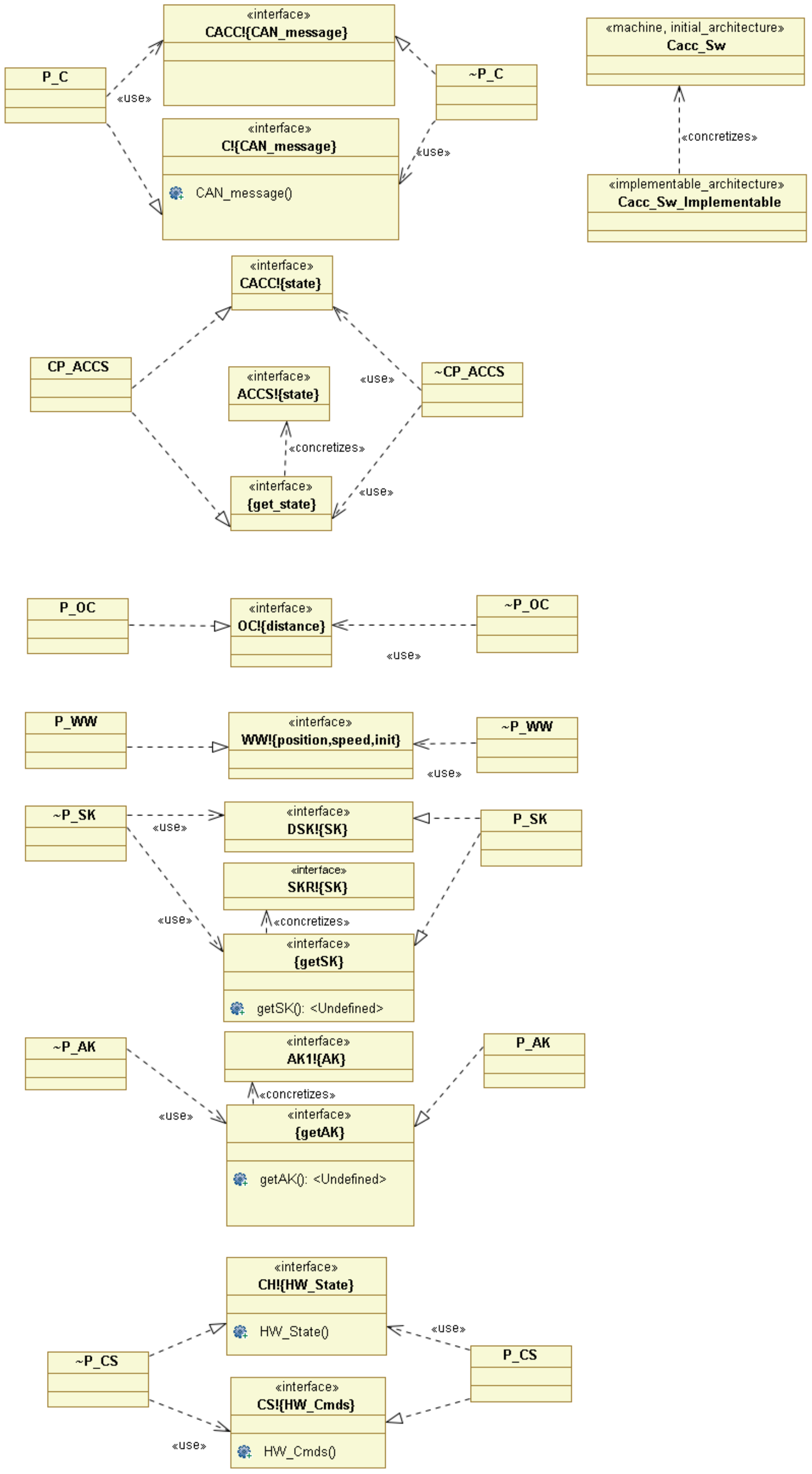


Figure B.2.: *Port Type Definition of External Ports*

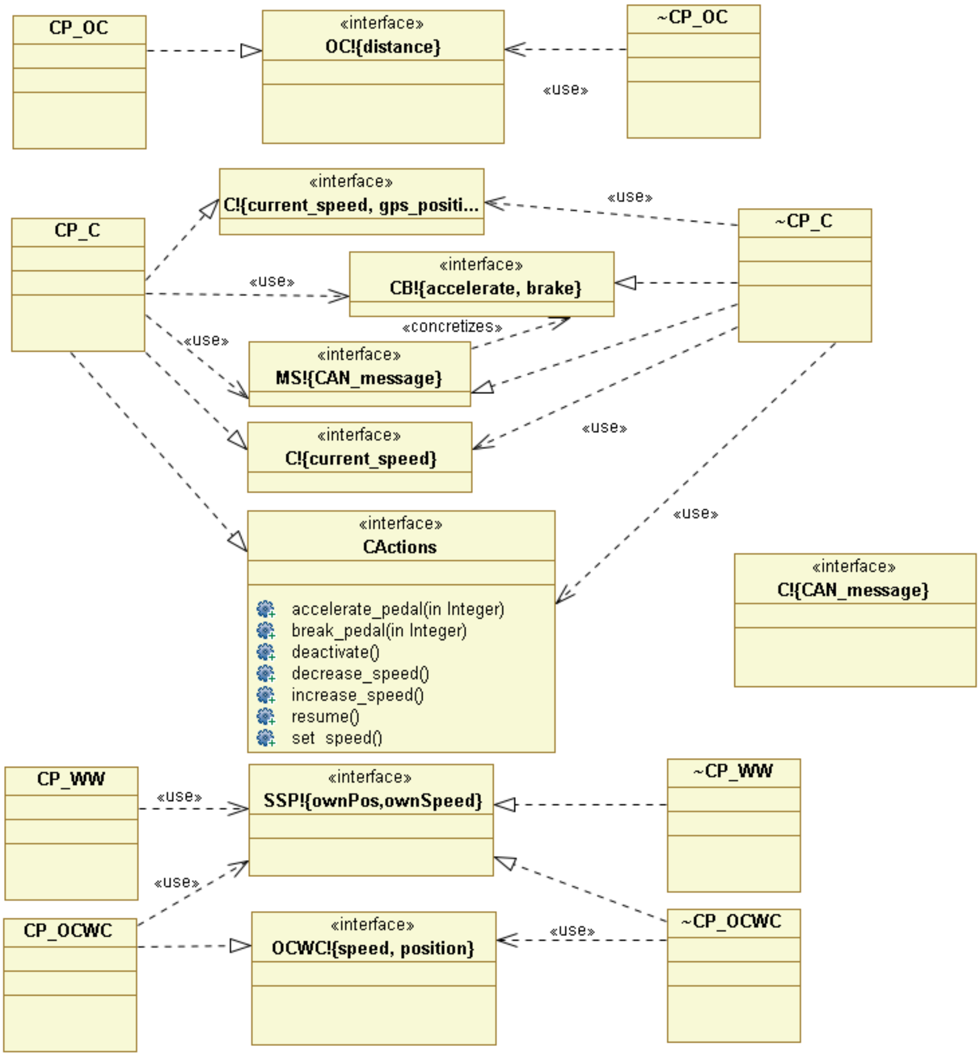


Figure B.3.: *Port Type Definition of Internal Ports*

Appendix C

OCL expressions

C.1. General constraints

```
1 Class.allInstances() ->select(
2           getAppliedStereotypes().name ->includes('Domain') or
3           getAppliedStereotypes().general.name ->includes('Domain') or
4           getAppliedStereotypes().general.general.name ->includes('Domain') or
5           getAppliedStereotypes().general.general.general.name ->includes('Domain'))
6 ->isUnique(name)
```

Listing C.1: *Name of domains must be unique*

We select the classes with the stereotype ≪Domain≫ or any of its sub-types (lines 1-5) and check if the names are unique using the built-in operation *isUnique* (line 6).

```
Interface.allInstances() ->isUnique(name)
```

Listing C.2: *Name of interfaces must be unique*

We check that all names of interfaces are unique using the built-in operation *isUnique*.

```
1 Class.allInstances() ->select(
2           getAppliedStereotypes().name ->includes('Statement') or
3           getAppliedStereotypes().general.name ->includes('Statement') or
4           getAppliedStereotypes().general.general.name ->includes('Statement') or
5           getAppliedStereotypes().general.general.general.name
                ->includes('Statement'))
6 ->isUnique(name)
```

Listing C.3: *Name of statements must be unique*

We select the classes with the stereotype ≪statements≫ or any of its sub-types (lines 1-5) and check if their names are unique using the build-in operation *isUnique* (line 6).

```
Class.allInstances() ->select(oe |
        oe.oclAsType(Class).getAppliedStereotypes().name ->includes('Domain') or
        oe.oclAsType(Class).getAppliedStereotypes().general.name ->includes('Domain')
            or
        oe.oclAsType(Class).getAppliedStereotypes().general.general.name
            ->includes('Domain') or
        oe.oclAsType(Class).getAppliedStereotypes().general.general.general.name
            ->includes('Domain'))
->select(c | c.oclAsType(Class).getValue(c.oclAsType(Class).getAppliedStereotypes()
    ->select(s |
        s.oclAsType(Stereotype).name ->includes('Domain') or
        s.oclAsType(Stereotype).general.name ->includes('Domain') or
        s.oclAsType(Stereotype).general.general.name ->includes('Domain') or
        s.oclAsType(Stereotype).general.general.general.name ->includes('Domain'))
        ->asSequence() ->first(),'abbreviation')=null) ->size()=0
```

Listing C.4: *The property `abbreviation` of a domain must be set*

First, we select all classes marked with the stereotype ≪Domain≫ or any of its sub-types (lines 1-5). Second, we select the values (keyword *getValue*, line 6) for the selected stereotypes (lines 6-10) and check whether the resulting bag of *abbreviation*-properties with the value *null* is empty (line 11).

```
 1 Class.allInstances() ->select(oe |
 2         oe.oclAsType(Class).getAppliedStereotypes().name ->includes('Domain') or
 3         oe.oclAsType(Class).getAppliedStereotypes().general.name ->includes('Domain')
               or
 4         oe.oclAsType(Class).getAppliedStereotypes().general.general.name
               ->includes('Domain') or
 5         oe.oclAsType(Class).getAppliedStereotypes().general.general.general.name
               ->includes('Domain'))
 6 ->collect(c | c.oclAsType(Class).getValue(c.oclAsType(Class).getAppliedStereotypes()
       ->select(s |
 7         s.oclAsType(Stereotype).name ->includes('Domain') or
 8         s.oclAsType(Stereotype).general.name ->includes('Domain') or
 9         s.oclAsType(Stereotype).general.general.name ->includes('Domain') or
10         s.oclAsType(Stereotype).general.general.general.name ->includes('Domain'))
11         ->asSequence() ->first(),'abbreviation')) ->isUnique(oclAsType(String))
```

Listing C.5: *The abbreviation of a domain must be unique*

We select the classes having the stereotype ≪Domain≫ or any of its sub-types assigned (lines 1-5). For these classes, we collect the property values (lines 6-10) and check whether the value of the property *abbreviation* is unique (line 11).

```
 1 Class.allInstances() ->forAll(oe |
 2         (oe.oclAsType(Class).getAppliedStereotypes().name ->includes('Domain') or
 3         oe.oclAsType(Class).getAppliedStereotypes().general.name ->includes('Domain')
               or
 4         oe.oclAsType(Class).getAppliedStereotypes().general.general.name
               ->includes('Domain') or
 5         oe.oclAsType(Class).getAppliedStereotypes().general.general.general.name
               ->includes('Domain'))
 6         implies
 7         oe.oclAsType(Class).getAppliedStereotypes() ->forAll(s |
 8                 (s.oclAsType(Stereotype).name ->includes('Domain') or
 9                 s.oclAsType(Stereotype).general.name ->includes('Domain') or
10                 s.oclAsType(Stereotype).general.general.name ->includes('Domain') or
11                 s.oclAsType(Stereotype).general.general.general.name
                       ->includes('Domain'))
12             implies
13             oe.oclAsType(Class).getValue(s,'abbreviation')=
14             oe.oclAsType(Class).getValue(oe.oclAsType(Class).getAppliedStereotypes()
                   ->select(s1 |
15                 s1.oclAsType(Stereotype).name ->includes('Domain') or
16                 s1.oclAsType(Stereotype).general.name ->includes('Domain') or
17                 s1.oclAsType(Stereotype).general.general.name ->includes('Domain') or
18                 s1.oclAsType(Stereotype).general.general.general.name
                       ->includes('Domain'))
19                 ->asSequence() ->first(),'abbreviation')))
```

Listing C.6: *The same abbreviation is used for all stereotypes of a domain*

It is possible to assign more than one stereotype to one domain. However, we only have one abbreviation per domain. Thus, the abbreviation property for the different domain stereotypes assigned to one domain must be the same. We check this condition as follows: For each class with the stereotype ≪Domain≫ or any its sub-types (lines 1-5) the domain stereotype (*s*, lines 7-11) must have the same (line 13) abbreviation as the first domain stereotype (line 14-19).

```
Dependency.allInstances()->select(
  getAppliedStereotypes().name-> includes('isPart')
      ).source->forAll(oclIsTypeOf(Package) or oclIsTypeOf(Model))
```

Listing C.7: *The ≪isPart≫-dependencies are only allowed with a package or a model as source*

This expression checks whether all dependencies with the stereotype ≪isPart≫ (line 1) have a package or the model as source (line 2).

```
1 Interface.allInstances() ->forAll(i | let dif: Set(Dependency) =
2         Dependency.allInstances()->select (d | d.target
              ->exists(oclAsType(Interface)=i))
3 in
4         dif->select(r | r.oclAsType(Relationship).getAppliedStereotypes().name
5         ->includes('controls')) ->size()=1
6         implies dif ->select(r |
              r.oclAsType(Relationship).getAppliedStereotypes().name
7         ->includes('observes')) ->size()>=1 )
```

Listing C.8: *A controlled interface must be observed by at least one domain*

We check that a relationship with the stereotype ≪controls≫ (lines 4 and 5) found within the set of interfaces (lines 1 and 2) must have at least one ≪observes≫ relationship (lines 6 and 7).

```
1  Dependency.allInstances() ->select(getAppliedStereotypes() .name
2         ->includes('observes') ).target ->forAll(ot | Dependency.allInstances()
3                 ->select(getAppliedStereotypes().name ->includes('controls')   )
4                 ->select(target->exists(ct| ct=ot)   )->size()=1 )
```

Listing C.9: *An observed interface must be controlled by exactly one domain*

An observed interface must be controlled by exactly one domain. We select all ≪observes≫-dependency targets (lines 1 and 2). Next, we select the the ≪controls≫-dependencies belonging to the aforementioned dependencies (lines 2 and 3). Afterwards, we check for each ≪controls≫-dependency whether there exists exactly one target (line 4). Note that used and provided interfaces of the architecture are not being considered at this place.

C.2. Constraints related to the context diagram

```
1 (*@\label{OneCDOne}@*) Package.allInstances()  ->select(getAppliedStereotypes().name
2         ->includes('ContextDiagram'))  ->size()=1
```

Listing C.10: *There must be exactly one context diagram*

All packages in the model (line 1) having the stereotype ≪ContextDiagram≫ assigned (keyword *getAppliedStereotypes* in line 2) are selected. We then check that the number of packages in the resulting bag is equal to one (line 2).

```
Package.allInstances() ->select(p |  p.oclAsType(Package).getAppliedStereotypes()
    .name
        ->includes('ContextDiagram')).ownedElement
            ->forAll(oe |
                (oe.oclIsTypeOf(Class) and
                  (oe.oclAsType(Class).getAppliedStereotypes().name
                      ->includes('Domain') or
                  oe.oclAsType(Class).getAppliedStereotypes().general.name
                      ->includes('Domain') or
                  oe.oclAsType(Class).getAppliedStereotypes().general.general.name
                      ->includes('Domain') or
                  oe.oclAsType(Class).getAppliedStereotypes().general.general
                                  .general.name ->includes('Domain') ) or
                oe.oclIsTypeOf(Interface) or
                (oe.oclIsTypeOf(Association) and
                  (oe.oclAsType(Association).getAppliedStereotypes().name
                      ->includes('connection') or
                  oe.oclAsType(Association).getAppliedStereotypes().general.name
                      ->includes('connection') or
                  oe.oclAsType(Association).getAppliedStereotypes().general.general
                                    .name ->includes('connection') or
16                    oe.oclAsType(Association).getAppliedStereotypes().general.general
17                                        .general.name ->includes('connection') ) ) or
18                  (oe.oclIsTypeOf(Dependency) and
19                    (oe.oclAsType(Dependency).getAppliedStereotypes().name
                          ->includes('controls') or
20                    oe.oclAsType(Dependency).getAppliedStereotypes().name
                          ->includes('observes') or
21                    oe.oclAsType(Dependency).getAppliedStereotypes().name
                          ->includes('isPart'))
22                  )or
23                  oe.oclIsTypeOf(Comment)
24          )
25 )
```

Listing C.11: *Allowed elements for a context diagram*

First, we select the package that is annotated with the stereotype ≪ContextDiagram≫ (lines 1-2) as well as all the elements associated to it (keyword *ownedElement*; line 2). Second, we check for each owned element *oe* (line 3) whether it is allowed to appear in a context diagram. The allowed elements are:

- Classes with the stereotype ≪Domain≫ or any of its sub-types (lines 4 - 9) assigned.
- Interfaces (line 10). Currently no restrictions considering the stereotypes for interfaces exist.
- Associations with the stereotype ≪connection≫ or any of its sub-types, e.g., ≪ui≫ for a user interface assigned (lines 11-17).
- Dependencies with ≪controls≫, ≪observes≫, or ≪isPart≫ as stereotype.
- Comments (line 23). Currently, no limitations considering comments exist.

```
1 Package.allInstances() ->select(p |
2   p.oclAsType(Package).getAppliedStereotypes().name ->includes('ContextDiagram')
3 ) -> forAll (p |
4   p.clientDependency ->select(getAppliedStereotypes().name ->includes('isPart'))
5   .target->select(cd_elem |
6     cd_elem .oclIsTypeOf(Class) and cd_elem
          .oclAsType(Class).getAppliedStereotypes().name ->includes('Machine')
7   ) ->size()>=1
8 )
```

Listing C.12: *A context diagram has at least one machine domain*

A context diagram must contain at least one machine: We first select all packages with the appropriate stereotype, i.e, ≪ContextDiagram≫ (lines 1 and 2). For the resulting package (we only have one package with this stereotype assigned, see Listing C.10 on the previous page) we collect all dependencies (keyword *clientDependency* line 4) and select those with the stereotype ≪isPart≫ (line 4). Using the target ends of the dependencies, we collect all elements of the package and select (line 5) those (cd_elem; line 6) being classes with the stereotype ≪Machine≫ (line 6) assigned. The size of the resulting bag must be greater than or equal to one (line 7).

```
1 Package.allInstances() ->select(p |  p.oclAsType(Package).getAppliedStereotypes()
      .name
2         ->includes('ContextDiagram')).clientDependency.target
3 ->select(oclIsTypeOf(Class)).oclAsType(Class)
4 ->select(oe | (
5   oe.oclAsType(Class).getAppliedStereotypes().name ->includes('BiddableDomain') or
6   oe.oclAsType(Class).getAppliedStereotypes().general.name
        ->includes('BiddableDomain')
7 ) and (
8   oe.oclAsType(Class).getAppliedStereotypes().name ->includes('CausalDomain') or
9   oe.oclAsType(Class).getAppliedStereotypes().general.name ->includes('CausalDomain')
        or
10  oe.oclAsType(Class).getAppliedStereotypes().general.general.name
        ->includes('CausalDomain')
11 ) ) ->size()=0
```

Listing C.13: *The stereotypes biddable and causal are not allowed to appear together*

It is possible to assign several stereotypes to one domain. However, not all combinations are allowed. For instance, it is not possible to have a domain which is biddable and causal at the same time. To check that the stereotypes, ≪BiddableDomain≫ and ≪CausalDomain≫ are not applied to the same class, we select the classes belonging to the context diagram (lines 1-3). Next, we select those classes that have the stereotype ≪BiddableDomain≫ or a direct sub-type of ≪BiddableDomain≫ (lines 4-7) and the stereotype ≪CausalDomain≫ or a sub-type of ≪CausalDomain≫ (lines 8-10) assigned. We then check whether the resulting bag is empty, i.e., the size of the bag is equal to zero (line 11).

```
1 Package.allInstances()->select(getAppliedStereotypes().name
      ->includes('ContextDiagram'))->asSequence()->first()
2 .ownedElement ->select(oclIsTypeOf(Association)).oclAsType(Association)
3 ->forAll(a: Association |
4    not (
5     a.endType
          ->select(oclIsTypeOf(Class)).oclAsType(Class).getAppliedStereotypes().name
          ->includes('BiddableDomain')
6     and a.endType
          ->select(oclIsTypeOf(Class)).oclAsType(Class).getAppliedStereotypes().name
          ->includes('LexicalDomain')
7   )
8 )
```

Listing C.14: *Biddable domains are not directly connected to lexical domains*

A biddable domain cannot directly interact with a lexical domain. Therefore, we check this condition as follows: For all associations of the context diagram (lines 1 and 2) we check that the association ends (keyword *endType*) do not connect biddable and lexical domains (lines 3-6).

```
1  Package.allInstances()
2  ->select(p | p.oclAsType(Package).getAppliedStereotypes().name
3        ->includes('ContextDiagram')  )
4  .clientDependency.target ->select(oe |
5          oe.oclAsType(Class).getAppliedStereotypes().name
               ->includes('ConnectionDomain'))
6  .oclAsType(Class)->forAll(c |
7          c->select (clientDependency.getAppliedStereotypes().name
               ->includes('observes')) ->size()>=1
8          and
9          c->select (clientDependency.getAppliedStereotypes().name
               ->includes('controls')) ->size()>=1
10 )
```

Listing C.15: *Connection domains in the context diagram have at least one observed and one controlled interface*

Connection domains in the context diagram (lines 1-6) must have at least one observed interface (line 7) and one controlled interface (line 9).

```
1 Package.allInstances()
2  ->select(p | p.oclAsType(Package).getAppliedStereotypes().name
3        ->includes('ContextDiagram')  )
4  .clientDependency.target  ->select(getAppliedStereotypes().name
       ->includes('Machine') )
5          ->forAll(   oclAsType(Class).clientDependency
6         ->select(getAppliedStereotypes().name->includes('controls'))
7            ->size()>=1   or
8             oclAsType(Class).generalization->size()>0 )
```

Listing C.16: *Each machine controls at least one interface*

To check that each machine controls at least one interface each, all classes in the context diagram with the stereotype ≪Machine≫ are selected (lines 1-3). For these classes, the dependencies (keyword *clientDependency*, line 5) with the stereotype ≪controls≫ assigned are collected (line 6). The number of these dependencies must be greater than or equal to one (line 7). This condition is not necessary for classes being a specialization of another machine (using generalization, line 8).

```
1 Package.allInstances() ->select(p |  p.oclAsType(Package).getAppliedStereotypes()
      .name
2         ->includes('ContextDiagram')).clientDependency.target
3 ->select(oclIsTypeOf(Class)).oclAsType(Class)
4 ->forAll(oe | (
5   oe.oclAsType(Class).getAppliedStereotypes().name ->includes('DesignedDomain')
6 ) implies (
7   oe.oclAsType(Class).getAppliedStereotypes().name ->includes('CausalDomain') or
8   oe.oclAsType(Class).getAppliedStereotypes().general.name ->includes('CausalDomain')
9 ) )
```

Listing C.17: *Designed domains are always lexical*

We check within the classes of the context diagram (lines 1-3), if a class with the stereotype ≪DesignedDomain≫ also has the stereotype ≪CausalDomain≫ or any of its sub-types assigned (line 4-8). Should this be the case, the condition fails.

C.3. Constraints related to problem diagrams and problem frames

```
1  Package.allInstances() ->select(p |
2    let n: Bag(String) = p.oclAsType(Package).getAppliedStereotypes().name
3    in  n->includes('ProblemDiagram') or n->includes('ProblemFrame'))
4        .ownedElement
5        ->forAll(el | (el.oclIsTypeOf(Class) and
6          (el.oclAsType(Class).getAppliedStereotypes().name ->includes('Domain') or
7          el.oclAsType(Class).getAppliedStereotypes().general.name ->includes('Domain')
              or
8          el.oclAsType(Class).getAppliedStereotypes().general.general.name
              ->includes('Domain') or
9          el.oclAsType(Class).getAppliedStereotypes().general.general
10                                    .general.name ->includes('Domain') or
11         el.oclAsType(Class).getAppliedStereotypes().name ->includes('Statement') or
12         el.oclAsType(Class).getAppliedStereotypes().general.name
              ->includes('Statement') or
13         el.oclAsType(Class).getAppliedStereotypes().general.general.name
              ->includes('Statement') or
14         el.oclAsType(Class).getAppliedStereotypes().general.general
15                        .general.name ->includes('Statement'))) or
16         el.oclIsTypeOf(Interface) or (el.oclIsTypeOf(Association) and
17         (el.oclAsType(Association).getAppliedStereotypes().name
              ->includes('connection') or
18         el.oclAsType(Association).getAppliedStereotypes().general.name
              ->includes('connection') or
19         el.oclAsType(Association).getAppliedStereotypes().general.general
20                                       .name ->includes('connection') or
21         el.oclAsType(Association).getAppliedStereotypes().general.general
22                                 .general.name ->includes('connection') or
23         el.oclAsType(Association).endType->forAll(
24           getAppliedStereotypes().name->includes('Statement')  or
25           getAppliedStereotypes().general.name->includes('Statement')))
26         ) or
27     (el.oclIsTypeOf(Dependency) and
28         (el.oclAsType(Dependency).getAppliedStereotypes().name ->includes(’refersTo’)
              or
29         el.oclAsType(Dependency).getAppliedStereotypes().name
              ->includes(’constrains’) or
30         el.oclAsType(Dependency).getAppliedStereotypes().name ->includes(’controls’)
              or
31         el.oclAsType(Dependency).getAppliedStereotypes().name ->includes(’observes’)
              or
32         el.oclAsType(Dependency).getAppliedStereotypes().name ->includes(’isPart’) or
33         el.oclAsType(Dependency).getAppliedStereotypes().name
              ->includes(’complements’))
34         )or
35     el.oclIsTypeOf(CallEvent) or
36     el.oclIsTypeOf(ProfileApplication) or
37     el.oclIsTypeOf(Comment)
38 )
```

Listing C.18: *Allowed elements for a problem diagram/frame*

First, we retrieve all packages (line 1). Second, we collect those packages that have the stereotypes ≪ProblemDiagram≫ or ≪ProblemFrame≫ (lines 2-3) assigned. as well as all associated elements (keyword *ownedElement*; line 4). Third, we check for each owned element *el* (line 4-5) whether it is:

- a class with the stereotype ≪Domain≫ or any of its sub-types (lines 5-10) assigned. Additionally, a class may also have the stereotype ≪Statement≫ or a sub-type, e.g., ≪Requirement≫ (lines 10-15)assigned.
- an interface (line 16; no restrictions considering the stereotypes apply here).
- an association. An association must have the stereotype ≪connection≫ or any of its sub-types assigned. (lines 16-22).
- an association between statements (e.g., aggregation) (lines 23-25).
- a dependency (line 27). In this case, it must have either ≪refersTo≫, ≪constrains≫, ≪controls≫, ≪observes≫, ≪isPart≫, or ≪complements≫ as stereotype (lines 28-33).
- a comment (line 37).

  For technical reasons, i.e., compatibility to PapyrusUML, *Profile Application* are allowed (lines 35-36), as well.

```
Package.allInstances() ->select(p |  p.oclAsType(Package).getAppliedStereotypes()
    .name ->includes(’ProblemDiagram’) or
    p.oclAsType(Package).getAppliedStereotypes() .name ->includes(’ProblemFrame’))
.clientDependency.target
->select(oclIsTypeOf(Class)).oclAsType(Class)
->select(oe | (
  oe.oclAsType(Class).getAppliedStereotypes().name ->includes(’BiddableDomain’) or
  oe.oclAsType(Class).getAppliedStereotypes().general.name
      ->includes(’BiddableDomain’)
) and (
  oe.oclAsType(Class).getAppliedStereotypes().name ->includes(’CausalDomain’) or
  oe.oclAsType(Class).getAppliedStereotypes().general.name ->includes(’CausalDomain’)
      or
  oe.oclAsType(Class).getAppliedStereotypes().general.general.name
      ->includes(’CausalDomain’)
) ) ->size()=0
```

Listing C.19: *The stereotypes biddable and causal are not allowed to appear together*

To check that the stereotypes, ≪BiddableDomain≫ and ≪CausalDomain≫ are not applied to the same class, we select the classes belonging to the problem diagram or problem frame (lines 1-3). Next, we select those classes that have the stereotype ≪BiddableDomain≫ or a direct sub-type of ≪BiddableDomain≫ (lines 4-7) and the stereotype ≪CausalDomain≫ or a sub-type of ≪CausalDomain≫ (lines 8-10) assigned. We then check whether the resulting bag is empty, i.e., the size of the bag is equal to zero (line 11).

```
1 Package.allInstances() ->select(p |
2   p.oclAsType(Package).getAppliedStereotypes().name ->includes(’ProblemDiagram’) or
3   p.oclAsType(Package).getAppliedStereotypes().name ->includes(’ProblemFrame’)
4 ) -> forAll (p |
5   p.clientDependency ->select(getAppliedStereotypes().name ->includes(’isPart’))
6   .target->select(pdf_elem |
7     pdf_elem .oclIsTypeOf(Class) and pdf_elem
          .oclAsType(Class).getAppliedStereotypes().name ->includes(’Machine’)
8   ) ->size()=1
9 )
```

Listing C.20: *A problem diagram/frame has exactly one machine domain*

A problem diagram/problem frame must contain exactly one machine: We first select all packages with the appropriate stereotype, i.e, ≪ProblemDiagram≫ or ≪ProblemFrame≫ (lines 1 -3). For each package we collect all dependencies (using clientDependency line 5) and select those with the stereotype ≪isPart≫ (line 5). Using the target ends of these dependencies, we collect all elements of the package and select (line 6) those elements (pdf_elem; line 6) being classes with the stereotype ≪Machine≫ (line 7). The size of the resulting bag must be greater than or equal to one (line 8).

```
1 Dependency.allInstances()
2 ->select(a | a.oclAsType(Dependency).getAppliedStereotypes().name
      ->includes(’constrains’)) ->forAll(
3         source.getAppliedStereotypes() .name->includes(’Requirement’) implies  not
4                 target.getAppliedStereotypes(). name->includes(’Machine’)
5                 or (    target.getAppliedStereotypes().name
                      ->includes(’CausalDomain’) or
6                         target.getAppliedStereotypes().general.name
                              ->includes(’CausalDomain’) or
7                         target.getAppliedStereotypes().general.general.name
                              ->includes(’CausalDomain’)
8                 )
9 )
```

Listing C.21: *A requirement does not constrain a machine domain*

We retrieve all dependencies (line 1). After that, we select those dependencies that have the stereotype ≪constrains≫ assigned (line 2). It is then necessary to verify that all dependencies originating (keyword *source*) from a requirement (line 3) do not point (keyword *target*) to a machine domain (line 4). The expression also passes if this dependency points to a machine that is also a causal domain or a sub-type (in another subproblem) (line 5-7).

```
1 Dependency.allInstances()
2         ->select(  oclAsType(Dependency).getAppliedStereotypes().name
              ->includes(’constrains’))
3                 ->forAll(  source.getAppliedStereotypes()
                      .name->includes(’Requirement’) implies  not
4                  target.getAppliedStereotypes(). name->includes(’BiddableDomain’))
```

Listing C.22: *A requirement does not constrain a biddable domain*

We retrieve all dependencies (line 1). After that, we select those dependencies that have the stereotype ≪constrains≫ assigned (line 2). It is then necessary to verify that all dependencies originating (keyword *source*) from a requirement (line 3) do not point (keyword *target*) to a biddable domain (line 4).

```
1  Package.allInstances()
2         ->select(p | p.oclAsType(Package).getAppliedStereotypes().name
              ->includes('ProblemDiagram') or
3                  p.oclAsType(Package).getAppliedStereotypes().name
                      ->includes('ProblemFrame') )
4                  ->forAll(   clientDependency ->select(getAppliedStereotypes().name
                      ->includes('isPart')).target->select(oe |
                      oe.oclIsTypeOf(Class) and
                      oe.oclAsType(Class).getAppliedStereotypes().name
                      ->includes('Requirement'))
5                         ->size()>=1 )
```

Listing C.23: *A problem diagram/frame contains at least one requirement*

We select all packages that have the stereotypes ≪ProblemDiagram≫ or ≪ProblemFrame≫ assigned (lines 1-3). For each diagram, we check that it contains at least one requirement (lines 4 and 5).

```
1  Dependency.allInstances()
2  ->select(a |  a.oclAsType(Dependency).getAppliedStereotypes().name
       ->includes('constrains')  or
3               a.oclAsType(Dependency).getAppliedStereotypes().name
                   ->includes('refersTo') )
4  ->forAll(d | d.oclAsType(Dependency).target.getAppliedStereotypes().name
       ->includes('Domain') or
5              d.oclAsType(Dependency).target.getAppliedStereotypes().general.name
                  ->includes('Domain') or
6              d.oclAsType(Dependency).target.getAppliedStereotypes().general.general.name
                  ->includes('Domain')
7  ) and Dependency.allInstances()
8  ->select(a |  a.oclAsType(Dependency).getAppliedStereotypes().name
       ->includes('constrains')  or
9          a.oclAsType(Dependency).getAppliedStereotypes().name ->includes('refersTo') )
10 ->forAll(d | d.oclAsType(Dependency).source.getAppliedStereotypes().name
       ->includes('Statement') or
11             d.oclAsType(Dependency).source.getAppliedStereotypes().general.name
                  ->includes('Statement') or
12             d.oclAsType(Dependency).source.getAppliedStereotypes().general.general.name
                  ->includes('Statement') )
```

Listing C.24: *Dependencies with stereotypes ≪constrains≫ or ≪refersTo≫ point from statements to domains*

A domain can never be the origin of a ≪constrains≫ or ≪refersTo≫ dependency. We check this condition as follows: We select the dependencies that have the stereotypes ≪constrains≫ or ≪refersTo≫ assigned (lines 1-3). We then select all dependencies that point (keyword *target*, line 4) to a domain or a sub-type of domain (lines 4-6). We check that the dependencies having ≪constrains≫ or ≪refersTo≫ as stereotype (lines 8-9) originate (keyword *source*, line 8) from a statement or sub-type of statement (lines 10-12).

```
1  Package.allInstances()
2  ->select(p | p.oclAsType(Package).getAppliedStereotypes().name
       ->includes('ProblemDiagram') or
3             p.oclAsType(Package).getAppliedStereotypes().name
                 ->includes('ProblemFrame') )
4  .clientDependency.target ->select(oe |
5          oe.oclAsType(Class).getAppliedStereotypes().name
               ->includes('ConnectionDomain'))
6  .oclAsType(Class)->forAll(c |
7          c->select (clientDependency.getAppliedStereotypes().name
               ->includes('observes')) ->size()>=1
8          and
```

```
 9        c->select (clientDependency.getAppliedStereotypes().name
              ->includes(’controls’)) ->size()>=1
10 )
```

Listing C.25: *Connection domains in a problem diagram or problem frame have at least one observed and one controlled interface*

Connection domains in problem diagrams or problem frames (lines 1-6) must have at least one observed interface (line 7) and one controlled interface (line 9).

```
1 Package.allInstances()
2  ->select(p | p.oclAsType(Package).getAppliedStereotypes().name
3        ->includes(’ProblemDiagram’) or
             p.oclAsType(Package).getAppliedStereotypes().name
4        ->includes(’ProblemFrame’) )
5  .clientDependency.target  ->select(getAppliedStereotypes().name
      ->includes(’Machine’) )
6         ->forAll(   oclAsType(Class).clientDependency
7         ->select(getAppliedStereotypes().name->includes(’controls’))
8            ->size()>=1   or
9             oclAsType(Class).generalization->size()>0 )
```

Listing C.26: *Each machine controls at least one interface*

To check that all machines control at least one interface, all classes in problem diagrams with the stereotype ≪Machine≫ are selected (lines 1-3). For these classes, the dependencies of the class (using *clientDependency*, line 5) with the stereotype ≪controls≫ assigned are collected (line 6). The number of these dependencies must be greater than or equal to one (line 7). This condition is not necessary for classes being a specialization of another machine (using generalization, lines 5 and 8).

```
1 Class.allInstances() ->select( getAppliedStereotypes().name ->includes(’Requirement’)
     or
2         getAppliedStereotypes().general.name -> includes(’Requirement’) or
3         getAppliedStereotypes().general.general.name -> includes(’Requirement’) or
4         getAppliedStereotypes().general.general.general.name ->
              includes(’Requirement’))
5                 ->reject(st | Class.allInstances().ownedElement
                      ->select(oclIsTypeOf(Property)).oclAsType(Property).type ->
                      includes(st))
6                 ->forAll(clientDependency->collect(r |
                      r.oclAsType(Dependency).getAppliedStereotypes().name
7                         -> includes(’constrains’))  ->count(true)>=1)
```

Listing C.27: *A requirement has at least one constrains dependency*

A requirement or composed requirement must have at least one constrains dependency: We collect all classes having the stereotype ≪Requirement≫ or any of its sub-types (lines 1 - 4) assigned. We ignore all requirements being part of another requirement (line 5). We then check for each of these dependencies that there is at least one ≪constrains≫-dependency (lines 6 and 7).

```
1 Dependency.allInstances() ->select(a |
2         a.oclAsType(Dependency).getAppliedStereotypes().name ->includes(’restricts’)
              or
3         a.oclAsType(Dependency).getAppliedStereotypes().name
              ->includes(’complements’) or
4         a.oclAsType(Dependency).getAppliedStereotypes().name ->includes(’similar’)
5 )
6 -> forAll(dep |
7   (
8     dep.source.getAppliedStereotypes().name-> includes(’Statement’) or
9     dep.source.getAppliedStereotypes().general.name-> includes(’Statement’) or
10     dep.source.getAppliedStereotypes().general.general.name-> includes('Statement') or
11     dep.source.getAppliedStereotypes().general.general.general.name->
           includes('Statement')
12   ) and
13   (
14     dep.target.getAppliedStereotypes().name-> includes('Statement') or
15     dep.target.getAppliedStereotypes().general.name-> includes('Statement') or
16     dep.target.getAppliedStereotypes().general.general.name-> includes('Statement') or
17     dep.target.getAppliedStereotypes().general.general.general.name->
           includes('Statement')
18   )
19 )
```

Listing C.28: *Allowed dependency stereotypes between statements*

To check that only allowed stereotypes are used for dependencies involving a statement or any of its sub-types, we collect all dependencies with the stereotypes ≪restricts≫, ≪complements≫, ≪similar≫, ≪remove≫, or ≪replaces≫ (lines 1-7) assigned and check for each dependency that it starts from a statement or any of its sub-types (lines 8-13) and that it points to a statement or any of its sub-types (lines 14-21).

```
1 Dependency.allInstances() ->select(a |
2         a.oclAsType(Dependency).getAppliedStereotypes().name ->includes('contains')
3 )
4 -> forAll(dep |
5     dep.source->forAll(oclIsTypeOf(Interface)) and
6     dep.target->forAll(oclIsTypeOf(Interface))
7 )
```

Listing C.29: *≪contains≫-dependencies are only between interfaces*

This expression checks if all dependencies with the stereotype ≪contains≫ (lines 1-3) are between two interfaces (lines 4-7), i.e., the source as well as the target of the dependency are both interfaces.

```
1 Dependency.allInstances() ->select(a |
2         a.oclAsType(Dependency).getAppliedStereotypes().name
              ->includes('concretizes')  or
3         a.oclAsType(Dependency).getAppliedStereotypes().name ->includes('refines')
4 )
5 -> forAll(dep |
6 (dep.source->forAll(oclIsTypeOf(Interface))
7 and
8 dep.target->forAll(oclIsTypeOf(Interface)) )
9 or
10     (dep.source.getAppliedStereotypes().name-> includes('Domain') or
11     dep.source.getAppliedStereotypes().general.name-> includes('Domain') or
12     dep.source.getAppliedStereotypes().general.general.name-> includes('Domain') or
13     dep.source.getAppliedStereotypes().general.general.general.name->
           includes('Domain') or
14     dep.source.getAppliedStereotypes().general.general.general.general.name->
           includes('Domain')
15     and
16     dep.target.getAppliedStereotypes().name-> includes('ConnectionDomain') or
17     dep.target.getAppliedStereotypes().general.name-> includes('ConnectionDomain') or
18     dep.target.getAppliedStereotypes().general.general.name->
           includes('ConnectionDomain')
19     )
20 or
21   (dep.source.getAppliedStereotypes().name-> includes('ConnectionDomain') or
22     dep.source.getAppliedStereotypes().general.name-> includes('ConnectionDomain') or
23     dep.source.getAppliedStereotypes().general.general.name->
           includes('ConnectionDomain')
24     and
25     ( dep.target.getAppliedStereotypes().name-> includes('Connection') or
26     dep.target.getAppliedStereotypes().general.name-> includes('Connection') or
27     dep.target.getAppliedStereotypes().general.general.name-> includes('Connection')
           or
28     dep.target.getAppliedStereotypes().general.general.general.name->
           includes('Connection') or
29     dep.target.getAppliedStereotypes().general.general.general.general.name->
           includes('Connection') or
30     dep.target->forAll(oclIsTypeOf(Interface)))
31   )
32 or
33   (dep.target.getAppliedStereotypes().name-> includes('Machine') and
34   (      dep.source.getAppliedStereotypes().name->
         includes('Implementable_architecture') or
35         dep.source.getAppliedStereotypes().name-> includes('Layered_architecture')
36   )
37   )
38 )
```

Listing C.30: *Allowed application of concretizes or refines dependencies*

This expression checks for all dependencies with the stereotypes ≪concretizes≫ or ≪refines≫ (lines 1-5) that the source and target are interfaces (lines 6-8), or that the source is a class with the stereotype ≪Domain≫ or any of its sub-types and the target is a class with the stereotype ≪ConnectionDomain≫ or any of its sub-types (lines 10-19), or that the source is a class with the stereotype ≪ConnectionDomain≫ or a subtype and the target is an interface or has the stereotype ≪Connection≫ (lines 21-31) assigned, or source and target have the stereotype ≪Machine≫ (lines 33-35).

```
1 Package.allInstances()->select(getAppliedStereotypes().name->includes('ProblemDiagram')
      or getAppliedStereotypes().name->includes('ProblemFrame'))
2 .ownedElement->select(oclIsTypeOf(Association)).oclAsType(Association)
3 ->forAll(a: Association |
4    not (
5     a.endType->select(oclIsTypeOf(Class)) .oclAsType(Class).getAppliedStereotypes()
          .name->includes('BiddableDomain')
6     and a.endType->select(oclIsTypeOf(Class))
          .oclAsType(Class).getAppliedStereotypes() .name->includes('LexicalDomain')
7   )
8 )
```

Listing C.31: *Biddable domains are not directly connected to lexical domains*

For all associations of problem diagrams or problem frames (lines 1 and 2), we check that the associations (keyword *endType*) do not connect biddable and lexical domains (lines 3-6).

```
1 Package.allInstances() ->select(p |  p.oclAsType(Package).getAppliedStereotypes()
      .name ->includes('ProblemDiagram') or
      p.oclAsType(Package).getAppliedStereotypes() .name
2         ->includes('ProblemFrame')).clientDependency.target
3 ->select(oclIsTypeOf(Class)).oclAsType(Class)
4 ->forAll(oe | (
5   oe.oclAsType(Class).getAppliedStereotypes().name ->includes('DesignedDomain')
6 ) implies (
7   oe.oclAsType(Class).getAppliedStereotypes().name ->includes('LexicalDomain') or
8   oe.oclAsType(Class).getAppliedStereotypes().general.name ->includes('LexicalDomain')
9 ) )
```

Listing C.32: *Designed domains are always lexical*

In the classes of the problem diagram and problem frame (lines 1-3), we check if a class with the stereotype ≪DesignedDomain≫ also has the stereotype ≪CausalDomain≫ or one of its sub-types assigned (line 4-8).

C.4. Constraints related to the consistency between the context diagram and problem diagrams

```
1  let m: Set(Class) =
2    Package.allInstances() ->select(getAppliedStereotypes().name
         ->includes('ContextDiagram')) ->asSequence() ->first()
3    .clientDependency.target
4    ->select(getAppliedStereotypes().name ->includes('Machine') or
5            getAppliedStereotypes().general.name ->includes('Machine'))
6    .oclAsType(Class) ->asSet()
7  in
8    m.oclAsType(Class).member
9    ->select(oclIsTypeOf(Property)).oclAsType(Property).type
10   ->select(cm |
11     cm->select(oclIsTypeOf(Class)) .oclAsType(Class).member
12     ->select(oclIsTypeOf(Property)).oclAsType(Property).type
13     ->select(getAppliedStereotypes().name ->includes('Machine') or
           getAppliedStereotypes().general.name ->includes('Machine'))->size()=0
14   )
15   ->union(
16     m.oclAsType(Class).member
17     ->select(oclIsTypeOf(Property)) .oclAsType(Property).type
18     ->select(oclIsTypeOf(Class)) .oclAsType(Class).member
19     ->select(oclIsTypeOf(Property)) .oclAsType(Property).type
20   )
21   ->select(oclIsTypeOf(Class)).oclAsType(Class)
22   ->select(getAppliedStereotypes().name ->includes('Machine') or
         getAppliedStereotypes().general.name ->includes('Machine')) ->asSet()
23   =
24     Package.allInstances() ->select(getAppliedStereotypes().name
           ->includes('ProblemDiagram'))
25     .clientDependency.target
26     ->select(getAppliedStereotypes().name ->includes('Machine') or
           getAppliedStereotypes().general.name ->includes('Machine'))
27     .oclAsType(Class) ->asSet()
```

Listing C.33: *The submachines of the problem diagrams must be part of the machine(s) of the context diagram*

A problem diagram is consistent to a context diagram, if the machine in the problem diagrams are part of the machines in the context diagram: First, we select the machine(s) of the context diagram (lines 1-7). Second, we collect the set of all directly contained machines (lines 8-9) for each machine in the context diagram. Third, we only consider machines with no contained machines (lines 10-14) and unify the machines with no contained machines with the machines contained in the other machines (lines 15-22). This set must be the same as the set of all machines found in problem diagrams (lines 23-27).

```
let cd_domains: Set(Class) =
  Package.allInstances() ->select(getAppliedStereotypes().name
      ->includes('ContextDiagram')) ->asSequence() ->first()
  .clientDependency.target
  ->select(
        getAppliedStereotypes().name ->includes('Domain') or
        getAppliedStereotypes().general.name ->includes('Domain') or
        getAppliedStereotypes().general.general.name ->includes('Domain') or
        getAppliedStereotypes().general.general.general.name ->includes('Domain'))
   .oclAsType(Class) ->asSet()
in
let cd_spec_dom: Set(Class) =
  Class.allInstances()->select(c| cd_domains->exists(cdd| c.general()->includes(cdd)))
in
let cd_contained_domains: Set(Class) =
  cd_domains.member
  ->select(oclIsTypeOf(Property)).oclAsType(Property).type
  ->select(oclIsTypeOf(Class)).oclAsType(Class) ->asSet()
in
let cd_ccontained_domains: Set(Class) =
  cd_contained_domains
  ->select(oclIsTypeOf(Property)).oclAsType(Property).type
  ->select(oclIsTypeOf(Class)).oclAsType(Class) ->asSet()
```

```
23 in
24 let cd_combined_domains: Set(Class) =
25   Class.allInstances()
26   ->select(cl |
27     let cl_members: Set(Class) =
28       cl.member
29       ->select(oclIsTypeOf(Property)) .oclAsType(Property).type
30       ->select(oclIsTypeOf(Class)) .oclAsType(Class) ->asSet()
31     in
32       cl_members->exists(clm | cd_domains->includes(clm))
33   )
34 in
35 let cd_connection_ifs: Set(Interface)=
36   Package.allInstances() ->select(getAppliedStereotypes().name
       ->includes('ContextDiagram')) ->asSequence() ->first()
37   .clientDependency.target
38   ->select(oclIsTypeOf(Interface)).oclAsType(Interface) ->asSet()
39 in
40 let cd_connection_if_parts: Set(Interface)=
41   cd_connection_ifs.member
42   ->select(oclIsTypeOf(Property)).oclAsType(Property).type
43   ->select(oclIsTypeOf(Interface)).oclAsType(Interface) ->asSet()
44 in
45   Package.allInstances() ->select(getAppliedStereotypes().name
       ->includes('ProblemDiagram'))
46   ->forAll(pd_tcd |
47     pd_tcd.clientDependency -> select(getAppliedStereotypes().name
         ->includes('isPart')).target
48     ->select(oclIsTypeOf(Class))
49     ->select(
50       getAppliedStereotypes().name ->includes('Domain') or
51       getAppliedStereotypes().general.name ->includes('Domain') or
52       getAppliedStereotypes().general.general.name ->includes('Domain') or
53       getAppliedStereotypes().general.general.general.name
           ->includes('Domain')).oclAsType(Class)
54     ->forAll(pd_domain |
55       cd_domains->includes(pd_domain) or
56       cd_domains.general()->includes(pd_domain) or
57       cd_spec_dom->includes(pd_domain) or
58       cd_contained_domains->includes(pd_domain) or
59       cd_ccontained_domains->includes(pd_domain) or
60       cd_combined_domains->includes(pd_domain)  or
61       let concr_ifs_of_pd_domain: Set(Interface) =
62         pd_domain.clientDependency->select(
63           getAppliedStereotypes().name ->includes('concretizes') or
64           getAppliedStereotypes().name ->includes('refines')
65         ).target.oclAsType(Interface) ->asSet()
66       in
67         concr_ifs_of_pd_domain -> exists(if_pd |
68           cd_connection_ifs->includes(if_pd) or
69           cd_connection_if_parts->includes(if_pd)
70         )
71     )  )
```

Listing C.34: *Domains in the problem diagrams are consistent to the domains in the context diagram*

A problem diagram is consistent to a context diagram, if additionally each domain in the problem diagram (lines 45-54)

- is a domain in the context diagram (line 55), or
- is a specialization of a domain of the context diagram (line 56), or
- is a generalization of a domain of the context diagram (line 57), or
- is part of a domain in the context diagram (line 58 and line 59, operator: split domain), or
- is composed of several domains being part of the context diagram (line 60, operator: merge domain), or

- refines or concretized an interface in the context diagram (lines 61-68, operator: introduce connection domain)
- refines or concretized a part of an interface in the context diagram (line 69, operator: introduce connection domain)

```
1  let cd_domains: Set(Class) =
2      Package.allInstances() ->select(getAppliedStereotypes().name
           ->includes(’ContextDiagram’)) ->asSequence() ->first()
3      .clientDependency.target
4      ->select(
5        getAppliedStereotypes().name ->includes(’Domain’) or
6        getAppliedStereotypes().general.name ->includes(’Domain’) or
7        getAppliedStereotypes().general.general.name ->includes(’Domain’) or
8        getAppliedStereotypes().general.general.general.name ->includes(’Domain’)
9      ).oclAsType(Class) ->asSet()
10   in
11   let cd_contained_domains: Set(Class) =
12     cd_domains.member
13     ->select(oclIsTypeOf(Property)).oclAsType(Property).type
14     ->select(oclIsTypeOf(Class)).oclAsType(Class) ->asSet()
15   in
16   let ifs_between_split_dom: Set(Interface) =
17     Interface.allInstances()->select(i |
18         cd_contained_domains.clientDependency ->select(getAppliedStereotypes().name
               ->includes(’controls’)).target ->includes(i) and
19         cd_contained_domains.clientDependency ->select(getAppliedStereotypes().name
               ->includes(’observes’)).target ->includes(i)
20      )
21   in
22   let cd_ifs: Set(Interface) =
23     Package.allInstances() ->select(getAppliedStereotypes().name
           ->includes(’ContextDiagram’)) ->asSequence() ->first()
24     .clientDependency.target
25     ->select(oclIsTypeOf(Interface)).oclAsType(Interface) ->asSet()
26   in
27   let cd_if_parts: Set(Interface) =
28     cd_ifs.member
29     ->select(oclIsTypeOf(Property)).oclAsType(Property).type
30     ->select(oclIsTypeOf(Interface)).oclAsType(Interface) ->asSet()
31   in
32   let cd_concr_ifs: Set(Interface)=
33     cd_ifs.clientDependency->select(
34       getAppliedStereotypes().name ->includes(’concretizes’) or
35       getAppliedStereotypes().name ->includes(’refines’) or
36       getAppliedStereotypes().name ->includes(’contains’)
37     ).target
38     ->select(oclIsTypeOf(Interface)).oclAsType(Interface) ->asSet()
39   in
40   let cd_concr_concr_ifs: Set(Interface)=
41     cd_concr_ifs.clientDependency->select(
42          getAppliedStereotypes().name ->includes(’concretizes’) or
43          getAppliedStereotypes().name ->includes(’refines’) or
44                 getAppliedStereotypes().name ->includes(’contains’)
45     ).target
46     ->select(oclIsTypeOf(Interface)).oclAsType(Interface) ->asSet()
47   in
48   let cd_conn_doms: Set(Class) =
49     Class.allInstances() ->select(cl |
50       cl.clientDependency
51       ->select(
52    getAppliedStereotypes().name ->includes(’concretizes’) or
53    getAppliedStereotypes().name ->includes(’refines’) )
54       ->select(target->asSequence() ->first().oclIsTypeOf(Interface))
55       ->exists(
56         cd_ifs->includes(target.oclAsType(Interface) ->asSequence() ->first()) or
57         cd_if_parts->includes(target.oclAsType(Interface) ->asSequence() ->first())
58       )
59     ) ->asSet()
60   in
```

```
61  let cd_con_dom_if: Set(Interface) =
62    cd_conn_doms.clientDependency->select(
63         getAppliedStereotypes().name ->includes('observes') or
64         getAppliedStereotypes().name ->includes('controls')
65    ).target
66    ->select(oclIsTypeOf(Interface)).oclAsType(Interface) ->asSet()
67  in
68    Package.allInstances() ->select(getAppliedStereotypes().name
        ->includes('ProblemDiagram'))
69    ->forAll(pd_tcd |
70      pd_tcd.clientDependency -> select(getAppliedStereotypes().name
          ->includes('isPart')).target
71      ->select(oclIsTypeOf(Interface)).oclAsType(Interface) ->asSet()
72      ->forAll(pd_if |
73   cd_ifs->includes(pd_if) or
74   cd_if_parts->includes(pd_if) or
75   cd_concr_ifs->includes(pd_if) or
76   cd_concr_concr_ifs->includes(pd_if) or
77   cd_con_dom_if->includes(pd_if) or
78   ifs_between_split_dom->includes(pd_if) or
79   let pd_concr_ifs: Set(Interface)=
80     pd_if.clientDependency->select(
81          getAppliedStereotypes().name ->includes('concretizes') or
82          getAppliedStereotypes().name ->includes('refines') or
83                        getAppliedStereotypes().name ->includes('contains')
84     ).target
85     ->select(oclIsTypeOf(Interface)).oclAsType(Interface) ->asSet()
86   in
87     pd_concr_ifs -> exists(if_pd | cd_ifs->includes(if_pd))
88   or
89   let pd_concr_domains: Set(Interface)=
90     pd_if.clientDependency->select(
91          getAppliedStereotypes().name ->includes('concretizes') or
92          getAppliedStereotypes().name ->includes('refines')
93     ).target
94     ->select(oclIsTypeOf(Class)).oclAsType(Interface) ->asSet()
95   in
96     pd_concr_domains-> exists(if_pd | cd_ifs->includes(if_pd))
97      )
98    )
```

Listing C.35: *Interfaces in the problem diagrams are consistent to the interfaces in the context diagram*

A problem diagram is consistent to a context diagram, if also each interface in the problem diagram (lines 68-72)

- is a (new) interface between a domain and a new part of the domain (line 78, operator: split domain),
- is an interface in the context diagram (line 73),
- is part of an interface (line 74, operator: reduce interface) in the context diagram,
- refines or concretizes an interface in the context diagram (line 75),
- refines or concretizes an interface in the context diagram indirectly (line 76),
- is observed or controlled by a connection domain introduced for an interface (or a part of this interface) in the context diagram (line 77),
- is refined or concretized by an interface in the context diagram (lines 79-87), or
- refines or concretizes a (connection-)domain in the context diagram (lines 89-96).

```
let cd_domains: Set(Class) =
  Package.allInstances() ->select(getAppliedStereotypes().name
      ->includes('ContextDiagram')) ->asSequence() ->first()
  .clientDependency.target
  ->select(
        getAppliedStereotypes().name ->includes(’Domain’) or
        getAppliedStereotypes().general.name ->includes(’Domain’) or
        getAppliedStereotypes().general.general.name ->includes(’Domain’) or
        getAppliedStereotypes().general.general.general.name ->includes(’Domain’))
   .oclAsType(Class) ->asSet()
in
let cd_domains_obs_if: Set(Interface) =
  cd_domains.clientDependency
  ->select(getAppliedStereotypes().name ->includes(’observes’))
  .target.oclAsType(Interface) ->asSet()
in
let cd_spec_dom: Set(Class) =
  Class.allInstances()->select(c| cd_domains->exists(cdd| c.general()->includes(cdd)))
in
let cd_spec_dom_obs_if: Set(Interface) =
  cd_domains.clientDependency
  ->select(getAppliedStereotypes().name ->includes(’observes’))
  .target.oclAsType(Interface) ->asSet()
in
let cd_gene_dom_obs_if: Set(Interface) =
  cd_domains.general().clientDependency
  ->select(getAppliedStereotypes().name ->includes(’observes’))
  .target.oclAsType(Interface) ->asSet()
in
let cd_domain_parts: Set(Class) =
  cd_domains.member
  ->select(oclIsTypeOf(Property)).oclAsType(Property).type
  ->select(oclIsTypeOf(Class)).oclAsType(Class) ->asSet()
in
let cd_domain_parts_obs_if: Set(Interface) =
  cd_domain_parts.clientDependency
  ->select(getAppliedStereotypes().name ->includes(’observes’))
  .target.oclAsType(Interface) ->asSet()
in
let cd_merged_domains: Set(Class) =
  Class.allInstances()
  ->select(cl |
    let cl_members: Set(Class) =
      cl.member
      ->select(oclIsTypeOf(Property)) .oclAsType(Property).type
      ->select(oclIsTypeOf(Class)) .oclAsType(Class) ->asSet()
    in
      cl_members->exists(clm | cd_domains->includes(clm))
  )
in
let cd_merged_domains_obs_if: Set(Interface) =
  cd_merged_domains.clientDependency
  ->select(getAppliedStereotypes().name ->includes(’observes’))
  .target.oclAsType(Interface) ->asSet()
in
let cd_ifs: Set(Interface)=
  Package.allInstances() ->select(getAppliedStereotypes().name
      ->includes(’ContextDiagram’)) ->asSequence() ->first()
  .clientDependency.target
  ->select(oclIsTypeOf(Interface)).oclAsType(Interface) ->asSet()
in
let cd_if_parts: Set(Interface)=
  cd_ifs.member
  ->select(oclIsTypeOf(Property)).oclAsType(Property).type
  ->select(oclIsTypeOf(Interface)).oclAsType(Interface) ->asSet()
in
let cd_conn_doms: Set(Class) =
  Class.allInstances() ->select(cl |
    cl.clientDependency
    ->select(
        getAppliedStereotypes().name ->includes(’concretizes’) or
        getAppliedStereotypes().name ->includes(’refines’))
    ->select(target->asSequence() ->first().oclIsTypeOf(Interface))
    ->exists(
      cd_ifs->includes(target.oclAsType(Interface) ->asSequence() ->first()) or
      cd_if_parts->includes(target.oclAsType(Interface) ->asSequence() ->first())
    )
  ) ->asSet()
in
  Package.allInstances() ->select(getAppliedStereotypes().name
      ->includes('ProblemDiagram'))
  ->forAll(pd_tcd |
    pd_tcd.clientDependency -> select(getAppliedStereotypes().name
        ->includes('isPart')).target
    ->select(oclIsTypeOf(Class))
    ->select(
          getAppliedStereotypes().name ->includes('Domain') or
          getAppliedStereotypes().general.name ->includes('Domain') or
          getAppliedStereotypes().general.general.name ->includes('Domain') or
          getAppliedStereotypes().general.general.general.name
              ->includes('Domain')).oclAsType(Class)
    ->forAll(pd_domain |
        cd_conn_doms->includes(pd_domain) or
        let pd_domain_obs_if: Set(Interface) =
          pd_domain.clientDependency
          ->select(getAppliedStereotypes().name ->includes('observes'))
          .target.oclAsType(Interface) ->asSet()
        in
        let pd_domain_concr_if: Set(Interface) =
                pd_domain.clientDependency->select(
                  getAppliedStereotypes().name ->includes('concretizes') or
                  getAppliedStereotypes().name ->includes('refines'))
                .target->select(oclIsTypeOf(Interface)).oclAsType(Interface) ->asSet()
        in
          cd_domains_obs_if->includesAll(pd_domain_obs_if) or
          cd_gene_dom_obs_if->includesAll(pd_domain_obs_if) or
          cd_spec_dom_obs_if->includesAll(pd_domain_obs_if) or
          cd_domain_parts_obs_if->includesAll(pd_domain_obs_if) or
          cd_merged_domains_obs_if->includesAll(pd_domain_obs_if) or
          (
            cd_ifs->includesAll(pd_domain_concr_if) and
            pd_domain_concr_if->notEmpty() and
            pd_domain.clientDependency->exists(
                getAppliedStereotypes().name ->includes('observes') and
                pd_tcd.clientDependency ->select(getAppliedStereotypes().name
                    ->includes('isPart')).target
                ->select(oclIsTypeOf(Interface))
                ->includesAll(target ->select(oclIsTypeOf(Interface))
                    .oclAsType(Interface))
            )
          ) or (
            cd_if_parts->includesAll(pd_domain_concr_if) and
            pd_domain_concr_if->notEmpty() and
            pd_domain.clientDependency->exists(
                getAppliedStereotypes().name ->includes('observes') and
                pd_tcd.clientDependency ->select(getAppliedStereotypes().name
                    ->includes('isPart')).target
                ->select(oclIsTypeOf(Interface))
                ->includesAll(target ->select(oclIsTypeOf(Interface))
                    .oclAsType(Interface))
            )
          )
    )
  )
```

Listing C.36: *Observed interfaces in the problem diagrams are consistent to the observed interfaces in the context diagram*

A problem diagram is consistent to a context diagram, if also each interface observed by domain d in the problem diagram is observed by the domain corresponding to d in the context diagram.

```
let cd_domains: Set(Class) =
   Package.allInstances() ->select(getAppliedStereotypes().name
       ->includes('ContextDiagram')) ->asSequence() ->first()
   .clientDependency.target
   ->select(
  getAppliedStereotypes().name ->includes(’Domain’) or
  getAppliedStereotypes().general.name ->includes(’Domain’) or
  getAppliedStereotypes().general.general.name ->includes(’Domain’) or
  getAppliedStereotypes().general.general.general.name ->includes(’Domain’))
    .oclAsType(Class) ->asSet()
 in
 let cd_domains_contr_if: Set(Interface) =
   cd_domains.clientDependency
   ->select(getAppliedStereotypes().name ->includes(’controls’))
   .target.oclAsType(Interface) ->asSet()
 in
let cd_spec_dom: Set(Class) =
  Class.allInstances()->select(c| cd_domains->exists(cdd| c.general()->includes(cdd)))
in
let cd_spec_dom_contr_if: Set(Interface) =
  cd_domains.clientDependency
  ->select(getAppliedStereotypes().name ->includes(’controls’))
  .target.oclAsType(Interface) ->asSet()
in
let cd_gene_dom_contr_if: Set(Interface) =
  cd_domains.general().clientDependency
  ->select(getAppliedStereotypes().name ->includes(’controls’))
  .target.oclAsType(Interface) ->asSet()
in
let cd_domain_parts: Set(Class) =
   cd_domains.member
   ->select(oclIsTypeOf(Property)).oclAsType(Property).type
   ->select(oclIsTypeOf(Class)).oclAsType(Class) ->asSet()
 in
 let cd_domain_parts_contr_if: Set(Interface) =
   cd_domain_parts.clientDependency
   ->select(getAppliedStereotypes().name ->includes(’controls’))
   .target.oclAsType(Interface) ->asSet()
 in
 let cd_merged_domains: Set(Class) =
   Class.allInstances()
   ->select(cl |
     let cl_members: Set(Class) =
       cl.member
       ->select(oclIsTypeOf(Property)) .oclAsType(Property).type
       ->select(oclIsTypeOf(Class)) .oclAsType(Class) ->asSet()
     in
       cl_members->exists(clm | cd_domains->includes(clm))
   )
 in
 let cd_merged_domains_contr_if: Set(Interface) =
   cd_merged_domains.clientDependency
   ->select(getAppliedStereotypes().name ->includes(’controls’))
   .target.oclAsType(Interface) ->asSet()
 in
 let cd_ifs: Set(Interface)=
   Package.allInstances() ->select(getAppliedStereotypes().name
       ->includes(’ContextDiagram’)) ->asSequence() ->first()
   .clientDependency.target
   ->select(oclIsTypeOf(Interface)).oclAsType(Interface) ->asSet()
 in
 let cd_if_parts: Set(Interface)=
   cd_ifs.member
   ->select(oclIsTypeOf(Property)).oclAsType(Property).type
   ->select(oclIsTypeOf(Interface)).oclAsType(Interface) ->asSet()
 in
 let cd_conn_doms: Set(Class) =
   Class.allInstances() ->select(cl |
     cl.clientDependency
     ->select(
  getAppliedStereotypes().name ->includes(’concretizes’) or
  getAppliedStereotypes().name ->includes(’refines’))
     ->select(target->asSequence() ->first().oclIsTypeOf(Interface))
     ->exists(
       cd_ifs->includes(target.oclAsType(Interface) ->asSequence() ->first()) or
       cd_if_parts->includes(target.oclAsType(Interface) ->asSequence() ->first())
     )
   ) ->asSet()
 in
   Package.allInstances() ->select(getAppliedStereotypes().name
       ->includes(’ProblemDiagram’))
   ->forAll(pd_tcd |
     pd_tcd.clientDependency -> select(getAppliedStereotypes().name
         ->includes(’isPart’)).target
     ->select(oclIsTypeOf(Class))
     ->select(
    getAppliedStereotypes().name ->includes(’Domain’) or
    getAppliedStereotypes().general.name ->includes(’Domain’) or
    getAppliedStereotypes().general.general.name ->includes(’Domain’) or
    getAppliedStereotypes().general.general.general.name
        ->includes(’Domain’)).oclAsType(Class)
     ->forAll(pd_domain |
         cd_conn_doms->includes(pd_domain) or
  let pd_domain_contr_if: Set(Interface) =
    pd_domain.clientDependency
    ->select(getAppliedStereotypes().name ->includes(’controls’))
    .target.oclAsType(Interface) ->asSet()
  in
  let pd_domain_concr_if: Set(Interface) =
                pd_domain.clientDependency->select(
                  getAppliedStereotypes().name ->includes(’concretizes’) or
                  getAppliedStereotypes().name ->includes(’refines’))
                .target->select(oclIsTypeOf(Interface)).oclAsType(Interface) ->asSet()
  in
    cd_domains_contr_if->includesAll(pd_domain_contr_if) or
    cd_gene_dom_contr_if->includesAll(pd_domain_contr_if) or
    cd_spec_dom_contr_if->includesAll(pd_domain_contr_if) or
    cd_domain_parts_contr_if->includesAll(pd_domain_contr_if) or
    cd_merged_domains_contr_if->includesAll(pd_domain_contr_if) or
    (
      cd_ifs->includesAll(pd_domain_concr_if) and
      pd_domain_concr_if->notEmpty() and
      pd_domain.clientDependency->exists(
   getAppliedStereotypes().name ->includes(’controls’) and
   pd_tcd.clientDependency ->select(getAppliedStereotypes().name
       ->includes(’isPart’)).target
   ->select(oclIsTypeOf(Interface))
   ->includesAll(target ->select(oclIsTypeOf(Interface)) .oclAsType(Interface))
      )
    ) or (
      cd_if_parts->includesAll(pd_domain_concr_if) and
      pd_domain_concr_if->notEmpty() and
      pd_domain.clientDependency->exists(
   getAppliedStereotypes().name ->includes(’controls’) and
   pd_tcd.clientDependency ->select(getAppliedStereotypes().name
       ->includes(’isPart’)).target
   ->select(oclIsTypeOf(Interface))
   ->includesAll(target ->select(oclIsTypeOf(Interface)) .oclAsType(Interface))
      )
    )
     )
   )
```

Listing C.37: *Controlled interfaces in the problem diagrams are consistent to the controlled interfaces in the context diagram*

A problem diagram is consistent to a context diagram, if also each interface controlled by domain d in the problem diagram is controlled by the domain corresponding to d in the context diagram.

```
let pd_domains: Set(Class) =
  Package.allInstances() ->select(getAppliedStereotypes().name
      ->includes(’ProblemDiagram’))
.clientDependency.target
  ->select(
        getAppliedStereotypes().name ->includes(’Domain’) or
        getAppliedStereotypes().general.name ->includes(’Domain’) or
```

```
 7         getAppliedStereotypes().general.general.name ->includes(’Domain’) or
 8         getAppliedStereotypes().general.general.general.name ->includes(’Domain’))
 9    .oclAsType(Class) ->asSet()
10 in
11 let pd_spec_dom: Set(Class) =
12   Class.allInstances()->select(c| pd_domains->exists(pdd| c.general()->includes(pdd)))
13 in
14 let pd_domain_parts: Set(Class) =
15   pd_domains.member
16   ->select(oclIsTypeOf(Property) and
       oclAsType(Property).type.oclIsTypeOf(Class)).oclAsType(Property).type
17   .oclAsType(Class) ->asSet()
18 in
19 let connection_domains: Set(Class) =
20   Class.allInstances() ->select(
21           getAppliedStereotypes().name ->includes(’Domain’) or
22           getAppliedStereotypes().general.name ->includes(’Domain’) or
23           getAppliedStereotypes().general.general.name ->includes(’Domain’) or
24           getAppliedStereotypes().general.general.general.name ->includes(’Domain’))
25   ->select(
26           clientDependency.getAppliedStereotypes().name ->includes(’refines’)  or
27           clientDependency.getAppliedStereotypes().name ->includes(’concretizes’) )
28   ->select(clientDependency->exists(target->forAll(oclIsTypeOf(Interface))))
29   ->asSet()
30 in
31   Package.allInstances() ->select(getAppliedStereotypes().name
       ->includes(’ContextDiagram’) ) ->asSequence() ->first()
32   ->forAll(cd  |
33     cd.ownedElement->select(oclIsTypeOf(Association)).oclAsType(Association)
34     ->select(a | a.endType->exists(et|
         et->select(oclIsTypeOf(Class)).oclAsType(Class).getAppliedStereotypes().name
         ->includes(’Machine’)))
35     .endType ->select(oclIsTypeOf(Class))
36     ->select(
37           getAppliedStereotypes().name ->includes(’Domain’) or
38           getAppliedStereotypes().general.name ->includes(’Domain’) or
39           getAppliedStereotypes().general.general.name ->includes(’Domain’) or
40           getAppliedStereotypes().general.general.general.name
               ->includes(’Domain’)).oclAsType(Class)
41     ->forAll(cd_dom |
42         pd_domains->includes(cd_dom) or
43         pd_domains.general()->includes(cd_dom) or
44         pd_spec_dom->includes(cd_dom) or
45         pd_domain_parts->includes(cd_dom) or
46         let cd_dom_parts: Set(Class) =
47             cd_dom.member
48             ->select(oclIsTypeOf(Property) and
                 oclAsType(Property).type.oclIsTypeOf(Class)).oclAsType(Property).type
49             ->select(oclIsTypeOf(Class))
50             ->select(
51                   getAppliedStereotypes().name ->includes(’Domain’) or
52                   getAppliedStereotypes().general.name ->includes(’Domain’) or
53                   getAppliedStereotypes().general.general.name ->includes(’Domain’) or
54                   getAppliedStereotypes().general.general.general.name
                       ->includes(’Domain’)).oclAsType(Class) ->asSet()
55         in
56           (cd_dom_parts->notEmpty() and
57           pd_domains->includesAll(cd_dom_parts))
58         or
59         connection_domains->includes(cd_dom))  )
```

Listing C.38: *Domains in the context diagram connected to the machine must be found in problem diagrams*

All domains in the context diagram must be (using the decomposition operators) in at least one problem diagram:

- as the domain itself (line 42),
- as generalized or specialized domains (lines 43 and 44),

- as a merged domain (line 45),
- as a split domain (lines 46-57), or
- as an interface concretizing a connection domain (line 59, operator: remove connection domain).

C.5. Constraints related to the consistency between problem diagrams and problem frames

```
 1 Dependency.allInstances() ->select(a |
 2 a.oclAsType(Dependency).getAppliedStereotypes().name ->includes('instanceOf') )
       ->forAll(d |
 3   d.oclAsType(Dependency).source ->forAll(oclIsTypeOf(Package)) and
 4   d.oclAsType(Dependency).source.getAppliedStereotypes().name
         ->includes('ProblemDiagram') and
 5   d.oclAsType(Dependency).target ->forAll(oclIsTypeOf(Package)) and
 6   d.oclAsType(Dependency).target.getAppliedStereotypes().name
         ->includes('ProblemFrame')
 7 )
```

Listing C.39: *The stereotype ≪instanceOf≫ points from ≪ProblemDiagram≫ to ≪ProblemFrame≫*

A dependency with the stereotype ≪instanceOf≫ (lines 1-2) only points from a problem diagram (lines 3-4) to a problem frame (lines 5-6).

```
 1 Dependency.allInstances() -> select(getAppliedStereotypes().name
       ->includes('instanceOf') ) ->forAll(
 2   target.oclAsType(Package)
 3   .clientDependency ->select(getAppliedStereotypes().name ->includes('isPart')).target
 4   ->select(getAppliedStereotypes().name ->includes('Requirement')).oclAsType(Class)
 5   .clientDependency ->select(getAppliedStereotypes().name ->includes('constrains')).
 6   target.getAppliedStereotypes().name
 7   ->reject(n | n='ConnectionDomain' or n='DesignedDomain')
 8   =
 9   source.oclAsType(Package)
10   .clientDependency -> select(getAppliedStereotypes().name ->includes('isPart'))
11   .target -> select(getAppliedStereotypes().name
         ->includes('Requirement')).oclAsType(Class)
12   .clientDependency -> select(getAppliedStereotypes().name ->includes('constrains')).
13   target.getAppliedStereotypes().name
14   ->reject(n | n='ConnectionDomain' or n='DesignedDomain')
15 )
```

Listing C.40: *Domain types of constrained domains in problem frame are the same as in instantiated frame*

The domain types of constrained domains in a problem frame (lines 2-7) are the same types as in the instantiated frame (lines 7-14).

```
 1 Dependency.allInstances() -> select(getAppliedStereotypes().name
       ->includes('instanceOf') ) ->forAll(
 2   source.oclAsType(Package)
 3   .clientDependency -> select(getAppliedStereotypes().name ->includes('isPart'))
 4   .target -> select(getAppliedStereotypes().name
         ->includes('Requirement')).oclAsType(Class)
 5   .clientDependency ->select(getAppliedStereotypes().name ->includes('refersTo'))
 6   .target.getAppliedStereotypes().name
 7   ->includesAll(
 8     target.oclAsType(Package)
 9     .clientDependency -> select(getAppliedStereotypes().name ->includes('isPart'))
10     .target -> select(getAppliedStereotypes().name
           ->includes('Requirement')).oclAsType(Class)
11     .clientDependency -> select(getAppliedStereotypes().name ->includes('refersTo'))
12     .target.getAppliedStereotypes().name
13      ->reject(n | n=’ConnectionDomain’ or n=’DesignedDomain’)
14    )
15  )
```

Listing C.41: *A referred domain in the problem frame corresponds to a domain in the problem diagram*

The referred domains in the problem frame (lines 8-13) correspond to referred domains in the problem diagram (lines 2-6).

```
1   Dependency.allInstances() -> select(getAppliedStereotypes().name
        ->includes(’instanceOf’) )
2   ->forAll(
3     target.oclAsType(Package)
4     .clientDependency -> select(getAppliedStereotypes().name ->includes(’isPart’))
5     .target -> select(getAppliedStereotypes().name
          ->includes(’Requirement’)).oclAsType(Class)
6     .clientDependency ->select(getAppliedStereotypes().name ->includes(’constrains’)).
7     target.getAppliedStereotypes().name
8   =
9     source.oclAsType(Package)
10    .clientDependency -> select(getAppliedStereotypes().name ->includes(’isPart’))
11    .target -> select(getAppliedStereotypes().name
          ->includes(’Requirement’)).oclAsType(Class)
12    .clientDependency -> select(getAppliedStereotypes().name ->includes(’constrains’)).
13    target.getAppliedStereotypes().name
14  )
```

Listing C.42: *A constrained domain in the problem frame corresponds to a domain in the problem diagram*

The domain types of the constrained domains in the problem frame are the same as in the problem diagram.

All dependencies in the model with the stereotype ≪instanceOf≫ (line 1) are selected. For these dependencies (line 2) the parts of the target (i.e, the problem frame) being requirements (lines 4 and 5) are selected. For these requirements, the dependencies with the stereotype ≪constrains≫ are selected (line 6). The target of these dependencies are the constrained classes, and the bag of their stereotype names (line 7) must be the same (line 8) as the bag of stereotype names of constrained domains in the problem diagram (lines 9-13).

```
1   Dependency.allInstances() -> select(getAppliedStereotypes().name
        ->includes(’instanceOf’) )
2   ->forAll(inst_of_dep |
3     Association.allInstances()
4     ->select(endType->forAll(oclIsTypeOf(Class))).oclAsType(Association)
5     ->select(ass |
6       ass.oclAsType(Association).endType->forAll (ass_end |
7         inst_of_dep.oclAsType(Dependency)
8         .target.oclAsType(Package)
9         .clientDependency -> select(getAppliedStereotypes().name ->includes(’isPart’))
10        .target->select(oclIsTypeOf(Class)).oclAsType(Class) ->asSet()
11        ->includes(ass_end.oclAsType(Class))
12      )
13    )->forAll(ass_in_pf |
14      Association.allInstances()
15      ->select(endType->forAll(oclIsTypeOf(Class))).oclAsType(Association) ->select(ass
            |
16        ass.oclAsType(Association).endType->forAll (ass_end |
              inst_of_dep.oclAsType(Dependency)
17          .source.oclAsType(Package)
18          .clientDependency -> select(getAppliedStereotypes().name ->includes(’isPart’))
19          .target->select(oclIsTypeOf(Class)).oclAsType(Class) ->asSet()
20          ->includes(ass_end.oclAsType(Class)))
21      ) ->exists(ass_in_pd |
22        ass_in_pd.endType.oclAsType(Class).getAppliedStereotypes().name ->includesAll(
23          ass_in_pf.endType.oclAsType(Class).getAppliedStereotypes().name
24        )
25      )
26    )
27  )
```

Listing C.43: *All connections in problem frames correspond to connections in a problem diagram (being an instance of that frame) (i.e., the connection connects same domain types)*

All connections in a problem frame (lines 1-13) correspond to connections in the problem diagram being an instance of that frame (lines 14-23), i.e., a connection connects same domain types.

```
1  Dependency.allInstances() ->
       select(getAppliedStereotypes().name->includes('instanceOf') )
2  ->forAll(inst_of_dep |
3    not
4      inst_of_dep.getValue(inst_of_dep.oclAsType(Dependency) .getAppliedStereotypes()
           ->select(name->includes('instanceOf'))
           ->asSequence()->first(),'weak').oclAsType(Boolean)
5    implies
6    Association.allInstances()
7    ->select(endType->forAll(oclIsTypeOf(Class))) .oclAsType(Association)
8    ->select(ass |
9    ass.oclAsType(Association).endType ->forAll (ass_end |
         inst_of_dep.oclAsType(Dependency)
10     .source.oclAsType(Package)
11     .clientDependency -> select(getAppliedStereotypes().name->includes('isPart'))
12     .target->select(oclIsTypeOf(Class)).oclAsType(Class)->asSet()
13     ->includes(ass_end.oclAsType(Class)))
14   )->forAll(ass_in_pd |
15   Association.allInstances()
16   ->select(endType->forAll(oclIsTypeOf(Class))) .oclAsType(Association)
17   ->select(ass |
18   ass.oclAsType(Association).endType ->forAll (ass_end |
         inst_of_dep.oclAsType(Dependency)
19     .target.oclAsType(Package)
20     .clientDependency -> select(getAppliedStereotypes().name->includes('isPart'))
21     .target->select(oclIsTypeOf(Class)).oclAsType(Class)->asSet()
22     ->includes(ass_end.oclAsType(Class)))
23   )->exists(ass_in_pf |
24       ass_in_pd.endType.oclAsType(Class) .getAppliedStereotypes().name
25       ->includesAll (
26         ass_in_pf.endType.oclAsType(Class) .getAppliedStereotypes().name
27         )
28     )
29   )
30 )
```

Listing C.44: *If not weak: All connections in problem diagrams correspond to connections in the instantiated problem frame (connect same domain types) - No additional connections exist)*

If the dependency between problem diagram and problem frame is not weak (lines 3-5) then all connections in the problem diagram (lines 6-14) correspond to connections in the problem frame that is instantiated (connect same domain types, lines 15-28) - no additional connections are allowed.

```
1  Dependency.allInstances() ->
       select(getAppliedStereotypes().name->includes('instanceOf') )
2  ->forAll( inst_of_dep |
3    let pf_domains: Bag(Class) =
4      inst_of_dep.target.oclAsType(Package)
5      .clientDependency -> select(getAppliedStereotypes().name -> includes('isPart'))
6      .target -> select(oclIsTypeOf(Class)) -> reject(getAppliedStereotypes().name ->
           includes('Requirement')).oclAsType(Class)
7    in
8    let pd_domains: Bag(Class) =
9      inst_of_dep.source.oclAsType(Package)
10     .clientDependency -> select(getAppliedStereotypes().name -> includes('isPart'))
11     .target -> select(oclIsTypeOf(Class)) -> reject(getAppliedStereotypes().name ->
           includes('Requirement')).oclAsType(Class)
12   in
13   let pf_ifs: Set(Interface) =
14     inst_of_dep.target.oclAsType(Package)
15     .clientDependency -> select(getAppliedStereotypes().name -> includes('isPart'))
16     .target -> select(oclIsTypeOf(Interface)).oclAsType(Interface) -> asSet()
17   in
18   let pd_ifs: Set(Interface) =
19     inst_of_dep.source.oclAsType(Package)
20     .clientDependency -> select(getAppliedStereotypes().name -> includes('isPart'))
21     .target -> select(oclIsTypeOf(Interface)).oclAsType(Interface) -> asSet()
22   in
23     pf_domains->select(getAppliedStereotypes().name -> includes('Machine'))
24     .clientDependency -> select(getAppliedStereotypes().name -> includes('controls'))
25     -> select(pf_ifs->includesAll(target.oclAsType(Interface))) -> asSet() ->size ()
26     =
27     pd_domains->select(getAppliedStereotypes().name -> includes('Machine'))
28     .clientDependency -> select(getAppliedStereotypes().name -> includes('controls'))
29     -> select(pd_ifs->includesAll(target.oclAsType(Interface))) -> asSet() ->size ()
30 )
```

Listing C.45: *Interfaces cannot be left out if they are controlled by the machine.*

Interfaces cannot be left out if they are controlled by the machine. This is expressed in the same way as in Expression C.42. Using the same definitions, we check that the number of controlled interfaces of each domain in each problem frame with the stereotype ≪machine≫ (lines 23-25) is equal to (line 26) the number of controlled interfaces of each domain in each problem diagram with the stereotype ≪machine≫ (lines 27-29).

C.6. Constraints related to the consistency between problem diagrams and sequence diagrams

```
Interaction.allInstances()->size()>0 implies
Package.allInstances()->select(getAppliedStereotypes().name
    ->includes('ProblemDiagram')) ->forAll( pd |
  Interaction.allInstances()
  ->exists(pd.oclAsType(Package).clientDependency.target ->select(oclIsTypeOf(Class))
        ->select(getAppliedStereotypes().name ->includes('Domain') or
                getAppliedStereotypes().general.name ->includes('Domain') or
                getAppliedStereotypes().general.general.name ->includes('Domain') or
                getAppliedStereotypes().general.general.general.name
                    ->includes('Domain'))
        .oclAsType(Class).name->includesAll(lifeline.name )
        and
        pd.oclAsType(Package).clientDependency.target ->select(oclIsTypeOf(Class))
        ->select(d | clientDependency.target
            ->select(oclIsTypeOf(Association)).oclAsType(Association)
        ->exists(endType->includes(d.oclAsType(Type)) and
            endType.getAppliedStereotypes().name->includes('Machine')))
        ->forAll(d | lifeline.name ->includes(d.oclAsType(Class).name))
  ) )
```

Listing C.46: *Problem diagram domains vs. lifelines in sequence diagrams*

This expression checks whether there exist sequence diagrams for all problem diagrams. If at least one sequence diagram exists (line 1), we have to check the following: For all packages with the stereotype ≪ProblemDiagram≫ (pd, line 2), check the set of all sequence diagrams (keyword *Interaction*) (line 3) whether there exists a sequence diagram where the names of the lifelines (line 9) are a subset of the names of the package elements (line 4) with the stereotype domain or a sub-type of domain (lines 5-8 as well as line 10). Additionally, all domains connected with the machine (lines 11-13) have to be represented as lifelines (line 14).

```
 1 Interaction.allInstances() ->forAll( sd |
 2   Package.allInstances()
         ->select(getAppliedStereotypes().name->includes(’ProblemDiagram’))
 3   ->exists(clientDependency.target ->select(oclIsTypeOf(Class))
 4     ->select(getAppliedStereotypes().name ->includes(’Domain’) or
 5       getAppliedStereotypes().general.name ->includes(’Domain’) or
 6       getAppliedStereotypes().general.general.name ->includes(’Domain’) or
 7       getAppliedStereotypes().general.general.general.name ->includes(’Domain’))
 8       .oclAsType(Class).name ->includesAll(sd.oclAsType(Interaction).lifeline.name)
 9   )
10   or
11   ( sd.oclAsType(Interaction).lifeline.name->forAll(ln|
12         Class.allInstances()->exists(c |
13           let names_of_included_classes:Set(String) =
14             c.member->select(oclIsTypeOf(Property)).oclAsType(Property).type
                 ->select(oclIsTypeOf(Class)).oclAsType(Class).name->asSet()
15           in
16           let ln_ss:Sequence(String) = Sequence{1..ln.size()}
               ->collect(i|ln.substring(i,i))
17           in
18           let class_name: String =
19             if ln_ss->indexOf(’:’) = null
20             then ln
21             else Sequence{(ln_ss->indexOf(’:’) + 1)..ln_ss->size()} ->iterate(i;
                 res:String=’’| res.concat(ln_ss->at(i)))
22             endif
23           in
24             names_of_included_classes->includes(class_name)
25         )
26         or ln = ’ENVIRONMENT’
27     )
28   )
29 )
```

Listing C.47: *Lifelines in sequence diagrams vs. problem diagram domains*

For all sequence diagrams (line 1), we check the set of all packages with the stereotype ≪ProblemDiagram≫ (line 2) whether there exists (line 3) a problem diagram where the names of the sequence diagram lifelines (line 8) are a subset of the names of the package elements (line 4) with the stereotype domain or a sub-type of domain (lines 4-8) or (line 10). To allow sequence diagrams that describe the behavior of components (needed in the design phase), we check that the set of names of the classes that are part of another class (variable names_of_included_classes) includes the set of lifeline names (variable class_name) (line 24) or that the lifeline name is 'ENVIRONMENT' (line 26). In order to be able to compare the names, we must extract the names of the lifelines that describe objects (variable class_name, lines 16-23) beforehand.

```
 1 let mchns: Set(Class) =
 2   Package.allInstances()->select(getAppliedStereotypes().name
       ->includes(’ProblemDiagram’))
 3   .clientDependency ->select(getAppliedStereotypes().name ->includes(’isPart’))
 4   .target ->select(getAppliedStereotypes().name ->includes(’Machine’) or
 5                   getAppliedStereotypes().general.name ->includes(’Machine’))
```

```
  .oclAsType(Class)->asSet()
in let doms: Set(Class) =
  Package.allInstances()->select(getAppliedStereotypes().name
      ->includes('ProblemDiagram'))
  .clientDependency ->select(getAppliedStereotypes().name ->includes('isPart'))
  .target ->select(getAppliedStereotypes().name ->includes('DisplayDomain') or
                  getAppliedStereotypes().general.name ->includes('DisplayDomain') or
                  getAppliedStereotypes().name ->includes('ConnectionDomain') or
                  getAppliedStereotypes().general.name ->includes('ConnectionDomain'))
  ->select(cddd |
    Association.allInstances()
    ->exists(as |
        as.oclAsType(Association).endType ->includes(cddd.oclAsType(Class)) and
        as.oclAsType(Association).endType ->exists(et | mchns
            ->includes(et.oclAsType(Class)))
    )
  )
  ->union(mchns)
  .oclAsType(Class)->asSet()
in
  doms.clientDependency ->select(getAppliedStereotypes().name
      ->includes('controls')).target
        ->intersection(Package.allInstances()->select(getAppliedStereotypes().name
            ->includes('ProblemDiagram'))
                .clientDependency ->select(getAppliedStereotypes().name
                    ->includes('isPart'))
                .target ->select(oclIsTypeOf(Interface)))
        .ownedElement
        ->select(oclIsTypeOf(Operation)).oclAsType(Operation).name ->asSet()
        ->reject(op |
          Class.allInstances()->select(getAppliedStereotypes().name
              ->includes('LexicalDomain'))
          .clientDependency ->select(getAppliedStereotypes().name
              ->includes('controls')).target.oclAsType(Interface).ownedElement
          ->select(oclIsTypeOf(Operation)).oclAsType(Operation).name
          ->includes( op )
        )
        ->forAll( phen |
                Lifeline.allInstances()
                .coveredBy
                ->select(oclIsTypeOf(MessageOccurrenceSpecification) and
                    name.substring(1,1)='S')
                .oclAsType(MessageOccurrenceSpecification).message->select(m | m <>
                    null).name
                ->includes(phen)
        )

and
let lexrmsgs: Set(String) =
  Lifeline.allInstances()
  ->select(ln | Class.allInstances()->select(getAppliedStereotypes().name
      ->includes('LexicalDomain'))->exists(name=ln.name))
  .coveredBy
  ->select(oclIsTypeOf(MessageOccurrenceSpecification) and name.substring(1,1)='R')
  .oclAsType(MessageOccurrenceSpecification).message.name
  .oclAsType(String)->asSet()
in
let contrphen: Set(String) =
   doms.clientDependency ->select(getAppliedStereotypes().name
       ->includes('controls')).target
             ->intersection(Package.allInstances()->select(getAppliedStereotypes().name
                 ->includes('ProblemDiagram'))

            .clientDependency ->select(getAppliedStereotypes().name
                ->includes('isPart'))
        .target ->select(oclIsTypeOf(Interface))).ownedElement
        ->select(oclIsTypeOf(Operation)).oclAsType(Operation).name
       .oclAsType(String)->asSet()
in
  Lifeline.allInstances()
```

```
63    ->reject(ln | Class.allInstances()->select(getAppliedStereotypes().name
          ->includes('LexicalDomain'))->exists(name=ln.name))
64    ->select(ln | doms ->exists(name=ln.name))
65    .coveredBy
66    ->select(oclIsTypeOf(MessageOccurrenceSpecification) and name.substring(1,1)='S')
67    .oclAsType(MessageOccurrenceSpecification).message.name
68    ->reject(n | lexrmsgs ->includes(n))
69    ->select(n | n<>'')
70    ->forAll(msg | contrphen ->includes(msg)  )
```

Listing C.48: *Sent messages in sequence diagrams vs. operations in controlled interfaces in problem diagrams*

This expression checks that all relevant controlled phenomena in the problem diagrams related to the machine are sent messages in the sequence diagrams (lines 24-42). By relevant we mean all phenomena (interface operations) controlled by `doms` (line 25) found in a problem diagram (lines 26-28), which are not controlled by a lexical domain (lines 29-36).

The expression also checks that the relevant sent messages in the sequence diagrams are controlled phenomena in the problem diagram (lines 62-70). All messages sent by lifelines that correspond to an element of `doms` (lines 62-66) and which are neither in the set of messages observed by lexical domains (variable `lexrmsgs`) nor are empty are considered as being relevant.

In order to check this, we set

- variable `mchns` to the set of machines found in a problem diagram (lines 1-7).
- variable `doms` to the set of connected classes with the stereotype ≪DisplayDomain≫, ≪ConnectionDomain≫, or any of the sub-types of the aforementioned stereotypes (lines 8-23).
- variable `lexrmsgs` to the set of all messages observed by lexical domains (lines 45-51).
- variable `contrphen` to the set of all operations in interfaces controlled by `doms` (line 54) and being part of a problem diagram (lines 55-57).

```
1  let mchns: Set(Class) =
2    Package.allInstances()->select(getAppliedStereotypes().name
         ->includes('ProblemDiagram'))
3    .clientDependency ->select(getAppliedStereotypes().name ->includes('isPart'))
4    .target ->select(getAppliedStereotypes().name ->includes('Machine') or
5                     getAppliedStereotypes().general.name ->includes('Machine'))
6    .oclAsType(Class)->asSet()
7  in
8  let doms: Set(Class) =
9    Package.allInstances()->select(getAppliedStereotypes().name
         ->includes('ProblemDiagram'))
10   .clientDependency ->select(getAppliedStereotypes().name ->includes('isPart'))
11   .target ->select(getAppliedStereotypes().name ->includes('DisplayDomain') or
12                    getAppliedStereotypes().general.name ->includes('DisplayDomain') or
13                    getAppliedStereotypes().name ->includes('ConnectionDomain') or
14                    getAppliedStereotypes().general.name ->includes('ConnectionDomain'))
15   ->select(cddd |
16     Association.allInstances()
17     ->exists(as |
18         as.oclAsType(Association).endType ->includes(cddd.oclAsType(Class)) and
19         as.oclAsType(Association).endType ->exists(et | mchns
               ->includes(et.oclAsType(Class)))
20     )
21   )
22   ->union(mchns)
23   .oclAsType(Class)->asSet()
24 in
25   doms.clientDependency ->select(getAppliedStereotypes().name
         ->includes('observes')).target
26         ->intersection(Package.allInstances()->select(getAppliedStereotypes().name
               ->includes('ProblemDiagram'))
27                 .clientDependency ->select(getAppliedStereotypes().name
                       ->includes('isPart'))
28                 .target ->select(oclIsTypeOf(Interface)))
29         .ownedElement
30         ->select(oclIsTypeOf(Operation)).oclAsType(Operation).name ->asSet()
31         ->reject(op |
32           Class.allInstances()->select(getAppliedStereotypes().name
                 ->includes('LexicalDomain'))
33           .clientDependency ->select(getAppliedStereotypes().name
                 ->includes('controls')).target.oclAsType(Interface).ownedElement
34           ->select(oclIsTypeOf(Operation)).oclAsType(Operation).name
35           ->includes( op )
36         )
37         ->forAll( phen |
38                 Lifeline.allInstances()
39                 .coveredBy
40                 ->select(oclIsTypeOf(MessageOccurrenceSpecification) and
                       name.substring(1,1)='R')
41                 .oclAsType(MessageOccurrenceSpecification).message.name
42                 ->includes(phen)
43         )
44 and
45 let lexsmsgs: Set(String) =
46   Lifeline.allInstances()
47   ->select(ln | Class.allInstances()->select(getAppliedStereotypes().name
         ->includes('LexicalDomain'))->exists(name=ln.name))
48   .coveredBy
49   ->select(oclIsTypeOf(MessageOccurrenceSpecification) and name.substring(1,1)='S')
50   .oclAsType(MessageOccurrenceSpecification).message.name
51   .oclAsType(String)->asSet()
52 in
53 let obsphen: Set(String) =
54    doms.clientDependency ->select(getAppliedStereotypes().name
        ->includes('observes')).target
55              ->intersection(Package.allInstances()->select(getAppliedStereotypes().name
                  ->includes('ProblemDiagram'))
56                 .clientDependency ->select(getAppliedStereotypes().name
                       ->includes('isPart'))
57                 .target ->select(oclIsTypeOf(Interface)))
58             .ownedElement
59         ->select(oclIsTypeOf(Operation)).oclAsType(Operation).name
60        .oclAsType(String)->asSet()
61 in
62   Lifeline.allInstances()
63   ->reject(ln | Class.allInstances()->select(getAppliedStereotypes().name
         ->includes('LexicalDomain'))->exists(name=ln.name))
64   ->select(ln | doms ->exists(name=ln.name))
65   .coveredBy
66   ->select(oclIsTypeOf(MessageOccurrenceSpecification) and name.substring(1,1)='R')
67   .oclAsType(MessageOccurrenceSpecification).message.name
68   ->reject(n | lexsmsgs ->includes(n))
69   ->select(n | n<>'')
70   ->forAll(msg | obsphen ->includes(msg)  )
```

Listing C.49: *Received messages in sequence diagrams vs. operations in observed interfaces in problem diagrams*

This expression checks that all relevant observed phenomena in the problem diagrams related to the machine are received messages in the sequence diagrams (lines 25-42). By relevant we mean all phenomena (interface operations) observed by `doms` (line 25) found in a problem diagram (lines 26-28), which are not controlled by a lexical domain (lines 29-36).

The expression also checks that the relevant sent messages in the sequence diagrams are controlled phenomena in the problem diagram (lines 62-70). All messages sent by lifelines that correspond to an element of `doms` (lines 62-66) and which are neither in the set of messages observed by lexical domains (variable `lexsmsgs`) nor are empty are considered as being relevant.

In order to check this, we set

- variable `mchns` to the set of machines found in a problem diagram (lines 1-7).

- variable `doms` to the set of connected classes with the stereotype ≪DisplayDomain≫, ≪ConnectionDomain≫, or any of the sub-types of the aforementioned stereotypes (lines 8-23).
- variable `lexsmsgs` to the set of all messages controlled by lexical domains (lines 45-51).
- variable `obphen` to the set of all operations in interfaces observed by `doms` (line 54) and being part of a problem diagram (lines 55-57).

C.7. Constraints related to technical context diagrams

```
1  Package.allInstances() ->select(p |  p.oclAsType(Package).getAppliedStereotypes()
       .name
2          ->includes('TechnicalContextDiagram')).ownedElement
3              ->forAll(oe |
4                  (oe.oclIsTypeOf(Class) and
5                    (oe.oclAsType(Class).getAppliedStereotypes().name
                         ->includes('Domain') or
6                    oe.oclAsType(Class).getAppliedStereotypes().general.name
                         ->includes('Domain') or
7                    oe.oclAsType(Class).getAppliedStereotypes().general.general.name
                         ->includes('Domain') or
8                    oe.oclAsType(Class).getAppliedStereotypes().general.general
9                                      .general.name ->includes('Domain') ) or
10                 oe.oclIsTypeOf(Interface) or
11                 (oe.oclIsTypeOf(Association) and
12                   (oe.oclAsType(Association).getAppliedStereotypes().name
                         ->includes('connection') or
13                   oe.oclAsType(Association).getAppliedStereotypes().general.name
                         ->includes('connection') or
14                   oe.oclAsType(Association).getAppliedStereotypes().general.general
15                                       .name ->includes('connection') or
16                   oe.oclAsType(Association).getAppliedStereotypes().general.general
17                                       .general.name ->includes('connection') ) ) or
18                 (oe.oclIsTypeOf(Dependency) and
19                   (oe.oclAsType(Dependency).getAppliedStereotypes().name
                         ->includes('controls') or
20                   oe.oclAsType(Dependency).getAppliedStereotypes().name
                         ->includes('observes') or
21                   oe.oclAsType(Dependency).getAppliedStereotypes().name
                         ->includes('isPart'))
22                 )or
23                 oe.oclIsTypeOf(Comment)
24         )
25 )
```

Listing C.50: *Allowed elements for a technical context diagram*

First, we select the package that is annotated with the stereotype ≪TechnicalContextDiagram≫ (lines 1-2) and all the elements associated to it (keyword *ownedElement*, line 2). Second, we check for each owned element *oe* (line 3) if it is a class with the stereotype ≪Domain≫ or any of its subtypes (lines 4 - 8), or if it is an interface (line 9; no restrictions considering the stereotypes apply here), or an association. An association must have the stereotype ≪connection≫ or a subtype of ≪connection≫, e.g., ≪ui≫ for a user interface (lines 10-17), or if it is a dependency (line 18). In this case, it must have either ≪controls≫, ≪observes≫, or ≪isPart≫, or as stereotype, or if it is a comment (line 23).

```
Package.allInstances() ->select(p |
  p.oclAsType(Package).getAppliedStereotypes().name
      ->includes('TechnicalContextDiagram')
) -> forAll (p |
  p.clientDependency ->select(getAppliedStereotypes().name ->includes('isPart'))
  .target->select(cd_elem |
    cd_elem .oclIsTypeOf(Class) and cd_elem
        .oclAsType(Class).getAppliedStereotypes().name ->includes('Machine')
7   ) ->size()>=1
8 )
```

Listing C.51: *A technical context diagram has at least one machine domain*

A technical context diagram must contain at least one machine: We first select all packages with the appropriate stereotype, i.e, ≪TechnicalContextDiagram≫ (lines 1 and 2). For this package we collect all dependencies (keyword clientDependency, line 4) and select those with the stereotype ≪isPart≫ (line 4). Using the target ends of these dependencies, we collect all elements of the package and select (line 5) those (variable cd_elem; line 6) being classes with the stereotype ≪Machine≫ (line 6). The size of the resulting bag must be greater than or equal to one (line 7).

C.8. Constraints related to the consistency between the context diagram and technical context diagrams

```
let cd_domains: Set(Class) =
  Package.allInstances() ->select(getAppliedStereotypes().name
      ->includes('ContextDiagram') or getAppliedStereotypes().name
      ->includes('ProblemDiagram'))
  .clientDependency.target
  ->select(
        getAppliedStereotypes().name ->includes('Domain') or
        getAppliedStereotypes().general.name ->includes('Domain') or
        getAppliedStereotypes().general.general.name ->includes('Domain') or
        getAppliedStereotypes().general.general.general.name ->includes('Domain'))
   .oclAsType(Class) ->asSet()
in
let cd_domain_parts: Set(Class) =
  cd_domains.member
  ->select(oclIsTypeOf(Property)).oclAsType(Property).type
  ->select(oclIsTypeOf(Class)).oclAsType(Class) ->asSet()
in
let cd_dom_concr: Set(Class) =
    Class.allInstances()->select(clientDependency->select(cdep|
        cdep.getAppliedStereotypes().name->includes('concretizes') and
        cdep.target->select(oclIsTypeOf(Class)).oclAsType(Class)
    ->exists(d| cd_domains->includes(d)))->size()>=1)
in
let cd_merged_domains: Set(Class) =
  Class.allInstances()
  ->select(cl |
    let cl_members: Set(Class) =
      cl.member
      ->select(oclIsTypeOf(Property)) .oclAsType(Property).type
      ->select(oclIsTypeOf(Class)) .oclAsType(Class) ->asSet()
    in
      cl_members ->exists(clm | cd_domains ->includes(clm))
  )
in
let cd_connection_ifs: Set(Interface)=
  Package.allInstances() ->select(getAppliedStereotypes().name
      ->includes('ContextDiagram')) ->asSequence() ->first()
  .clientDependency.target
  ->select(oclIsTypeOf(Interface)).oclAsType(Interface) ->asSet()
in
let cd_connection_if_parts: Set(Interface)=
  cd_connection_ifs.member
  ->select(oclIsTypeOf(Property)).oclAsType(Property).type
  ->select(oclIsTypeOf(Interface)).oclAsType(Interface) ->asSet()
in
  Package.allInstances() ->select(getAppliedStereotypes().name
      ->includes('TechnicalContextDiagram'))
  ->forAll(pd_tcd |
43     pd_tcd.clientDependency -> select(getAppliedStereotypes().name
           ->includes(’isPart’)).target
44     ->select(oclIsTypeOf(Class))
45     ->select(
46       getAppliedStereotypes().name ->includes(’Domain’) or
47       getAppliedStereotypes().general.name ->includes(’Domain’) or
48       getAppliedStereotypes().general.general.name ->includes(’Domain’) or
49       getAppliedStereotypes().general.general.general.name
             ->includes(’Domain’)).oclAsType(Class)
50     ->forAll(pd_domain |
51       cd_domains ->includes(pd_domain) or
52       cd_domain_parts ->includes(pd_domain) or
53       cd_merged_domains ->includes(pd_domain)  or
54       cd_dom_concr ->includes(pd_domain) or
55       let concr_ifs_of_pd_domain: Set(Interface) =
56         pd_domain.clientDependency ->select(
57           getAppliedStereotypes().name ->includes(’concretizes’) or
58           getAppliedStereotypes().name ->includes(’refines’)
59         ).target.oclAsType(Interface) ->asSet()
60       in
61         concr_ifs_of_pd_domain -> exists(if_pd |
62           cd_connection_ifs ->includes(if_pd) or
63           cd_connection_if_parts ->includes(if_pd)
64         )
65     )
66   )
```

Listing C.52: *Domains in the technical context diagram are consistent to the domains in the context diagram*

A technical context diagram is consistent to a context diagram, if each domain in the technical context diagrams (lines 41-50)

- is a domain in the context diagram (line 51), or
- is part of a domain in the context diagram (line 52, operator: split domain), or
- is composed of several domains being part of the context diagram (line 53, operator: merge domain),
- refines or concretized a domain in the context diagram (line 54, operator: introduce connection domain),
- refines or concretized an interface in the context diagram (lines 55-62, operator: introduce connection domain), or
- refines or concretized a part of an interface in the context diagram (line 63, operator: introduce connection domain)

C.9. Constraints related to the consistency between problem diagrams and technical context diagrams

```
let pd_domains: Set(Class) =
  Package.allInstances() ->select(getAppliedStereotypes().name
      ->includes(’ProblemDiagram’))
  .clientDependency.target
  ->select(
        getAppliedStereotypes().name ->includes(’Domain’) or
        getAppliedStereotypes().general.name ->includes(’Domain’) or
        getAppliedStereotypes().general.general.name ->includes(’Domain’) or
        getAppliedStereotypes().general.general.general.name ->includes(’Domain’))
    .oclAsType(Class) ->asSet()
in
```

```
11 let pd_domain_parts: Set(Class) =
12   pd_domains.member
13   ->select(oclIsTypeOf(Property) and
         oclAsType(Property).type.oclIsTypeOf(Class)).oclAsType(Property).type
14   .oclAsType(Class) ->asSet()
15 in
16 let pd_dom_concr: Set(Class) =
17     Class.allInstances()->select(clientDependency->select(cdep|
           cdep.getAppliedStereotypes().name->includes('concretizes') and
           cdep.target->select(oclIsTypeOf(Class)).oclAsType(Class)
18     ->exists(d| pd_domains->includes(d)))->size()>=1)
19 in
20 let connection_domains: Set(Class) =
21   Class.allInstances() ->select(
22           getAppliedStereotypes().name ->includes('Domain') or
23           getAppliedStereotypes().general.name ->includes('Domain') or
24           getAppliedStereotypes().general.general.name ->includes('Domain') or
25           getAppliedStereotypes().general.general.general.name ->includes('Domain'))
26   ->select(
27           clientDependency.getAppliedStereotypes().name ->includes('refines')  or
28           clientDependency.getAppliedStereotypes().name ->includes('concretizes') )
29   ->select(clientDependency->exists(target->forAll(oclIsTypeOf(Interface))))
30   ->asSet()
31 in
32   Package.allInstances() ->select(getAppliedStereotypes().name
         ->includes('TechnicalContextDiagram') )
33   ->forAll(cd |
34     cd.clientDependency -> select(getAppliedStereotypes().name
           ->includes('isPart')).target
35     ->select(oclIsTypeOf(Class))
36
37     ->select(
38           getAppliedStereotypes().name ->includes('Domain') or
39           getAppliedStereotypes().general.name ->includes('Domain') or
40           getAppliedStereotypes().general.general.name ->includes('Domain') or
41           getAppliedStereotypes().general.general.general.name
               ->includes('Domain')).oclAsType(Class)
42     ->forAll(cd_dom |
43         pd_domains->includes(cd_dom) or
44         pd_domain_parts->includes(cd_dom) or
45         pd_dom_concr->includes(cd_dom) or
46         let cd_dom_parts: Set(Class) =
47             cd_dom.member
48             ->select(oclIsTypeOf(Property) and
                   oclAsType(Property).type.oclIsTypeOf(Class)).oclAsType(Property).type
49             ->select(oclIsTypeOf(Class))
50             ->select(
51                   getAppliedStereotypes().name ->includes('Domain') or
52                   getAppliedStereotypes().general.name ->includes('Domain') or
53                   getAppliedStereotypes().general.general.name ->includes('Domain') or
54                   getAppliedStereotypes().general.general.general.name
                         ->includes('Domain')).oclAsType(Class) ->asSet()
55         in
56           (cd_dom_parts->notEmpty() and
57           pd_domains->includesAll(cd_dom_parts))
58         or
59         connection_domains->includes(cd_dom)
60     )
61   )
```

Listing C.53: *Domains in the technical context diagram connected to the machine must be found in problem diagrams*

All domains in technical context diagrams must be (using the decomposition operators) in at least on problem diagram (lines 32-42). The domain is considered

- as the domain itself (line 43),
- as a merged domain (line 44),

- as a concretized or refined domain (line 45),
- as a split domain (lines 46-57), or
- as an interface concretizing a connection domain (line 59), operator: remove connection domain)

C.10. Constraints related to the consistency between sequence diagrams and operation specifications

```
 1 Lifeline.allInstances().coveredBy
 2   ->select(oclIsTypeOf(MessageOccurrenceSpecification) and name.substring(1,1)='R')
 3   .oclAsType(MessageOccurrenceSpecification).message.name
 4 ->forAll(receivedMsg |
 5     Package.allInstances()->select(getAppliedStereotypes().name
         ->includes('ProblemDiagram'))
 6     .clientDependency->select(getAppliedStereotypes().name ->includes('isPart'))
 7       .target->select(getAppliedStereotypes().name ->includes('Machine'))
 8       .ownedElement
 9             ->select(oclIsTypeOf(Operation)) .oclAsType(Operation)
                 .precondition->size()>0
10 )
```

Listing C.54: *Precondition exists for problem diagram machine operations used in sequence diagrams*

For each operation of problem diagram machines (lines 4-8) which are used in sequence diagrams (lines 1-3) a precondition must exist (line 9).

```
 1 Lifeline.allInstances().coveredBy
 2   ->select(oclIsTypeOf(MessageOccurrenceSpecification) and name .substring(1,1)='R')
 3   .oclAsType(MessageOccurrenceSpecification).message.name
 4 ->forAll(receivedMsg |
 5     Package.allInstances()->select(getAppliedStereotypes().name
         ->includes('ProblemDiagram'))
 6     .clientDependency->select(getAppliedStereotypes().name ->includes('isPart'))
 7       .target->select(getAppliedStereotypes().name ->includes('Machine'))
 8       .ownedElement
 9             ->select(oclIsTypeOf(Operation)) .oclAsType(Operation) .postcondition
                 ->size()>0
10 )
```

Listing C.55: *Postcondition exists for problem diagram machine operations used in sequence diagrams*

For each operation of problem diagram machines (lines 4-8) which are used in sequence diagrams (lines 1-3) a postcondition must exist (line 9).

C.11. Dependability

C.11.1. All

```
1 Class.allInstances()->select(
2 (getAppliedStereotypes().name->includes('Dependability') or
3 getAppliedStereotypes().general.name->includes('Dependability') or
4 getAppliedStereotypes().general.general.name->includes('Dependability') )
5 and getAppliedStereotypes().name->includes('Requirement'))
6 ->forAll(clientDependency->select(d |
7     d.oclAsType(Dependency).getAppliedStereotypes().name -> includes('complements'))
8     .oclAsType(Dependency).target.getAppliedStereotypes().name ->
        includes('Requirement')->count(true)>=1 )
```

Listing C.56: *Each dependability requirement complements another requirement*

A dependability requirement always complements (stereotype ≪complements≫) a functional requirement. In this OCL expression, all classes with a stereotype indicating a dependability statement or a subtype (e.g., ≪Integrity_att≫ or ≪Availability_rnd≫) and additionally the stereotype ≪Requirement≫ are selected in lines 1-5. In all of these requirement classes, it is checked that their dependencies (line 6) with the stereotype ≪complements≫ (line 7) point to at least one class with the stereotype ≪Requirement≫ (line 8).

C.11.2. Confidentiality

```
 1 Class.allInstances()->select(getAppliedStereotypes().name ->
       includes(’Confidentiality’))
 2 ->select(getAppliedStereotypes().name -> includes(’Statement’) or
 3   getAppliedStereotypes().name -> includes(’Requirement’))->forAll(
 4   clientDependency -> select(r | r.oclAsType(Dependency).getAppliedStereotypes().name
         -> includes(’constrains’))
 5 .oclAsType(Dependency).target.getAppliedStereotypes() -> collect(
 6   name->includes(’CausalDomain’) or
 7   general.name->includes(’CausalDomain’) or
 8   general.general.name->includes(’CausalDomain’)
 9 )->count(true)>=1)
```

Listing C.57: *Each confidentiality statement constrains a causal domain*

All classes in the model with the stereotypes ≪Confidentiality≫ and also ≪Statement≫ or ≪Requirement≫ are selected, and for all confidentiality statements the following condition is checked (lines 1-3). The dependencies starting at this class (clientDependency) with the stereotype ≪constrains≫ (line 4) are considered. The targets of the ’constrains’ are checked to have the stereotype ≪CausalDomain≫ or a subtype, and the boolean results are collected (lines 5-8). It is checked by counting the positive results if there is at least one causal domain (or a subtype) constrained (line 9).

```
 1 Class.allInstances()->select(getAppliedStereotypes().name ->
       includes(’Confidentiality’))
 2 ->select(getAppliedStereotypes().name -> includes(’Statement’) or
 3 getAppliedStereotypes().name -> includes(’Requirement’))->forAll(c|
 4   c.oclAsType(Class).getValue(c.oclAsType(Class) .getAppliedStereotypes() -> select(s
       |
 5             s.oclAsType(Stereotype).name -> includes(’Confidentiality’) )
 6             ->asSequence()->first(),’constrained’)
                   .oclAsType(ProblemFrames::CausalDomain).base_Class->asSet()
 7   =
 8   clientDependency -> select(r | r.oclAsType(Dependency).getAppliedStereotypes().name
         -> includes(’constrains’))
 9   .oclAsType(Dependency).target.oclAsType(Class)->asSet()
10 )
```

Listing C.58: *The attribute ’constrained’ contains the constrained domains*

For all classes in the model with the stereotypes ≪Confidentiality≫ and also ≪Statement≫ or ≪Requirement≫ (lines 1-3), the attribute constrained (lines 4-6) has the same elements (line 7) as the target of the dependencies starting at this class (clientDependency) with the stereotype ≪constrains≫ (line 8-9).

```
1 Class.allInstances()->select(getAppliedStereotypes().name ->
      includes(’Confidentiality’))->forAll(c |
2   c.oclAsType(Class).getValue(
3     c.oclAsType(Class).getAppliedStereotypes() ->select(s |
4       s.oclAsType(Stereotype).name ->includes(’Confidentiality’)
5      )->asSequence() ->first(),’attacker’
6   )->size()>=1
7 )
```

Listing C.59: *A confidentiality stereotype refers to at least one attacker*

The confidentiality stereotype needs to consider an attacker.

All classes in the model with the stereotype ≪Confidentiality≫ are selected, and for all confidentiality statements the following condition is checked (line 1). At least one element must exist in the attribute attacker (lines 2-7).

```
 1 Class.allInstances()->select(getAppliedStereotypes().name ->
       includes('Confidentiality'))->forAll(c |
 2   let stakeholders: Set(Class) =
 3     c.oclAsType(Class).getValue(
 4       c.oclAsType(Class).getAppliedStereotypes() ->select(s |
 5         s.oclAsType(Stereotype).name ->includes('Confidentiality')
 6       )->asSequence() ->first(),'stakeholder'
 7     ).oclAsType(Class)->asSet()
 8   in
 9   let attackers: Set(Class) =
10     c.oclAsType(Class).getValue(
11       c.oclAsType(Class).getAppliedStereotypes() ->select(s |
12         s.oclAsType(Stereotype).name ->includes('Confidentiality')
13       )->asSequence() ->first(),'attacker'
14     ).oclAsType(Class)->asSet()
15   in
16   stakeholders->size()>=1
17   and stakeholders->forAll(sh |
18     not attackers->includes(sh)
19   )
20 )
```

Listing C.60: *A confidentiality stereotype refers to at least one biddable domain (stakeholder - not the attacker)*

A confidentiality statement also needs to refer to the data's stakeholder. The Stakeholder is referred to, because we want to allow the access only to Stakeholders with legitimate interest. The instances of Stakeholder and Attacker must be disjoint. In the OCL expression in Listing C.60 for each confidentiality statement (Line 1) the stakeholders are retrieved (lines 2-8), the attackers are retrieved (lines 9-15), it is checked that at least one stakeholder is defined (line 16), and the stakeholders are not in the set of attackers (lines 17-19).

C.11.3. Integrity

```
 1 Class.allInstances() ->select(
 2   getAppliedStereotypes().general.name -> includes('Integrity')
 3   and getAppliedStereotypes().name -> includes('Requirement')
 4 )->forAll(ir |
 5   ir.oclAsType(Class).clientDependency
 6   ->select(r |  r.oclAsType(Dependency).getAppliedStereotypes().name ->
       includes('refersTo'))
 7   .oclAsType(Dependency).target->asSet() -> includesAll(
 8     ir.oclAsType(Class).clientDependency
 9     ->select(r | r.oclAsType(Dependency).getAppliedStereotypes().name ->
         includes('complements'))
10     .oclAsType(Dependency).target.oclAsType(Class) .clientDependency
11     ->select(r | r.oclAsType(Dependency).getAppliedStereotypes().name ->
         includes('constrains'))
12     .oclAsType(Dependency).target->asSet()
13   )
14 )
```

Listing C.61: *Integrity requirements refer to constrained domains*

An integrity requirement needs to refer to the domain constrained by the supplemented functional requirement. To validate that integrity requirements refer to the domain constrained in the functional statement, we check for all classes with the stereotypes ≪Integrity≫ and ≪Requirement≫ (lines 1-4 of Listing C.61) that the set of all classes referred to includes (lines 5-7) the set of all constrained classes (lines 11 and 12) of the supplemented functional requirements (lines 8-10).

```
Class.allInstances()->select(oe |
        oe.oclAsType(Class).getAppliedStereotypes().name -> includes('Integrity'))
->size()=0
```

Listing C.62: *No class with stereotype 'Integrity'*

Checks that no classes with the stereotype ≪integrity≫ exists.

```
1  Class.allInstances()->select(getAppliedStereotypes().name ->
       includes('Integrity_rnd') or getAppliedStereotypes().name ->
       includes('Integrity_att'))
2  ->select(getAppliedStereotypes().name -> includes('Statement') or
3  getAppliedStereotypes().name -> includes('Requirement'))->forAll(
4  clientDependency -> select(r | r.oclAsType(Dependency).getAppliedStereotypes().name
       -> includes('refersTo'))
5  .oclAsType(Dependency).target.getAppliedStereotypes() -> collect(
6  name->includes('CausalDomain') or
7  general.name->includes('CausalDomain') or
8  general.general.name->includes('CausalDomain')
9  )->count(true)>=1)
```

Listing C.63: *Integrity requirements refer to causal domains*

All classes in the model with the stereotypes ≪Integrity_att≫ or ≪Integrity_rnd≫ and also ≪Statement≫ or ≪Requirement≫ are selected, and for all confidentiality statements the following condition is checked (lines 1-3). The dependencies starting at this class (clientDependency) with the stereotype ≪refersTo≫ (line 4) are considered. The targets of the 'refersTo' are checked to have the stereotype ≪CausalDomain≫ or a subtype, and the boolean results are collected (lines 5-8). It is checked by counting the positive results if there is at least one causal domain (or a subtype) constrained (line 9).

```
1  Class.allInstances()->select(getAppliedStereotypes().name ->
       includes('Integrity_rnd') or getAppliedStereotypes().name ->
       includes('Integrity_att'))
2  ->select(getAppliedStereotypes().name -> includes('Statement') or
3  getAppliedStereotypes().name -> includes('Requirement'))->forAll(c|
4    c.oclAsType(Class).getValue(c.oclAsType(Class) .getAppliedStereotypes() -> select(s
       |
5              s.oclAsType(Stereotype).general.name -> includes('Integrity') )
6              ->asSequence()->first(),'constrainedByFunctional')
                   .oclAsType(ProblemFrames::CausalDomain).base_Class->asSet()
7    =
8    c.clientDependency -> select(r |
         r.oclAsType(Dependency).getAppliedStereotypes().name -> includes('refersTo'))
9    .oclAsType(Dependency).target.oclAsType(Class)->asSet()
10 )
```

Listing C.64: *'constrainedByFunctional' corresponds to 'refersTo'-references*

For all classes in the model with the ≪Integrity_att≫ or ≪Integrity_rnd≫ ≪Confidentiality≫ and also ≪Statement≫ or ≪Requirement≫ (lines 1-3), the attribute influencedIfViolation (lines 4-6) has the same elements (line 7) as the targets of the dependencies starting at this class (clientDependency) with the stereotype ≪constrains≫ (line 8-9).

```
1  Class.allInstances()->select(getAppliedStereotypes().name ->
       includes('Integrity_rnd') or getAppliedStereotypes().name ->
       includes('Integrity_att'))
2  ->select(getAppliedStereotypes().name -> includes('Statement') or
3  getAppliedStereotypes().name -> includes('Requirement'))->forAll(
4      clientDependency -> select(r |
           r.oclAsType(Dependency).getAppliedStereotypes().name ->
           includes('constrains'))
5      .oclAsType(Dependency).target.getAppliedStereotypes() -> collect(
6      name->includes('CausalDomain') or
7      general.name->includes('CausalDomain') or
8      general.general.name->includes('CausalDomain')
9 )->count(true)>=1)
```

Listing C.65: *Integrity requirements constrain causal domains*

All classes in the model with the stereotypes ≪Integrity_att≫ or ≪Integrity_rnd≫ and also ≪Statement≫ or ≪Requirement≫ are selected, and for all confidentiality statements the following condition is checked (lines 1-3). The dependencies starting at this class (clientDependency) with the stereotype ≪constrains≫ (line 4) are considered. The targets of the 'constrains' are checked to have the stereotype ≪CausalDomain≫ or a subtype, and the boolean results are collected (lines 5-8). It is checked by counting the positive results if there is at least one causal domain (or a subtype) constrained (line 9).

```
1  Class.allInstances()->select(getAppliedStereotypes().name -> includes('Integrity'))
2  ->select(getAppliedStereotypes().name -> includes('Statement') or
3  getAppliedStereotypes().name -> includes('Requirement'))->forAll(c |
4    c.oclAsType(Class).getValue(c.oclAsType(Class) .getAppliedStereotypes() -> select(s
       |
5              s.oclAsType(Stereotype).general.name -> includes('Integrity') )
6              ->asSequence()->first(),'influencedIfViolation')
                   .oclAsType(ProblemFrames::CausalDomain).base_Class->asSet()
7    =
8    clientDependency -> select(r | r.oclAsType(Dependency).getAppliedStereotypes().name
       -> includes('constrains'))
9    .oclAsType(Dependency).target.oclAsType(Class)->asSet()
10   and
11   c.oclAsType(Class).getValue(c.oclAsType(Class) .getAppliedStereotypes() -> select(s
       |
12             s.oclAsType(Stereotype).general.name -> includes('Integrity') )
13             ->asSequence()->first(),'actionIfViolation') .oclAsType(String)->asSet()
14   =
15   clientDependency -> select(r | r.oclAsType(Dependency).getAppliedStereotypes().name
       -> includes('constrains'))
16   .oclAsType(Dependency).name->asSet()
17 )
```

Listing C.66: *'influencedIfViolation' and 'actionIfViolation' correspond to constraining references*

For all classes in the model with the ≪Integrity_att≫ or ≪Integrity_rnd≫ ≪Confidentiality≫ and also ≪Statement≫ or ≪Requirement≫ (lines 1-3), the attribute influencedIfViolation (lines 4-6) has the same elements (line 7) as the targets of the dependencies starting at this class (clientDependency) with the stereotype ≪constrains≫ (line 8-9).

```
1 Class.allInstances()->select(oe |
2         oe.oclAsType(Class).getAppliedStereotypes().name -> includes('Integrity_rnd'))
3 ->forAll(c |
4         c.oclAsType(Class).getValue(c.oclAsType(Class) .getAppliedStereotypes() ->
              select(s |
5         s.oclAsType(Stereotype).name -> includes('Integrity_rnd') )
6         ->asSequence()->first(),'probability') <> null )
```

Listing C.67: *Integrity statements considering random faults contain probabilities*

For all integrity statements (lines 1-3), probability is relevant (and should be set, lines 4-6).

```
1 Class.allInstances()->select(oe |
2         oe.oclAsType(Class).getAppliedStereotypes().name -> includes('Integrity_att'))
3 ->forAll(c |
4         c.oclAsType(Class).getValue(c.oclAsType(Class) .getAppliedStereotypes() ->
              select(s |
5         s.oclAsType(Stereotype).name -> includes('Integrity_att') )
6         ->asSequence()->first(),'attacker') ->size()>0
7 )
```

Listing C.68: *Integrity statements considering an attacker refer to an attacker*

For an integrity statement (lines 1-3) attacker is relevant (and shall be set, lines 4-6).

C.11.4. Availability

```
Class.allInstances()->select(oe |
        oe.oclAsType(Class).getAppliedStereotypes().name -> includes('Availability'))
->size()=0
```

Listing C.69: *No class with stereotype 'Availability'*

Checks that no classes with the stereotype ≪Availability≫ exists.

```
1  Class.allInstances() ->select(
2    getAppliedStereotypes().general.name -> includes('Availability')
3    and getAppliedStereotypes().name -> includes('Requirement')
4  ) ->forAll(ar |
5    ar.oclAsType(Class).clientDependency
6    ->select(r | r.oclAsType(Dependency).getAppliedStereotypes().name ->
         includes('constrains'))
7    .oclAsType(Dependency).target->asSet() ->includesAll(
8      ar.oclAsType(Class).clientDependency
9      ->select(r | r.oclAsType(Dependency).getAppliedStereotypes().name ->
           includes('complements'))
10     .oclAsType(Dependency).target.oclAsType(Class) .clientDependency
11     ->select(r | r.oclAsType(Dependency).getAppliedStereotypes().name ->
           includes('constrains'))
12     .oclAsType(Dependency).target->asSet()
13   )
14 )
```

Listing C.70: *Availability requirements constrain constrained domains*

For all classes with the stereotypes ≪Availability_rnd≫ or ≪Availability_add≫ and ≪Requirement≫ (lines 1-4 of Listing C.70), the set of its constrained classes includes (lines 5-7) the set of all constrained classes (lines 11 and 12) of the complemented functional requirements (lines 8-10).

```
1 Class.allInstances()->select(getAppliedStereotypes().name ->
      includes('Availability_rnd') or getAppliedStereotypes().name ->
      includes('Availability_att'))
2 ->select(getAppliedStereotypes().name -> includes('Statement') or
3 getAppliedStereotypes().name -> includes('Requirement'))->forAll(
4   clientDependency -> select(r | r.oclAsType(Dependency).getAppliedStereotypes().name
        -> includes('constrains'))
5   .oclAsType(Dependency).target.getAppliedStereotypes() -> collect(
6   name->includes('CausalDomain') or
7   general.name->includes('CausalDomain') or
8   general.general.name->includes('CausalDomain')
9 )->count(true)>=1)
```

Listing C.71: *Availability requirements constrain causal domains*

All classes in the model with the stereotypes ≪Availability_att≫ or ≪Availability_rnd≫ and also ≪Statement≫ or ≪Requirement≫ are selected, and for all availability statements the following condition is checked (lines 1-3). The dependencies starting at this class (clientDependency) with the stereotype ≪constrains≫ (line 4) are considered. The targets of the 'constrains'-dependency are checked to have the stereotype ≪CausalDomain≫ or a subtype, and the boolean results are collected (lines 5-8). It is checked by counting the positive results if there is at least one causal domain (or a subtype) constrained (line 9).

```
Class.allInstances()->select(getAppliedStereotypes().name ->
    includes('Availability_rnd') or getAppliedStereotypes().name ->
    includes('Availability_att'))
->select(getAppliedStereotypes().name -> includes('Statement') or
getAppliedStereotypes().name -> includes('Requirement'))->forAll(c|
  c.oclAsType(Class).getValue(c.oclAsType(Class) .getAppliedStereotypes() -> select(s
      |
           s.oclAsType(Stereotype).general.name -> includes('Availability') )
6              ->asSequence()->first(),’constrained’)
                   .oclAsType(ProblemFrames::CausalDomain).base_Class->asSet()
7    =
8    c.clientDependency -> select(r |
         r.oclAsType(Dependency).getAppliedStereotypes().name -> includes(’constrained’))
9    .oclAsType(Dependency).target.oclAsType(Class)->asSet()
10 )
```

Listing C.72: *’constrained’ corresponds to constraining references*

For all classes in the model with the ≪Availability_att≫ or ≪Availability_rnd≫ and also ≪Statement≫ or ≪Requirement≫ (lines 1-3), the attribute constrained (lines 4-6) has the same elements (line 7) as the targets of the dependencies starting at this class (clientDependency) with the stereotype ≪constrains≫ (line 8-9).

```
1  Class.allInstances()->select(oe |
2          oe.oclAsType(Class).getAppliedStereotypes().name ->
               includes(’Availability_att’))
3  ->forAll(c |
4              (c.oclAsType(Class).getValue(c.oclAsType(Class) .getAppliedStereotypes()
                   -> select(s |
5              s.oclAsType(Stereotype).name -> includes(’Availability_att’) )
6              ->asSequence()->first(),’attacker’) ->size()>0)
7              and (c.oclAsType(Class).getValue(c.oclAsType(Class)
                   .getAppliedStereotypes() -> select(s |
8              s.oclAsType(Stereotype).name -> includes(’Availability_att’) )
9              ->asSequence()->first(),’forGroup’) ->size()>0)
10 )
```

Listing C.73: *Availability statements considering an attacker refer to attackers and stakeholders*

For available requirements considering an attacker the stereotype attributes forGroup and attacker must be specified, as required by the OCL expression in Listing C.73. For all availability statements (Lines 1-3) is checked that values forGroup and attacker are set (Lines 4-9).

```
1  Class.allInstances()->select(oe |
2          oe.oclAsType(Class).getAppliedStereotypes().name ->
               includes(’Availability_rnd’))
3  ->forAll(c |
4              c.oclAsType(Class).getValue(c.oclAsType(Class) .getAppliedStereotypes()
                   -> select(s |
5              s.oclAsType(Stereotype).name -> includes(’Availability_rnd’) )
6              ->asSequence()->first(),’probability’) <> null )
```

Listing C.74: *Availability statements considering random faults contain probabilities*

For all availability statements considering random faults (lines 1-3), the probability is relevant (and should be set, lines 4-6).

C.11.5. Reliability

```
Class.allInstances()->select(oe |
        oe.oclAsType(Class).getAppliedStereotypes().name -> includes(’Reliability’))
->size()=0
```

Listing C.75: *No class with stereotype ’Reliability’*

Checks that no classes with the stereotype ≪Reliability≫ exists.

```
Class.allInstances() ->select(
  getAppliedStereotypes().general.name -> includes(’Reliability’)
  and getAppliedStereotypes().name -> includes(’Requirement’)
) ->forAll(ar |
  ar.oclAsType(Class).clientDependency
  ->select(r | r.oclAsType(Dependency).getAppliedStereotypes().name ->
      includes(’constrains’))
```

```
 7    .oclAsType(Dependency).target->asSet() ->includesAll(
 8      ar.oclAsType(Class).clientDependency
 9      ->select(r | r.oclAsType(Dependency).getAppliedStereotypes().name ->
          includes('complements'))
10      .oclAsType(Dependency).target.oclAsType(Class) .clientDependency
11      ->select(r | r.oclAsType(Dependency).getAppliedStereotypes().name ->
          includes('constrains'))
12      .oclAsType(Dependency).target->asSet()
13    )
14  )
```

Listing C.76: *Reliability requirements constrain constrained domains*

Reliability requirements constrain (lines 1-6) the domain constrained by the complemented functional requirement (lines 7-12).

```
1  Class.allInstances()->select(getAppliedStereotypes().name ->
       includes('Reliability_rnd') or getAppliedStereotypes().name ->
       includes('Reliability_att'))
2  ->select(getAppliedStereotypes().name -> includes('Statement') or
3  getAppliedStereotypes().name -> includes('Requirement'))->forAll(
4    clientDependency -> select(r | r.oclAsType(Dependency).getAppliedStereotypes().name
         -> includes('constrains'))
5    .oclAsType(Dependency).target.getAppliedStereotypes() -> collect(
6    name->includes('CausalDomain') or
7    general.name->includes('CausalDomain') or
8    general.general.name->includes('CausalDomain')
9  )->count(true)>=1)
```

Listing C.77: *Reliability requirements constrain causal domains*

All classes in the model with the stereotypes ≪Reliability_att≫ or ≪Reliability_rnd≫ and also ≪Statement≫ or ≪Requirement≫ are selected, and for all availability statements the following condition is checked (lines 1-3). The dependencies starting at this class (clientDependency) with the stereotype ≪constrains≫ (line 4) are considered. The targets of the 'constrains'-dependency are checked to have the stereotype ≪CausalDomain≫ or a subtype, and the boolean results are collected (lines 5-8). It is checked by counting the positive results if there is at least one causal domain (or a subtype) constrained (line 9).

```
 1  Class.allInstances()->select(getAppliedStereotypes().name ->
        includes('Reliability_rnd') or getAppliedStereotypes().name ->
        includes('Reliability_att'))
 2  ->select(getAppliedStereotypes().name -> includes('Statement') or
 3  getAppliedStereotypes().name -> includes('Requirement'))->forAll( c |
 4    c.oclAsType(Class).getValue(c.oclAsType(Class) .getAppliedStereotypes() -> select(s
        |
 5              s.oclAsType(Stereotype).general.name -> includes('Reliability') )
 6              ->asSequence()->first(),'constrained')
                    .oclAsType(ProblemFrames::CausalDomain).base_Class->asSet()
 7              =
 8    c.clientDependency -> select(r |
        r.oclAsType(Dependency).getAppliedStereotypes().name -> includes('constrains'))
 9    .oclAsType(Dependency).target.oclAsType(Class)->asSet()
10  )
```

Listing C.78: *'constrained' corresponds to constraining references*

For all classes in the model with the ≪Reliability_att≫ or ≪Reliability_rnd≫ and also ≪Statement≫ or ≪Requirement≫ (lines 1-3), the attribute constrained (lines 4-6) has the same elements (line 7) as the targets of the dependencies starting at this class (clientDependency) with the stereotype ≪constrains≫ (line 8-9).

```
1  Class.allInstances()->select(oe |
2          oe.oclAsType(Class).getAppliedStereotypes().name ->
               includes('Reliability_att'))
3  ->forAll(c |
 4              (c.oclAsType(Class).getValue(c.oclAsType(Class) .getAppliedStereotypes()
                    -> select(s |
 5              s.oclAsType(Stereotype).name -> includes('Reliability_att') )
 6              ->asSequence()->first(),'attacker') ->size()>0)
 7              and (c.oclAsType(Class).getValue(c.oclAsType(Class)
                    .getAppliedStereotypes() -> select(s |
 8              s.oclAsType(Stereotype).name -> includes('Reliability_att') )
 9              ->asSequence()->first(),'forGroup') ->size()>0)
10 )
```

Listing C.79: *Reliability statements considering an attacker refer to attacker and forGroup*

For all reliability statement considering an attacker (lines 1-3), attacker and forGroup are relevant (and shall be set, lines 4-9).

```
1 Class.allInstances()->select(oe |
2         oe.oclAsType(Class).getAppliedStereotypes().name ->
              includes('Reliability_rnd'))
3 ->forAll(c |
4             c.oclAsType(Class).getValue(c.oclAsType(Class) .getAppliedStereotypes()
                  -> select(s |
5             s.oclAsType(Stereotype).name -> includes('Reliability_rnd') )
6             ->asSequence()->first(),'probability') <> null )
```

Listing C.80: *Reliability statements considering random fault contain probabilities*

For all relibility statement considering random faults (lines 1-3) the probability is relevant (and should be set, lines 4-6).

C.11.6. Authenticity

```
 1 Class.allInstances() ->select(
 2   getAppliedStereotypes().name -> includes('Authentication')
 3 ) ->forAll(ar |
 4   ar.oclAsType(Class).clientDependency
 5   ->select(r | r.oclAsType(Dependency).getAppliedStereotypes().name ->
         includes('constrains'))
 6   .oclAsType(Dependency).target->asSet()
 7   ->forAll(
 8     getAppliedStereotypes().name -> includes('CausalDomain')
 9     or getAppliedStereotypes().general.name -> includes('CausalDomain')
10     or getAppliedStereotypes().general.general.name -> includes('CausalDomain')
11   )
12 )
```

Listing C.81: *Each authentication statement constrains a causal domain*

Each authentication statement (lines 1-3) constrains (lines 4-6) a causal domain or a subtype (lines 7-10).

```
 1 Class.allInstances()->select(getAppliedStereotypes().name -> includes('Authenticity'))
 2 ->select(getAppliedStereotypes().name -> includes('Statement') or
 3 getAppliedStereotypes().name -> includes('Requirement'))->forAll(c|
 4   c.oclAsType(Class).getValue(c.oclAsType(Class) .getAppliedStereotypes() -> select(s
         |
 5             s.oclAsType(Stereotype).name -> includes('Authenticity') )
 6             ->asSequence()->first(),'influenced')
                   .oclAsType(ProblemFrames::CausalDomain).base_Class->asSet()
 7   =
 8   clientDependency -> select(r | r.oclAsType(Dependency).getAppliedStereotypes().name
         -> includes('constrains'))
 9   .oclAsType(Dependency).target.oclAsType(Class)->asSet()
10 )
```

Listing C.82: *'constrained' corresponds to constraining references*

For all classes in the model with the ≪Authenticity≫ and also ≪Statement≫ or ≪Requirement≫ (lines 1-3), the attribute constrained (lines 4-6) has the same elements (line 7) as the targets of the dependencies starting at this class (clientDependency) with the stereotype ≪constrains≫ (line 8-9).

```
1  Class.allInstances()->select(getAppliedStereotypes().name -> includes(’Authenticity’))
2  ->select(getAppliedStereotypes().name -> includes(’Statement’) or
3  getAppliedStereotypes().name -> includes(’Requirement’))->forAll(c|
4    c.oclAsType(Class).getValue(c.oclAsType(Class) .getAppliedStereotypes() -> select(s
       |
5              s.oclAsType(Stereotype).name -> includes(’Authenticity’) )
6              ->asSequence()->first(),’known’)
                   .oclAsType(ProblemFrames::Domain).base_Class->asSet()
7    =
8    c.clientDependency -> select(r |
9      r.oclAsType(Dependency).getAppliedStereotypes().name -> includes(’refersTo’)
10     and r.name=’known’
11   ).oclAsType(Dependency).target.oclAsType(Class)->asSet()
12 )
```

Listing C.83: *’known’ corresponds to referenced domain with dependency name ’known’*

For all classes in the model with the ≪Authenticity≫ and also ≪Statement≫ or ≪Requirement≫ (lines 1-3), the attribute known (lines 4-6) has the same elements (line 7) as the targets of the dependencies starting at this class (clientDependency) with the stereotype ≪refersTo≫ and the name known (line 8-11).

```
1  Class.allInstances()->select(oe |
2    oe.oclAsType(Class).getAppliedStereotypes().name -> includes(’Authenticity’))
3  ->forAll(c |
4    c.oclAsType(Class).getValue(c.oclAsType(Class) .getAppliedStereotypes() -> select(s
       |
5    s.oclAsType(Stereotype).name -> includes(’Authenticity’) )
6    ->asSequence()->first(),’known’) ->size()>0
7  )
```

Listing C.84: *Each authentication statement knows at least one domain being allowed to access*

In each authenticity statement (lines 1-3), the set known contains at least one domain (lines 4-6).

```
1  Class.allInstances()->select(getAppliedStereotypes().name -> includes(’Authenticity’))
2  ->select(getAppliedStereotypes().name -> includes(’Statement’) or
3  getAppliedStereotypes().name -> includes(’Requirement’))->forAll(c|
4    c.oclAsType(Class).getValue(c.oclAsType(Class) .getAppliedStereotypes() -> select(s
       |
5              s.oclAsType(Stereotype).name -> includes(’Authenticity’) )
6              ->asSequence()->first(),’unknown’)
                   .oclAsType(ProblemFrames::Domain).base_Class->asSet()
7    =
8    c.clientDependency -> select(r |
9      r.oclAsType(Dependency).getAppliedStereotypes().name -> includes(’refersTo’)
10     and r.name=’unknown’
11   ).oclAsType(Dependency).target.oclAsType(Class)->asSet()
12 )
```

Listing C.85: *’unknown’ corresponds to referenced domain with dependency name ’unknown’*

For all classes in the model with the ≪Authenticity≫ and also ≪Statement≫ or ≪Requirement≫ (lines 1-3), the attribute unknown (lines 4-6) has the same elements (line 7) as the targets of the dependencies starting at this class (clientDependency) with the stereotype ≪refersTo≫ and the name unknown (line 8-11).

C.11.7. Security management

```
1  Class.allInstances()->select(getAppliedStereotypes().name ->
       includes('SecurityManagement'))
2  ->select(getAppliedStereotypes().name -> includes('Statement') or
3  getAppliedStereotypes().name -> includes('Requirement'))->forAll(c|
4    c.oclAsType(Class).getValue(c.oclAsType(Class) .getAppliedStereotypes() -> select(s
         |
5              s.oclAsType(Stereotype).name -> includes('SecurityManagement') )
6              ->asSequence()->first(),'securityData')
                   .oclAsType(ProblemFrames::LexicalDomain).base_Class->asSet()
7    =
8    clientDependency -> select(r | r.oclAsType(Dependency).getAppliedStereotypes().name
         -> includes('constrains'))
9    .oclAsType(Dependency).target.oclAsType(Class)->asSet()
10 )
```

Listing C.86: *’constrained’ corresponds to constraining references*

For all classes in the model with the ≪SecurityManagement≫ and also ≪Statement≫ or ≪Requirement≫ (lines 1-3), the attribute constrained (lines 4-6) has the same elements (line 7) as the targets of the dependencies starting at this class (clientDependency) with the stereotype ≪constrains≫ (line 8-9).

```
1  Class.allInstances() ->select(
2    getAppliedStereotypes().name -> includes('SecurityManagement')
3  ) ->forAll(ar |
4    ar.oclAsType(Class).clientDependency
5    ->select(r | r.oclAsType(Dependency).getAppliedStereotypes().name ->
         includes('constrains'))
6    .oclAsType(Dependency).target->asSet()
7    ->forAll(
8      getAppliedStereotypes().name -> includes('LexicalDomain')
9    )
10 )
```

Listing C.87: *Each management statement constrains a CausalDomain*

Each security management statement (lines 1-3) constrains (lines 4-6) a lexical domain (lines 7-9).

```
1  Class.allInstances()->select(oe |
2    oe.oclAsType(Class).getAppliedStereotypes().name -> includes('SecurityManagement'))
3  ->forAll(c |
4    c.oclAsType(Class).getValue(c.oclAsType(Class) .getAppliedStereotypes() -> select(s
         |
5    s.oclAsType(Stereotype).name -> includes('SecurityManagement') )
6    ->asSequence()->first(),'validClient') ->size()>0
7    and
8    c.oclAsType(Class).getValue(c.oclAsType(Class) .getAppliedStereotypes() -> select(s
         |
9    s.oclAsType(Stereotype).name -> includes('SecurityManagement') )
10   ->asSequence()->first(),'attacker') ->size()>0
11 )
```

Listing C.88: *Each management statement contains a valid client and an attacker*

In each security management statement (lines 1-3), the set validClient contains at least one domain (lines 4-6) and at least one attacker is specified (lines 8-10).

C.11.8. Secret Distribution

```
1  Class.allInstances()->select(getAppliedStereotypes().name ->
       includes('SecretDistribution'))
2  ->select(getAppliedStereotypes().name -> includes('Statement') or
3  getAppliedStereotypes().name -> includes('Requirement'))->forAll(c|
4    c.oclAsType(Class).getValue(c.oclAsType(Class) .getAppliedStereotypes() -> select(s
         |
 5             s.oclAsType(Stereotype).name -> includes('SecretDistribution') )
 6             ->asSequence()->first(),'secret')
                   .oclAsType(ProblemFrames::LexicalDomain).base_Class->asSet()
 7   =
 8   c.clientDependency -> select(r |
         r.oclAsType(Dependency).getAppliedStereotypes().name -> includes('constrains'))
 9   .oclAsType(Dependency).target.oclAsType(Class)->asSet()
10 )
```

Listing C.89: *'constrained' corresponds to constraining references*

For all classes in the model with the ≪SecretDistribution≫ and also ≪Statement≫ or ≪Requirement≫ (lines 1-3), the attribute constrained (lines 4-6) has the same elements (line 7) as the targets of the dependencies starting at this class (clientDependency) with the stereotype ≪constrains≫ (line 8-9).

```
 1 Class.allInstances() ->select(
 2   getAppliedStereotypes().name -> includes('SecretDistribution')
 3 ) ->forAll(ar |
 4   ar.oclAsType(Class).clientDependency
 5   ->select(r | r.oclAsType(Dependency).getAppliedStereotypes().name ->
         includes('constrains'))
 6   .oclAsType(Dependency).target->asSet()
 7   ->forAll(
 8     getAppliedStereotypes().name -> includes('LexicalDomain')
 9   )
10 )
```

Listing C.90: *Each secret distribution statement constrains a causal domain*

Each secret distribution statement (lines 1-3) constrains (lines 4-6) a lexical domain (lines 7-9).

```
 1 Class.allInstances()->select(oe |
 2   oe.oclAsType(Class).getAppliedStereotypes().name -> includes('SecretDistribution'))
 3 ->forAll(c |
 4   c.oclAsType(Class).getValue(c.oclAsType(Class) .getAppliedStereotypes() -> select(s
         |
 5   s.oclAsType(Stereotype).name -> includes('SecretDistribution') )
 6   ->asSequence()->first(),'validClient') ->size()>0
 7   and
 8   c.oclAsType(Class).getValue(c.oclAsType(Class) .getAppliedStereotypes() -> select(s
         |
 9   s.oclAsType(Stereotype).name -> includes('SecretDistribution') )
10   ->asSequence()->first(),'attacker') ->size()>0
11 )
```

Listing C.91: *Each secret distribution statement contains a valid client and an attacker*

In each secret distribution statement (lines 1-3), the set validClient contains at least one domain (lines 4-6) and at least one attacker is specified (lines 8-10).

C.12. Constraints related to the consistency between technical context diagram and software architectures

Listing C.92: *Machine interfaces in technical context diagrams are covered by machine port interfaces*

```
let tcd_ifs: Set(Interface) =
  Package.allInstances() ->select(getAppliedStereotypes().name
      ->includes('TechnicalContextDiagram'))
  .clientDependency->select(getAppliedStereotypes().name ->includes('isPart')).target
  ->select(oclIsTypeOf(Interface)).oclAsType(Interface)->asSet()
in
  Package.allInstances() ->select(getAppliedStereotypes().name
      ->includes('TechnicalContextDiagram'))
  .clientDependency->select(getAppliedStereotypes().name ->includes('isPart')).target
```

```
8    ->select(getAppliedStereotypes().name ->includes('Machine')).oclAsType(Class)
9    ->forAll(tcd_machine |
10     let tcd_machine_ifs: Set(Interface) =
11       tcd_machine.clientDependency
12       ->select(
13           getAppliedStereotypes().name ->includes('observes') or
14           getAppliedStereotypes().name ->includes('controls'))
15       .target.oclAsType(Interface)->select(mif | tcd_ifs->includes(mif))
16       ->asSet()
17     in
18     let tcd_machine_port_ifs: Set(Interface) =
19       tcd_machine.member ->select(oclIsTypeOf(Port)).oclAsType(Port).required ->union(
20       tcd_machine.member ->select(oclIsTypeOf(Port)).oclAsType(Port).provided)
21       ->asSet()
22     in
23     let tcd_mp_contained_ifs: Set(Interface) =
24       tcd_machine_port_ifs.member
25       ->select(oclIsTypeOf(Property)) .oclAsType(Property).type
26       ->select(oclIsTypeOf(Interface)) .oclAsType(Interface) ->asSet()
27     in
28     let tcd_mp_combined_ifs: Set(Interface) =
29       Interface.allInstances()->select(
30         let comb_elem: Set(Interface) =
31           member->select(oclIsTypeOf(Property)) .oclAsType(Property).type
32           ->select(oclIsTypeOf(Interface)) .oclAsType(Interface) ->asSet()
33         in
34           comb_elem->size()>0 and
35           tcd_machine_port_ifs->includesAll(comb_elem)
36       )
37     in
38     let tcd_mp_concr_ifs: Set(Interface) =
39       tcd_machine_port_ifs.clientDependency->select(
40         getAppliedStereotypes().name ->includes('concretizes') or
41         getAppliedStereotypes().name ->includes('refines')
42       ).target
43       ->select(oclIsTypeOf(Interface)).oclAsType(Interface) ->asSet()
44     in
45     let tcd_mp_abstr_ifs: Set(Interface) =
46       Interface.allInstances()->select(
47         let abstr_ifs: Set(Interface) =
48           clientDependency->select(
49             getAppliedStereotypes().name ->includes('concretizes') or
50             getAppliedStereotypes().name ->includes('refines')
51           ).target
52           ->select(oclIsTypeOf(Interface)).oclAsType(Interface) ->asSet()
53         in
54           tcd_machine_port_ifs->exists(ti | abstr_ifs->includes(ti))
55         )
56     in
57       tcd_machine_ifs ->forAll( tmi|
58             tcd_machine_port_ifs ->includes(tmi) or
59             tcd_mp_contained_ifs ->includes(tmi) or
60             tcd_mp_combined_ifs ->includes(tmi) or
61             tcd_mp_concr_ifs ->includes(tmi) or
62             tcd_mp_abstr_ifs ->includes(tmi)
63       )
64   )
```

The condition checks that for each observed or controlled interface of each machine in the technical context diagram, the corresponding machine component of the architecture has ports providing or requiring the corresponding interfaces. The OCL expression in Listing C.92 on the preceding page checks that for each observed or controlled interface (line 6) of each machine in technical context diagrams (lines 1-5) the machine has ports providing or requiring

- exactly the observed or controlled interface,
- interfaces combined from observed or controlled interface,
- interfaces contained in observed or controlled interface,

- interfaces being refined or concretized by observed or controlled interface, or
- interfaces refining or concretizing observed or controlled interface.

Listing C.93: *Each association stereotype corresponds to a connection stereotype*

```
1  let tcd_domains: Set(Class) =
2    Package.allInstances() ->select(getAppliedStereotypes().name
         ->includes(’TechnicalContextDiagram’))
3    .clientDependency->select(getAppliedStereotypes().name ->includes(’isPart’)).target
4    ->select(oclIsTypeOf(Class)).oclAsType(Class)->asSet()
5  in
6    Package.allInstances() ->select(getAppliedStereotypes().name
         ->includes(’TechnicalContextDiagram’))
7    .clientDependency->select(getAppliedStereotypes().name ->includes(’isPart’)).target
8    ->select(getAppliedStereotypes().name ->includes(’Machine’)).oclAsType(Class)
9    ->forAll(tcd_machine |
10     let machine_associations: Set(Association) =
11       Association.allInstances()
12       ->select(endType->select(oclIsTypeOf(Class))
             .oclAsType(Class)->includes(tcd_machine))
13       ->select(endType->select(oclIsTypeOf(Class)) .oclAsType(Class)->forAll(ae |
             tcd_domains->includes(ae)))
14       ->asSet()
15     in
16     let tcd_machine_ports: Set(Port) =
17       tcd_machine.member->select(oclIsTypeOf(Port)).oclAsType(Port)
18       ->asSet()
19     in
20     let m_connection: Set(Connector)=
21       Connector.allInstances()
22       ->select(
23                 end->exists(e | tcd_machine_ports.end->includes(e))
24       )
25     in
26       machine_associations.getAppliedStereotypes().name ->forAll(masn |
27                 m_connection.getAppliedStereotypes() .name->includes(masn) or
28                 m_connection.getAppliedStereotypes() .general.name->includes(masn) or
29                 m_connection.getAppliedStereotypes()
                       .general.general.name->includes(masn) or
30                 m_connection.getAppliedStereotypes()
                       .general.general.general.name->includes(masn)
31       )
32   )
```

This condition checks for each machine in the technical context diagram (lines 6-9) that the stereotype names of all associations related to the machine (line 27) is included in the set of stereotype names of the connectors from internal components to external interfaces inside the machine (line 16-26). We also consider specializations of the stereotypes (lines 11-15 and 28-31).

Listing C.94: *Each connection stereotype corresponds to an association stereotype*

```
let tcd_domains: Set(Class) =
  Package.allInstances()->select(getAppliedStereotypes().name
      ->includes(’TechnicalContextDiagram’))
  .clientDependency->select(getAppliedStereotypes().name ->includes(’isPart’)).target
  ->select(oclIsTypeOf(Class)).oclAsType(Class)->asSet()
in
  Package.allInstances()->select(getAppliedStereotypes().name
      ->includes(’TechnicalContextDiagram’))
  .clientDependency->select(getAppliedStereotypes().name ->includes(’isPart’)).target
  ->select(getAppliedStereotypes().name ->includes(’Machine’)).oclAsType(Class)
  ->forAll(tcd_machine |
    let machine_associations: Set(Association) =
      Association.allInstances()
      ->select(endType->select(oclIsTypeOf(Class))
          .oclAsType(Class)->includes(tcd_machine))
      ->select(endType->select(oclIsTypeOf(Class)) .oclAsType(Class)->forAll(ae |
          tcd_domains->includes(ae)))
      ->asSet()
    in
    let tcd_machine_ports: Set(Port) =
      tcd_machine.member->select(oclIsTypeOf(Port)) .oclAsType(Port)->asSet()
    in
    let m_connection: Set(Connector)=
      Connector.allInstances()
      ->select(
        end->exists(e | tcd_machine_ports.end->includes(e))
      )
    in
      m_connection.getAppliedStereotypes()->forAll(mcs |
                machine_associations.getAppliedStereotypes().name ->includes(mcs.name
                    ->asSequence()->first()) or
                machine_associations.getAppliedStereotypes().name
                    ->includes(mcs.general.name ->asSequence()->first()) or
                machine_associations.getAppliedStereotypes().name
                    ->includes(mcs.general.general.name ->asSequence()->first()) or
                machine_associations.getAppliedStereotypes().name
                    ->includes(mcs.general.general.general.name
                    ->asSequence()->first()) or
                machine_associations.getAppliedStereotypes().name
                    ->includes(mcs.general.general .general.general.name
                    ->asSequence()->first())
      )
  )
```

The condition checks that each stereotype name of the connectors from components to external interfaces inside the machine must be included in the set of associations (or their specialization) to the machine in the technical context diagram.

C.13. Constraints related to the consistency between the different software architectures

Listing C.95: *Machines in problem diagrams must be components*

```
1 let contained_classes_in_machines_and_components: Set(Class) =
2   Class.allInstances()
3   ->select(getAppliedStereotypes().name ->includes(’Component’) or
        getAppliedStereotypes().name ->includes(’Machine’))
4   .member->select(oclIsTypeOf(Property)) .oclAsType(Property).type
        ->select(oclIsTypeOf(Class)) .oclAsType(Class)->asSet()
5 in
6   Class.allInstances()
7   ->select(getAppliedStereotypes().name ->includes(’Component’) or
        getAppliedStereotypes().name ->includes(’ReusedComponent’))->asSet()
8   ->forAll(component | contained_classes_in_machines_and_components
        ->includes(component))
```

This condition checks that all components must either be combined into the machine or into another component. The OCL expression in Listing C.95, defines the set contained_classes_in_machines_and_components by selecting all classes in machines or components (lines 1-5). We then check that all components (lines 6 and 7) only contain classes of contained_classes_in_machines_and_components (line 8).

Listing C.96: *Operations in required interfaces are provided*

```
Class.allInstances()->select(
  getAppliedStereotypes().name->includes(’Machine’) or
  getAppliedStereotypes().name->includes(’Component’)
)->forAll(mc |
  let mc_comps: Set(Class) =
    mc.member->select(oclIsTypeOf(Property)) .oclAsType(Property).type
    ->select(oclIsTypeOf(Class))
        ->select(getAppliedStereotypes().name->includes('Component'))
    .oclAsType(Class) ->asSet()
  in
    mc_comps.oclAsType(Class).ownedPort->forAll( mcc_p |
      let mcc_p_conn_p: Set(Port) =
        mc_comps.ownedPort ->select(mc_comps_p |
          Connector.allInstances()->exists( conne |
            let ce_mc_comps_p: Set(ConnectorEnd) =
              conne.end->select(ce | mc_comps_p.end->includes(ce))
            in
              ce_mc_comps_p->size()=1 and
              conne.end->exists(ce | mcc_p.end->includes(ce) and not
                  ce_mc_comps_p->includes(ce))
          )
        )->asSet()
       in
         mcc_p.oclAsType(Port).required.ownedOperation.name ->forAll(opn |
           mcc_p_conn_p.provided.ownedOperation.name ->includes(opn)
           or mcc_p_conn_p->size()=0
         )
    )
)
```

The condition checks that for each operation in a required interface of a port of a component there exists a connector to a port providing an interface with this operation.

Listing C.97: *Each operation in external interfaces of the machine corresponds to an internal one*

```
Class.allInstances()->select(
  getAppliedStereotypes().name->includes('Machine') or
  getAppliedStereotypes().name->includes('Component')
)->forAll(mc |
  let mc_comps: Set(Class) =
    mc.member->select(oclIsTypeOf(Property)) .oclAsType(Property).type
    ->select(getAppliedStereotypes().name->includes('Component'))
    .oclAsType(Class) ->asSet()
  in
    mc.oclAsType(Class).ownedPort->forAll( mc_p |
      let mc_p_conn_p: Set(Port) =
        mc_comps.ownedPort ->select(mc_comps_p |
          Connector.allInstances()->exists( conne |
            conne.end->exists(ce | mc_comps_p.end->includes(ce)) and
            conne.end->exists(ce | mc_p.end->includes(ce))
          )
        )->asSet()
       in
         (mc_p.oclAsType(Port).provided.ownedOperation.name ->forAll(opn |
           mc_p_conn_p.provided.ownedOperation.name ->includes(opn)
         ) and
         mc_p.oclAsType(Port).required.ownedOperation.name ->forAll(opn |
            mc_p_conn_p.required.ownedOperation.name ->includes(opn)
         )) or mc_p_conn_p->size()=0
    ) )
```

This condition checks that the component's interfaces must fit to the connected interfaces of the machine, i.e., each operation in a required interface of a port must correspond to an operation of a provided interface of the connected port.

Listing C.98: *Active classes only contain active classes*

```
1 Class.allInstances() ->forAll(c |
2     c.member->select(oclIsTypeOf(Property) and
          oclAsType(Property).type.oclIsTypeOf(Class)) .oclAsType(Property)
3     ->forAll(type.oclAsType(Class).isActive implies c.isActive))
```

The condition checks that passive components cannot contain any active components: The OCL expression in Listing C.98, checks for all classes c (line 1) that if they contain other classes (line 2) that these classes must also be active (line 3).

Listing C.99: *Allowed stereotypes for machines*

```
Class.allInstances() ->forAll(c |
  not Package.allInstances() ->select(getAppliedStereotypes().name
    ->includes(’ProblemFrame’)).clientDependency.target
    ->select(getAppliedStereotypes().name->includes(’Machine’))
        .oclAsType(Class)->includes(c)
  and
  c.getAppliedStereotypes().name->includes(’Machine’)
  implies (
    c.getAppliedStereotypes().name->includes(’Component’) or
    c.getAppliedStereotypes().name->includes(’ReusedComponent’) or
    c.getAppliedStereotypes().name->includes(’Task’) or
    c.getAppliedStereotypes().name->includes(’Process’) or
    c.getAppliedStereotypes().name->includes(’Local’) or
    c.getAppliedStereotypes().name->includes(’Distributed’)
  )
)
```

This condition checks that classes with the stereotype ≪machine≫ must either have the stereotype ≪Distributed≫, ≪Local≫, ≪Process≫, ≪Task≫, ≪Component≫ or ≪ReusedComponent≫ assigned.

Listing C.100: *Allowed sub-stereotypes for ≪Task≫, ≪Process≫, and ≪Local≫*

```
Class.allInstances() ->forAll(c |
  c.member->select(oclIsTypeOf(Property) and
      oclAsType(Property).type.oclIsTypeOf(Class)) .oclAsType(Property)
  ->forAll(
    (
      c.getAppliedStereotypes().name->includes(’Task’) implies (
      not type.oclAsType(Class).getAppliedStereotypes().name ->includes(’Process’) and
      not type.oclAsType(Class).getAppliedStereotypes().name ->includes(’Local’) and
      not type.oclAsType(Class).getAppliedStereotypes().name
          ->includes(’Distributed’))
    ) and
    (
      c.getAppliedStereotypes().name->includes(’Process’) implies (
      not type.oclAsType(Class).getAppliedStereotypes().name ->includes(’Local’) and
      not type.oclAsType(Class).getAppliedStereotypes().name
          ->includes(’Distributed’))
    ) and
    (
      c.getAppliedStereotypes().name->includes(’Local’) implies (
      not type.oclAsType(Class).getAppliedStereotypes().name
          ->includes(’Distributed’))
    )
  )
)
```

The condition checks that a class with the stereotype ≪Task≫ cannot contain classes with the stereotype ≪Process≫, ≪Local≫, or ≪Distributed≫. Furthermore, it checks that a class with the stereotype ≪Process≫ cannot contain classes with the stereotype ≪Local≫ or ≪Distributed≫. Finally, it checks that a class with the stereotype ≪Local≫ cannot contain classes with the stereotype ≪Distributed≫.

Listing C.101: *The connector types are the same*

```
Class.allInstances()->select(
  getAppliedStereotypes().name ->includes(’intermediate_architecture’) or
  getAppliedStereotypes().name ->includes(’layered_architecture’)
)->forAll(ila |
  Connector.allInstances()->select( con |
    con.end->exists( ce |
      ila.general.oclAsType(Class).ownedPort.end->includes(ce)
    )
  ).getAppliedStereotypes().name->asSet()
  =
11   Connector.allInstances()->select( con |
12     con.end->exists( ce |
13       ila.ownedPort.end->includes(ce)
14     )
15   ).getAppliedStereotypes().name->asSet()
16 )
```

With this expression we check the following: For all implementable/layered architectures (lines 1-4) the connectors with a connection to a port of the more general architecture are selected (lines 5-7) and their stereotypes must be the same (lines 8 and 9) as the stereotypes of the connectors to ports of the specialized architecture (lines 10-15).

Listing C.102: *The stereotypes ≪Physical≫ and ≪ui≫ and their subtypes are not between components*

```
 1 let components: Set(Class) =
 2   Class.allInstances()->select(not getAppliedStereotypes().name
       ->includes(’Distributed’)).ownedAttribute.type
 3   ->select(oclIsTypeOf(Class)).oclAsType(Class) ->select(getAppliedStereotypes().name
       ->includes(’Component’))->asSet()
 4 in
 5 let component_ports: Set(Port) =
 6   components.oclAsType(Class).ownedPort->asSet()
 7 in
 8   Connector.allInstances()->select( c |
 9     c.end->select(connector_end |
10       component_ports.end->includes(connector_end)
11     )->size()=2
12   )->select( c |
13     c.getAppliedStereotypes().name->includes(’ui’) or
14     c.getAppliedStereotypes().name->includes(’physical’) or
15     c.getAppliedStereotypes().general.name->includes(’ui’) or
16     c.getAppliedStereotypes().general.name->includes(’physical’) or
17     c.getAppliedStereotypes().general.general.name ->includes(’ui’) or
18     c.getAppliedStereotypes().general.general.name ->includes(’physical’)
19   )->select( c |
20     Port.allInstances()->select(p |
21       p.end->exists(pe | c.end->includes(pe)) and
22       Connector.allInstances()->select(getAppliedStereotypes() =
           c.getAppliedStereotypes()).end->exists(ce | p.end->includes(ce))
23     )->size()<>2
24   )->size()=0
```

This expression checks that the stereotypes ≪physical≫ and ≪ui≫ as well as their sub-types are not used between two components of a class with stereotype ≪Local≫, ≪Process≫, or ≪Task≫. This is achieved as follows: The set components is defined to be all classes being not in classes with the stereotype ≪Distributed≫ with the stereotype component (lines 1-4), the set component_ports is defined to be all ports of these components (line 5-7). In line 8-11 all connectors connecting component_ports are selected. In line 12-18 the connectors with the stereotypes ≪physical≫ and ≪ui≫ as well as their sub-types are selected. The returned set must be empty (line 24). To allow that connectors with these stereotypes are allowed within internal components connected to external ports, we ignore the connectors where a second connector with the same stereotype is connected to the port (lines 19-23).

Listing C.103: *Implementable/layered architecture: ports connected to external ports have the same type*

```
 1 Class.allInstances()->select(
 2   getAppliedStereotypes().name ->includes(’implementable_architecture’) or
 3   getAppliedStereotypes().name ->includes(’layered_architecture’)
 4 )->forAll(ila |
 5   let a_ports: Set(Port) =
 6     ila.oclAsType(Class).ownedPort->asSet()
 7   in
 8   let a_parts: Set(Class) =
 9     ila.ownedElement ->select(oclIsTypeOf(Property)).oclAsType(Property)
10     .type->select(oclIsTypeOf(Class)).oclAsType(Class)->asSet()
11   in
12   let a_part_ports: Set(Port) =
13     a_parts.oclAsType(Class).ownedPort->asSet()
14   in
15   a_ports->forAll(ap |
16     let ap_connected_ports: Set(Port) =
17       a_part_ports->select( app |
18         Connector.allInstances()->exists( c |
19           c.end->exists(connector_end |
20             ap.end->includes(connector_end)
21           )  and
22           c.end->exists(connector_end |
23             app.end->includes(connector_end)
24           )
25         )
26       )->asSet()
27     in
28       ap_connected_ports->size()=1 and
29       ap_connected_ports.type->asSequence()->first()=ap.type
30   )
31 )
```

The condition checks that ports of components of implementable/layered architectures connected to external ports require and provide the same interfaces. This is achieved as follows: For all implementable and layered architectures (lines 1-4)

- The owned ports of the architecture are determined (variable ap_port, lines 5-7).
- The contained classes are determined (variable a_parts, lines 8-11).
- The ports of contained classes are determined (variable a_part_ports, lines 12-14).

For all ports (variable ap) of all implementable and layered architectures (line 15) the ports connected to them (variable ap_connected_ports) are determined by checking if a connector exists (line 18)

- with a connector end being also a connector end of the port ap (lines 19-21) and
- with a connector end being also a connector end of the ports of the contained elements app (lines 23-25).

For each port (ap) of all implementable and layered architectures (line 15) it is then checked that it has only one connector to an internal component (line 28) and the type of the connected port app is of the same type as ap and they therefore require and provide the same interfaces (line 29).

Listing C.104: *The stereotypes ≪Shared memory≫, ≪unix_pipe≫ as well as their sub-types and ≪call_and_return≫ are not allowed between classes with the stereotypes ≪Local≫ or ≪Distributed≫*

```
let local_dist_ports: Set(Port) =
  Class.allInstances()->select(
    getAppliedStereotypes().name->includes(’Local’) or
    getAppliedStereotypes().name->includes(’Distributed’)
  ).ownedPort->asSet()
in
  Connector.allInstances()->select( c |
    c.end->select(connector_end |
      local_dist_ports.end->includes(connector_end)
    )->size()=2
  )->select( c |
    c.getAppliedStereotypes().name->includes(’shared_memory’) or
    c.getAppliedStereotypes().name->includes(’unix_pipe’) or
    c.getAppliedStereotypes().name ->includes(’call_return’) or
    c.getAppliedStereotypes().general.name ->includes(’shared_memory’) or
    c.getAppliedStereotypes().general.name ->includes(’unix_pipe’) or
    c.getAppliedStereotypes().general.name ->includes(’call_return’) or
    c.getAppliedStereotypes().general.general.name ->includes(’shared_memory’) or
    c.getAppliedStereotypes().general.general.name ->includes('unix_pipe') or
    c.getAppliedStereotypes().general.general.name ->includes('call_return')
  )->size()=0
```

This expression checks that connectors with the stereotypes ≪shared_memory≫, ≪unix_pipe≫, ≪call_return≫, or their sub-types do not connect classes with the stereotype ≪Local≫ or ≪Distributed≫ assigned.

Appendix D

OCL expressions for UMLsec

The following listing contains the specification for step 1 expressed in OCL:

```
createDeploymentDiagram(diagramName: String)
PRE   -- package with name inDiagram does not exist
     Package.allInstances() ->select(name=diagramName)
         ->size()=0 and
     -- exactly one context diagram exists
     Package.allInstances() ->select(getAppliedStereotypes()
       .name ->includes('ContextDiagram')) ->size()=1
POST -- package with name inDiagram exists
     Package.allInstances() ->select(name=diagramName)
         ->size()=1
```

Listing D.1: *createDeploymentDiagram(diagramName: String)*

The following listing contains the specification for step 2 expressed in OCL:

```
addSecureLinksStereotype(diagramName: String, adv: String)
PRE   -- package with name diagramName exists
     Package.allInstances() ->select(name=diagramName)
         ->size()=1  and
     (adv='default' or adv='insider')
POST Package.allInstances() ->select(name=diagramName)
     .getAppliedStereotypes().name ->includes('secure links') and
     Package.allInstances() ->select(name=diagramName)
     .getValue(Package.allInstances() ->select(name=diagramName)
        .getAppliedStereotypes()
         ->select(s .oclAsType(Stereotype).name ->includes('secure links'))
         ->asSequence() ->first(),'adversary')
      .oclAsType(String) ->includes(adv)
```

Listing D.2: *addSecureLinksStereotype(inDiagram: String, adv: String)*

The following listing contains the specification for step 3 expressed in OCL:

```
createNodes(inDiagram: String)
PRE:  Package.allInstances() ->select(name=diagramName)
         ->size()=1 and
      Package.allInstances() ->select(getAppliedStereotypes()
       .name ->includes('ContextDiagram')) ->size()=1
POST: Package.allInstances() ->select(name=inDiagram).ownedElement
         ->select(oclIsTypeOf(Node)) .oclAsType(Node).name =
      Package.allInstances() ->select(getAppliedStereotypes()
        .name ->includes('ContextDiagram')) .clientDependency
        .target ->select(oclIsTypeOf(Class) and
        not getAppliedStereotypes().name
         ->includes('BiddableDomain')) .oclAsType(Class).name and
        not getAppliedStereotypes().name
         ->includes('ConnectionDomain')) .oclAsType(Class).name and
```

Listing D.3: *createNodes(inDiagram: String)*

For step 4, either a nested node or a nested class can be generated. The following listing contains the specification for step 4, generating a nested node:

```
createNestedNodes(domainNames: String[])
PRE   -- exactly on context diagram exists
      Package.allInstances() ->select(getAppliedStereotypes()
       .name ->includes('ContextDiagram')) ->size()=1 and
      -- For all domainNames: A node exists that corresponds to the domain containing the domain with domainName
      domainNames ->forAll(dn |
        let nodeName: String =
          Package.allInstances() ->select(getAppliedStereotypes()
            .name ->includes('ContextDiagram')) .clientDependency
            .target ->select(oclIsTypeOf(Class) and
            not getAppliedStereotypes().name
             ->includes('BiddableDomain')) .oclAsType(Class)
             ->select(member ->select(oclIsTypeOf(Property))
               .oclAsType(Property).type ->select(oclIsTypeOf(Class))
               .oclAsType(Class).name ->includes(dn)
            ) ->asSequence() ->first().name
          in
            Node.allInstances().name->exists(nodeName) and
      -- verify that the given domains are part of another domain
      Package.allInstances() ->select(getAppliedStereotypes()
        .name ->includes('ContextDiagram')) .clientDependency
        .target ->select(oclIsTypeOf(Class) and
        not getAppliedStereotypes().name
         ->includes('BiddableDomain')) .oclAsType(Class)
        .member ->select(oclIsTypeOf(Property))
         .oclAsType(Property).type ->select(oclIsTypeOf(Class))
         .oclAsType(Class).name ->includesAll(domainNames)
POST domainNames ->forAll(dn |
        let nodeName: String =
          Package.allInstances() ->select(getAppliedStereotypes()
            .name ->includes('ContextDiagram')) .clientDependency
            .target ->select(oclIsTypeOf(Class) and
            not getAppliedStereotypes().name
             ->includes('BiddableDomain')) .oclAsType(Class)
             ->select(member ->select(oclIsTypeOf(Property))
               .oclAsType(Property).type ->select(oclIsTypeOf(Class))
               .oclAsType(Class).name ->includes(dn)
            ) ->asSequence() ->first().name
        in
          Package.allInstances() ->select(name=inDiagram).ownedElement
           ->select(oclIsTypeOf(Node)) .oclAsType(Node)
           ->select(name=nodeName).ownedElement
           ->select(oclIsTypeOf(Node)) .oclAsType(Node)
           ->select(name=dn) ->size()=1
     )
```

Listing D.4: *createNestedNodes(domainNames: Set(String))*

The following listing contains the specification for step 4, generating a nested class:

```
createNestedClasses(domainNames: String[])
PRE   -- exactly on context diagram exists
      Package.allInstances() ->select(getAppliedStereotypes()
       .name ->includes('ContextDiagram')) ->size()=1 and
      -- For all domainNames: A node exists that corresponds to the domain containing the domain with domainName
      domainNames ->forAll(dn |
        let nodeName: String =
          Package.allInstances() ->select(getAppliedStereotypes()
            .name ->includes('ContextDiagram')) .clientDependency
            .target ->select(oclIsTypeOf(Class) and
            not getAppliedStereotypes().name
             ->includes('BiddableDomain')) .oclAsType(Class)
             ->select(member ->select(oclIsTypeOf(Property))
               .oclAsType(Property).type ->select(oclIsTypeOf(Class))
               .oclAsType(Class).name ->includes(dn)
            ) ->asSequence() ->first().name
          in
            Node.allInstances().name->exists(nodeName) and
      -- given domains are part of another domain
      Package.allInstances() ->select(getAppliedStereotypes()
        .name ->includes('ContextDiagram')) .clientDependency
        .target ->select(oclIsTypeOf(Class) and
        not getAppliedStereotypes().name
         ->includes('BiddableDomain')) .oclAsType(Class)
        .member ->select(oclIsTypeOf(Property))
         .oclAsType(Property).type ->select(oclIsTypeOf(Class))
         .oclAsType(Class).name ->includesAll(domainNames)
POST  domainNames ->forAll(dn |
        let nodeName: String =
          Package.allInstances() ->select(getAppliedStereotypes()
            .name ->includes('ContextDiagram')) .clientDependency
            .target ->select(oclIsTypeOf(Class) and
            not getAppliedStereotypes().name
             ->includes('BiddableDomain')) .oclAsType(Class)
             ->select(member ->select(oclIsTypeOf(Property))
               .oclAsType(Property).type ->select(oclIsTypeOf(Class))
               .oclAsType(Class).name ->includes(dn)
            ) ->asSequence() ->first().name
        in
         Package.allInstances() ->select(name=inDiagram).ownedElement
           ->select(oclIsTypeOf(Node)) .oclAsType(Node)
           ->select(name=nodeName).ownedElement
           ->select(oclIsTypeOf(Class)) .oclAsType(Class) -- only difference
           ->select(name=dn) ->size()=1
      )
```

Listing D.5: *createNestedClasses(domainNames: Set(String))*

The following listing contains the specification for step 5, generating the communication paths and stereotypes for those associations that can be derived directly:

```
createCommunicationPaths(inDiagram: String)
PRE  -- package with name inDiagram exists
     Package.allInstances() ->select(name=diagramName)
         ->size()=1 and
     -- exactly one context diagram exists
     Package.allInstances() ->select(getAppliedStereotypes()
       .name ->includes('ContextDiagram')) ->size()=1 and
     -- associations between transformed domains do not
     -- contain <<ui>>, <<event>>, ... and subtypes
     Package.allInstances() ->select(getAppliedStereotypes()
      .name ->includes('ContextDiagram')) .clientDependency
      .target ->select(oclIsTypeOf(Association)) .oclAsType(Association)
       ->select(not endType.getAppliedStereotypes().name
         ->includes('BiddableDomain')
      ).getAppliedStereotypes() ->forAll(rel_ass_st|
          not rel_ass_st.name ->includes('ui') and
          not rel_ass_st.general.name ->includes('ui') and
          not rel_ass_st.name ->includes('event') and
          not rel_ass_st.general.name ->includes('event') and
          not rel_ass_st.name ->includes('call_return') and
          not rel_ass_st.general.name ->includes('stream') and
          not rel_ass_st.name ->includes('stream') and
          not rel_ass_st.general.name ->includes('shared_memory') and
          not rel_ass_st.name ->includes('shared_memory')
      )
POST -- Names of nodes connected by each communication path are the same
     -- as the names of domains connected by an association in the
     -- context diagram
     Package.allInstances() ->select(name=inDiagram).ownedElement
      ->select(oclIsTypeOf(CommunicationPath)) .oclAsType(CommunicationPath
     .endType.name =
     Package.allInstances() ->select(getAppliedStereotypes()
     .name ->includes('ContextDiagram')) .clientDependency
     .target ->select(oclIsTypeOf(Association)) .oclAsType(Association)
      ->select(not endType.getAppliedStereotypes().name
      ->includes('BiddableDomain')).endType.name and
     -- for each relevant association exists a communication path
     -- - with same name
     -- - connecting domains/nodes with same names
     -- - with stereotype <<wire>> if the corresponding association
     --   stereotype is <<physical>> or a subtype
     Package.allInstances() ->select(getAppliedStereotypes()
     .name ->includes('ContextDiagram')) .clientDependency
     .target ->select(oclIsTypeOf(Association)) .oclAsType(Association)
      ->select(not endType.getAppliedStereotypes().name
      ->includes('BiddableDomain')) ->forAll(rel_ass|
       Package.allInstances() ->select(name=inDiagram).ownedElement
        ->select(oclIsTypeOf(CommunicationPath)) .oclAsType(CommunicationPath)
        ->exists(cp |
         cp.name = rel_ass.name and
         cp.endType.name = rel_ass.endType.name and
         ( cp.getAppliedStereotypes().name ->includes('physical') implies
           rel_ass.getAppliedStereotypes().name ->includes('wire')) and
         ( cp.getAppliedStereotypes().general.name ->includes('physical') implies
           rel_ass.getAppliedStereotypes().name ->includes('wire'))
       )
     )
```

Listing D.6: *createCommunicationPaths(inDiagram: String)*

The following listing contains the specification for step 5, retrieving the relevant network connections:

```
getNetworkConnections(): String[]
PRE Package.allInstances() ->select(getAppliedStereotypes()
    .name ->includes('ContextDiagram')) ->size()=1
POST result =  Package.allInstances() ->select(getAppliedStereotypes()
    .name ->includes('ContextDiagram')) .clientDependency
    .target ->select(oclIsTypeOf(Association)) .oclAsType(Association)
     ->select(getAppliedStereotypes().name ->includes('network_connection') or
    getAppliedStereotypes().general.name ->includes('network_connection') or
    getAppliedStereotypes().name ->includes('remote_call')).name
```

Listing D.7: *getNetworkConnections(): Set(String)*

The following listing contains the specification for step 5, setting the type of network connections:

```
setCommunicationPathType(inDiagram: String, assName: String, type: String)
PRE  -- inDiagram exists once
     Package.allInstances() ->select(name=diagramName)
         ->size()=1 and
     -- association with assName exists
     Package.allInstances() ->select(getAppliedStereotypes()
       .name ->includes('ContextDiagram')) .clientDependency
       .target ->select(oclIsTypeOf(Association)) .oclAsType(Association)
        ->select(not endType.getAppliedStereotypes().name
        ->includes('BiddableDomain')).name
        ->includes(assName)
     and Association.allInstances() ->select(name=assName
          and oclIsTypeOf(Association) ->forAll(
       getAppliedStereotypes().name ->includes('network_connection') or
       getAppliedStereotypes().general.name ->includes('network_connection') or
       getAppliedStereotypes().general.general.name ->includes('network_connection')
           or
       getAppliedStereotypes().name ->includes('remote_call'))
     -- type is connect
     and (type='Internet' or type='LAN'or type='encrypted')
POST -- in package inDiagram a communication path exists that
     -- - connect domains/nodes with same names as association with assName
     -- - have the stereotype given in type
     Package.allInstances() ->select(name=inDiagram).ownedElement
      ->select(oclIsTypeOf(CommunicationPath)) .oclAsType(CommunicationPath)
      ->exists(cp |
         cp.endType.name ->asSet() = Association.allInstances()
          ->select(name=assName and oclIsTypeOf(Association)).endType.name ->asSet()
              and
         cp.getAppliedStereotypes().name ->includes(type)
     )
```

Listing D.8: *setCommunicationPathType(inDiagram: String, assName: String, type: String)*

The following listing contains the specification for step 5b, expressed in OCL:

```
createDependencies(inDiagram: String)
PRE  -- package with name inDiagram exists
     Package.allInstances() ->select(name=diagramName)
         ->size()=1 and
     -- exactly one context diagram exists
     Package.allInstances() ->select(getAppliedStereotypes()
       .name ->includes('ContextDiagram')) ->size()=1 and
     -- At least one interface exists exists that is
     -- observed by a of the classes corresponding to the connected nodes and
     -- controlled by a of the classes corresponding to the connected nodes
     Package.allInstances() ->select(name=diagramName) ->asSequence() ->first()
     .ownedElement ->select(oclIsTypeOf(CommunicationPath)) ->forAll(cp |
       Interface.allInstances()
        ->exists(interf |
         cp .oclAsType(CommunicationPath).endType.name ->collect(cpn |
           Class.allInstances() ->select(name=cpn and not oclIsTypeOf(Node))
            ->asSequence() ->first()
         ) .clientDependency ->select(
           target ->select(oclIsTypeOf(Interface))
            .oclAsType(Interface) ->includes(interf)
         ).getAppliedStereotypes().name ->includes('controls')
         and
         cp .oclAsType(CommunicationPath).endType.name ->collect(cpn |
           Class.allInstances() ->select(name=cpn and not oclIsTypeOf(Node))
            ->asSequence() ->first()
         ) .clientDependency ->select(
           target ->select(oclIsTypeOf(Interface))
            .oclAsType(Interface) ->includes(interf)
         ).getAppliedStereotypes().name ->includes('observes')
       )
     )
POST -- For each relevant communication path
     Package.allInstances() ->select(name=diagramName) ->asSequence() ->first()
     .ownedElement ->select(oclIsTypeOf(CommunicationPath)) ->forAll(cp |
       -- determine corresponding interfaces
       let cpifs: Set(Interface) =
         Interface.allInstances()
          ->select(interf |
           cp .oclAsType(CommunicationPath).endType.name ->collect(cpn |
             Class.allInstances() ->select(name=cpn and not oclIsTypeOf(Node))
              ->asSequence() ->first()
           ) .clientDependency ->select(
             target ->select(oclIsTypeOf(Interface))
              .oclAsType(Interface) ->includes(interf)
           ).getAppliedStereotypes().name ->includes('controls')
           and
           cp .oclAsType(CommunicationPath).endType.name ->collect(cpn |
             Class.allInstances() ->select(name=cpn and not oclIsTypeOf(Node))
              ->asSequence() ->first()
           ) .clientDependency ->select(
             target ->select(oclIsTypeOf(Interface))
              .oclAsType(Interface) ->includes(interf)
           ).getAppliedStereotypes().name ->includes('observes')
         )
       in
         -- For each interface
         cpifs ->forAll(cpif |
            let contrDomainNames: Bag(String) =
              Class.allInstances()
               ->select( clientDependency ->exists(
                getAppliedStereotypes().name ->includes('controls') and
                target ->select(oclIsTypeOf(Interface)) .oclAsType(Interface)
                    ->includes(cpif)
              )).name
            in
            let observDomainNames: Bag(String) =
              Class.allInstances()
               ->select( clientDependency ->exists(
                getAppliedStereotypes().name ->includes('observes') and
                target ->select(oclIsTypeOf(Interface)) .oclAsType(Interface)
                    ->includes(cpif)
              )).name
            in
              -- If a confidentiality statement constrains a class refining or
                  concretizing the interface
              Class.allInstances() ->exists(
                getAppliedStereotypes().name ->includes('Confidentiality') and
                clientDependency ->select(
                  getAppliedStereotypes().name ->includes('constrains')
                ).target ->select(oclIsTypeOf(Class)) .oclAsType(Class)
                 .clientDependency ->select(
                  getAppliedStereotypes().name ->includes('refines') or
                  getAppliedStereotypes().name ->includes('concretizes')
                ) ->size()>0
              ) implies
              -- From all nodes corresponding to observing domains
              -- to all nodes corresonding to controlling domains
              -- a dependencies with the sterotype <<secrecy>> exists
              observDomainNames ->forAll(odn|
                contrDomainNames ->forAll(cdn|
                  Node.allInstances() ->select(name=odn)
                   .clientDependency ->select(target ->select(oclIsTypeOf(Class))
                   .oclAsType(Class).name ->includes(cdn)) ->exists(
                    getAppliedStereotypes().name ->includes('secrecy')
                  )
                )
              ) and
              -- If a integrity statement refersTo a class refining or concretizing
                  the interface
              Class.allInstances() ->exists(
                getAppliedStereotypes().name ->includes('Integrity') and
                clientDependency ->select(
                  getAppliedStereotypes().name ->includes('refersTo')
                ).target ->select(oclIsTypeOf(Class)) .oclAsType(Class)
                 .clientDependency ->select(
                  getAppliedStereotypes().name ->includes('refines') or
                  getAppliedStereotypes().name ->includes('concretizes')
                ) ->size()>0
              ) implies
              -- From all nodes corresponding to observing domains
              -- to all nodes corresonding to controlling domains
              -- a dependencies with the sterotype <<integrity>> exists
              observDomainNames ->forAll(odn|
                contrDomainNames ->forAll(cdn|
                  Node.allInstances() ->select(name=odn)
                   .clientDependency ->select(target ->select(oclIsTypeOf(Class))
                   .oclAsType(Class).name ->includes(cdn)) ->exists(
                    getAppliedStereotypes().name ->includes('integrity')
                  )
                )
              )
         )
     )
```

Listing D.9: *createDependencies(inDiagram: String)*

The following listing contains the specification for creating a key exchange protocol, expressed in OCL:

```
createKeyExchangeProtocol(initiatorNodeName: String, responderNodeName: String,
    newPackage: String);
PRE   -- Node with initiatorNodeName exists once
      -- Node with responderNodeName exists once
      Node.allInstances() ->select(name=initiatorNodeName) ->size()=1 and
      Node.allInstances() ->select(name=responderNodeName) ->size()=1 and
      -- CommunicationPath has stereotype "encrypted", "Internet", or "LAN"
      let cp_types: Bag(String) =
        CommunicationPath.allInstances()->select( cp |
          cp.endType->includes(Node.allInstances()
              ->select(name=initiatorNodeName)->asSequence()->first() ) and
          cp.endType->includes(Node.allInstances()
              ->select(name=responderNodeName)->asSequence()->first() )
        ).getAppliedStereotypes().name
      in
        cp_types->includes('encrypted') or cp_types->includes('Internet') or
            cp_types->includes('LAN') and
      -- Package with name "newPackage" does not exist
      Package.allInstances() ->select(name=newPackage) ->size()=0

POST  -- Package with name "newPackage" exists
      Package.allInstances() ->select(name=newPackage) ->size()=1 and

      -- Stereotype with attributes exists
      ...

      -- Class with initiatorNodeName exists once
      -- Class with responderNodeName exists once
      Class.allInstances() ->select(name=initiatorNodeName)
          ->select(oclIsTypeOf(Class)) ->size()=1 and
      Class.allInstances() ->select(name=responderNodeName)
          ->select(oclIsTypeOf(Class)) ->size()=1 and
      -- create dependencies with secrecy and integrity between initiator and
          responder (both direction)
      ...

      -- attributes for initiator and responder class exist
      ...
      Class.allInstances() ->select(name=initiatorNodeName)
          ->select(oclIsTypeOf(Class)).ownedAttribute
        ->select(name='inv(K_T)').type ->select(name = 'Keys') -> size() = 1 and
      ...

      -- methods and their parameters exist
      Class.allInstances() ->select(name=initiatorNodeName)
          ->select(oclIsTypeOf(Class)).ownedOperation
      ->select(name='resp')
      ->select( member->forAll(oclIsTypeOf(Parameter))) .member ->forAll( par  |
        par->select( name->includes('shrd')) ->one(
            oclAsType(Parameter).type.name->includes('Data')) xor
        par->select( name->includes('cert')) ->one(
            oclAsType(Parameter).type.name->includes('Data'))
      ) and
      ...

      -- stereotype and tags for initiator and responder class exist
      ...

      -- lifeline for initiator and for responder exist
      let intera : Bag(Interaction) =
        Package.allInstances() ->select(name=newPackage) .ownedElement
            ->select(oclIsTypeOf(Collaboration))
        .ownedElement ->select(oclIsTypeOf(Interaction)) .oclAsType(Interaction)
      in
        intera.ownedElement ->select(oclIsTypeOf(Lifeline)) .oclAsType(Lifeline).name
            ->includes(initiatorNodeName) and
        intera.ownedElement ->select(oclIsTypeOf(Lifeline)) .oclAsType(Lifeline).name
            ->includes(responderNodeName) and

      -- messages in sequence diagram exist
        intera.ownedElement ->select(oclIsTypeOf(Message)) .oclAsType(Message).name
            ->includes('init(N_i,K_T,Sign(inv(K_T),T::K_T))') and
        intera.ownedElement ->select(oclIsTypeOf(Message)) .oclAsType(Message).name
            ->includes('resp({Sign(inv(K_P_i),k_j::N'::K'_T)}_K'_T,
            Sign(inv(K_CA),P_i::K_P_i))') and
        intera.ownedElement ->select(oclIsTypeOf(Message)) .oclAsType(Message).name
            ->includes('xchd({s_i}_k)') and

      -- conditions in sequence diagram exist
      ...
```

Listing D.10: *createKeyExchangeProtocol(initiatorNodeName: String, responderNodeName: String, newPackage: String)*

The following listing contains the specification for creating a protocol for MAC secured transmission, expressed in OCL:

```
createMACSecuredTransmission(senderNodeName: String, receiverNodeName: String,
    newPackage: String)
PRE   Node.allInstances() ->select(name=senderNodeName) ->size()=1 and
      Node.allInstances() ->select(name=receiverNodeName) ->size()=1 and
      let cp_types: Bag(String) =
        CommunicationPath.allInstances()->select( cp |
          cp.endType->includes(Node.allInstances()
              ->select(name=senderNodeName)->asSequence()->first() ) and
          cp.endType->includes(Node.allInstances()
              ->select(name=receiverNodeName)->asSequence()->first() )
        ).getAppliedStereotypes().name
      in
        cp_types->includes('encrypted') or cp_types->includes('Internet') or
            cp_types->includes('LAN') and
      Package.allInstances() ->select(name=newPackage) ->size()=0

POST  Package.allInstances() ->select(name=newPackage) ->size()=1 and
      -- ... Stereotype with attributes exists
      Class.allInstances() ->select(name=senderNodeName) ->select(oclIsTypeOf(Class))
          ->size()=1 and
      Class.allInstances() ->select(name=receiverNodeName)
          ->select(oclIsTypeOf(Class)) ->size()=1 and
      -- ... dependencies with integrity between initiator and responder (both
          direction) created ...
      Class.allInstances() ->select(name=senderNodeName)
          ->select(oclIsTypeOf(Class)).ownedAttribute
        ->select(name='inv(AuthKey)').type ->select(name = 'Keys') -> size() = 1 and
      -- ... other attributes exist...
      Class.allInstances() ->select(name=receiverNodeName)
          ->select(oclIsTypeOf(Class)).ownedOperation
      ->select(name='resp')
      ->select( member->forAll(oclIsTypeOf(Parameter))) .member ->forAll( par  |
        par->select( name->includes('encrData')) ->one(
            oclAsType(Parameter).type.name->includes('Data'))
      ) and
      -- ... other operations exist
      -- ... stereotype and tags for initiator and responder class exist
      let intera : Bag(Interaction) =
        Package.allInstances() ->select(name=newPackage) .ownedElement
            ->select(oclIsTypeOf(Collaboration))
        .ownedElement ->select(oclIsTypeOf(Interaction)) .oclAsType(Interaction)
      in
        intera.ownedElement ->select(oclIsTypeOf(Lifeline)) .oclAsType(Lifeline).name
            ->includes(senderNodeName) and
        intera.ownedElement ->select(oclIsTypeOf(Lifeline)) .oclAsType(Lifeline).name
            ->includes(receiverNodeName) and
        intera.ownedElement ->select(oclIsTypeOf(Message)) .oclAsType(Message).name
            ->includes('init(Encr(inv(AuthKey),SessionKey))') and
        intera.ownedElement ->select(oclIsTypeOf(Message)) .oclAsType(Message).name
            ->includes('resp(Sign(snd(Dec(inv(AuthKey)),data))')  and
      -- ... conditions in sequence diagram exist
```

Listing D.11: *createMACSecuredTransmission(senderNodeName: String, receiverNodeName: String, newPackage: String)*

References

Alebrahim, A., Hatebur, D., & Heisel, M. (2011). Towards Systematic Integration of Quality Requirements into Software Architecture. In *Ecsa* (p. 17-25). (Cited on pages xii, 137 and 139.)

Ashenden, P. J. (2002). *The Designer's Guide to VHDL.* San Francisco 2002: Morgan Kaufmann Publishers. (Cited on page 12.)

Auguston, M., Michael, J. B., & Shing, M.-T. (2005). Environment Behavior Models for Scenario Generation and Testing Automation. In *Proc. First International Workshop on Advances in Model-Based Testing, ICSE 2005* (pp. 1–6). ACM. (Cited on page 170.)

Avizienis, A., Laprie, J.-C., Randall, B., & Landwehr, C. (2004). Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing*, *1*(1), 11–33. (Cited on page 5.)

Barroca, L., Fiadeiro, J. L., Jackson, M., Laney, R. C., & Nuseibeh, B. (2004). Problem Frames: A Case for Coordination. In R. D. Nicola, G. L. Ferrari, & G. Meredith (Eds.), *Coordination Models and Languages, 6th International Conference, COORDINATION 2004, Pisa, Italy, February 24-27, 2004, Proceedings* (p. 5-19). Springer. (Cited on page 135.)

Bass, L., Clements, P., & Kazman, R. (1998). *Software Architecture in Practice* (1st ed.). Addison-Wesley. (Cited on page 12.)

Becker, S., Dešić, S., Doppelhamer, J., Huljenić, D., Koziolek, H., Kruse, E., et al. (2009). *Q-ImPrESS Project Deliverable D1.1 – Requirements document* (final version). Q-ImPrESS Consortium. (Cited on page 145.)

Becker, S., Koziolek, H., & Reussner, R. (2009). The Palladio component model for model-driven performance prediction. *Journal of Systems and Software*, *82*, 3 – 22. (Cited on page 145.)

Belli, F., & Hollmann, A. (2007). Holistic Testing with Basic Statecharts. In W.-G. Bleek, H. Schwentner, & H. Züllighoven (Eds.), *Software Engineering 2007 – Beiträge zu den Workshops* (pp. 91–100). Ges. f. Informatik. (Cited on page 170.)

Bernardi, S., Merseguer, J., Cortellessa, V., & Berardinelli, L. (2009, October). UML Profiles for Non-functional Properties at Work: Analyzing Reliability, Availability and Performance. In *Non-Functional System Properties in Domain Specific Modeling Languages* (Vol. 553). CEUR-WS. Available from `http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-553/paper3.pdf` (Cited on page 78.)

Bharadwaj, R., & Heitmeyer, C. (1999). Hardware/Software Co-Design and Co-Validation using the SCR Method. In *Proceedings IEEE International High-Level Design Validation and Test Workshop (HLDV 99).* (Cited on page 13.)

Bianco, V. D., & Lavazza, L. (2006, February). Enhancing problem frames with scenarios and histories: a preliminary study. In *Proc. 2nd International Workshop on Advances and Applications of Problem Frames* (Vol. 25, p. 25-32). New York: ACM. (Cited on page 118.)

Bitsch, F., Buth, B., Ehrenberger, W., Hatebur, D., Heisel, M., Nordland, O., et al. (2011). *Rules for Object-Oriented Software in Safety Systems.* (Cited on pages 147 and 153.)

Brinksma, E. (1988). A Theory for the Derivation of Tests. In *Protocol Specification, Testing and Verification.* North-Holland. (Cited on page 170.)

Brinksma, E., & Tretmans, J. (2001). Testing Transition Systems: An Annotated Bibliography . *Lecture Notes in Computer Science*, 187–195. (Cited on page 170.)

British Standards Institution (BSI). (1998). *Glossary of Terms used in Quality Assurance Including Reliability and Maintainability Terms.* BS 4778. (Cited on page 62.)

Broy, M., & Pree, W. (2003, Februar). Ein Wegweiser für Forschung und Lehre im Software-Engineering eingebetteter Systeme. *Informatik Spektrum*, *18*, 3–7. (Cited on page 1.)

Burke, B., & Monson-Haefel, R. (2006). *Enterprise JavaBeans 3.0.* O'Reilly Media. (Cited on page 147.)

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., & Stal, M. (1996). *Pattern-Oriented Software Architecture: A System of Patterns.* John Wiley & Sons. (Cited on page 1.)

Cariou, E., Marvie, R., Seinturier, L., & Duchien, L. (2004). OCL for the Specification of Model Transformation Contracts. In *Proceedings of the Workshop on OCL and Model Driven Engineering at the International UML Conference LNCS 3273.* Springer. (Cited on page 118.)

Charfi, A., Gamatié, A., Honoré, A., Dekeyser, J.-L., & Abid, M. (2008). Validation de modèles dans un cadre d'IDM dédié à la conception de systèmes sur puce. In *4èmes jounées sur l'ingénierie dirigée par les modèles (idm 08).* Mulhouse (France). (Cited on page 50.)

Cheesman, J., & Daniels, J. (2001). *UML Components – A Simple Process for Specifying Component-Based Software.* Addison-Wesley. (Cited on pages xiii, 149 and 150.)

Choppy, C., Hatebur, D., & Heisel, M. (2005). Architectural Patterns for Problem Frames. *IEEE Proceedings – Software, Special Issue on Relating Software Requirements and Architecture*, *152*(4), 198–208. (Cited on pages 35 and 121.)

Choppy, C., Hatebur, D., & Heisel, M. (2006). Component composition through architectural patterns for problem frames. In *Proceedings of the Asia Pacific Software Engineering Conference (APSEC)* (pp. 27–34). Washington, DC, USA: IEEE Computer Society. (Cited on page 121.)

Choppy, C., Hatebur, D., & Heisel, M. (2011). Systematic Architectural Design based on Problem Patterns. In P. Avgeriou, J. Grundy, J. Hall, P. Lago, & I. Mistrik (Eds.), *Relating Software Requirements and Architectures* (pp. 133–159). Springer. (Cited on pages 15 and 121.)

Choppy, C., & Heisel, M. (2003). *Systematic Transition From Problems to Architectural Designs* (Tech. Rep. No. LIPN-2003-05). France: Université Paris XIII. (Cited on page 135.)

Choppy, C., & Heisel, M. (2004). Une approche à base de "patrons" pour la spécification et le développement de systèmes d'information. In *Proceedings Approches Formelles dans l'Assistance au Développement de Logiciels - AFADL'2004* (pp. 61–76). (Cited on page 135.)

Choppy, C., & Reggio, G. (2000). Using CASL to Specify the Requirements and the Design: A Problem Specific Approach. In D. Bert, C. Choppy, & P. D. Mosses (Eds.), *Recent Trends in Algebraic Development Techniques, 14th WADT, Selected Papers* (pp. 104–123). Springer Verlag. Available from `ftp://ftp.disi.unige.it/person/ReggioG/ChoppyReggio99a.ps` (Cited on page 135.)

Choppy, C., & Reggio, G. (2005). A UML-based approach for problem frame oriented software development. *Journal of Information and Software Technology*, 929–954. (Cited on page 118.)

Coleman, D., Arnold, P., Bodoff, S., Dollin, C., Gilchrist, H., Hayes, F., et al. (1994). *Object-Oriented Development: The Fusion Method.* Prentice Hall. ((out of print)) (Cited on page 18.)

Colombo, P., Bianco, V. del, & Lavazza, L. (2008). Towards the integration of SysML and problem frames. In *IWAAPF '08: Proceedings of the 3rd international workshop on Applications and advances of problem frames* (pp. 1–8). New York, NY, USA: ACM. (Cited on page 50.)

Côté, I., Hatebur, D., & Heisel, M. (2008). *Automated Checking of Integrity Constraints for* *a Model- and Pattern-Based Requirements Engineering Method (Technical Report)* (Tech. Rep.). (http://swe.uni-due.de/techrep/autocheck.pdf) (Cited on pages 24 and 101.)

Côté, I., Hatebur, D., Heisel, M., Schmidt, H., & Wentzlaff, I. (2008). A Systematic Account of Problem Frames. In *Proceedings of the European Conference on Pattern Languages of Programs (EuroPLoP)* (pp. 749–767). Universitätsverlag Konstanz. (Cited on pages 9, 23, 27 and 125.)

Courtois, P.-J. (1997, June). Safety, Reliability and Software Based Systems Requirements. *Contribution to the UK ACSNI Report of the Study Group on the safety of Operational Computer Systems.* Available from http://www.info.ucl.ac.be/Bienvenue/PagesPersonnelles/courtois/Courtois/Safety%20and%20Reliability.pdf (Cited on page 6.)

Dröschel, W., & Wiemers, M. (1999). *Das V- Modell 97*. Oldenbourg. (Cited on page 20.)

Eclipse - An Open Development Platform. (2008, May). Available from http://www.eclipse.org/ (Cited on page 23.)

Eclipse Modeling Framework Project (EMF). (2008, May). Available from http://www.eclipse.org/modeling/emf/ (Cited on page 23.)

Eclipse Modeling Framework Project (EMF). (2009). Retrieved 08-09-2009, from http://www.eclipse.org/modeling/emf/ (Cited on page 118.)

Fabian, B., Gürses, S., Heisel, M., Santen, T., & Schmidt, H. (2010). A Comparison of Security Requirements Engineering Methods. *Requirements Engineering – Special Issue on Security Requirements Engineering*, *15*(1), 7–40. (Cited on pages 78 and 102.)

Fayad, M. E., & Johnson, R. E. (1999). *Domain-Specific Application Frameworks*. John Wiley & Sons. (Cited on page 1.)

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995a). *Design Patterns – Elements of Reusable Object-Oriented Software*. Boston, USA: Wiley & Sons. (Cited on page 1.)

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995b). *Design Patterns – Elements of Reusable Object-Oriented Software*. Reading: Addison Wesley. (Cited on pages 130, 137, 141 and 223.)

Giorgini, P., & Mouratidis, H. (2007). Secure Tropos: A Security-Oriented Extension of the Tropos Methodology. *International Journal of Software Engineering and Knowledge Engineering*, *17*(2), 285 - 309. (Cited on page 118.)

Gürses, S., Jahnke, J. H., Obry, C., Onabajo, A., Santen, T., & Price, M. (2005). Eliciting confidentiality requirements in practice. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)* (pp. 101–116). IBM Press. (Cited on page 57.)

Haley, C. B. (2003, May). Using Problem Frames With Distributed Architectures: A Case for Cardinality on Interfaces. *The Second International Software Requirements to Architectures Workshop (STRAW'03).* (Cited on page 49.)

Haley, C. B., Laney, R. C., Moffett, J. D., & Nuseibeh, B. (2008). Security Requirements Engineering: A Framework for Representation and Analysis. *IEEE Transactions on Software Engineering*, *34*(1), 133–153. (Cited on page 78.)

Hall, J. G., Jackson, M., Laney, R. C., Nuseibeh, B., & Rapanotti, L. (2002). Relating Software Requirements and Architectures using Problem Frames. In *Proceedings of the IEEE International Requirements Engineering Conference (RE)* (pp. 137–144). Washington, DC, USA: IEEE Computer Society. (Cited on page 135.)

Hall, J. G., Rapanotti, L., & Jackson, M. (2005). Problem frame semantics for software development. *Software and System Modeling*, *4*(2), 189–198. (Cited on page 49.)

Hall, J. G., Rapanotti, L., & Jackson, M. A. (2008, April). Problem oriented software engineering: Solving the package router control problem. In (Vol. 34). (Cited on page 135.)

Hamner, R. S. (2007). *Patterns for Fault Tolerant Software*. Wiley. (Cited on pages 82, 84, 85 and 86.)

Hatebur, D. (2006). *A Pattern- and Component-Based Process for Embedded Systems Development*. Unpublished master's thesis, University Duisburg–Essen. Available from http://www.uni-due.de/imperia/md/content/swe/papers/2006hatebur.pdf (Cited on page 15.)

Hatebur, D., & Heisel, M. (2005a). Problem Frames and Architectures for Security Problems. In B. A. Gran, R. Winter, & G. Dahll (Eds.), *Proceedings of the International Conference on Computer Safety, Reliability and Security (SAFECOMP)* (pp. 390–404). Springer Berlin / Heidelberg / New York. (Cited on page 75.)

Hatebur, D., & Heisel, M. (2005b). Problem Frames and Architectures for Security Problems. In *Proceedings of the 24th International Conference on Computer Safety, Reliability and Security (SAFECOMP)* (pp. 390–404). Springer-Verlag. (Cited on page 137.)

Hatebur, D., & Heisel, M. (2009a). Deriving Software Architectures from Problem Descriptions. In *Software Engineering 2009 - Workshopband* (pp. 383–302). GI. (Cited on pages 2, 15, 23, 121 and 122.)

Hatebur, D., & Heisel, M. (2009b). A Foundation for Requirements Analysis of Dependable Software. In B. Buth, G. Rabe, & T. Seyfarth (Eds.), *Proceedings of the International Conference on Computer Safety, Reliability and Security (SAFECOMP)* (pp. 311–325). Springer Berlin / Heidelberg / New York. (Cited on pages 20, 45, 53, 81 and 103.)

Hatebur, D., & Heisel, M. (2010a). Making Pattern- and Model-Based Software Development more Rigorous. In J. S. Dong & H. Zhu (Eds.), *International Conference on Formal Engineering Methods (ICFEM)* (p. 253-269). Springer Berlin / Heidelberg / New York. (Cited on page 24.)

Hatebur, D., & Heisel, M. (2010b). A UML Profile for Requirements Analysis of Dependable Software. In E. Schoitsch (Ed.), *Proceedings of the International Conference on Computer Safety, Reliability and Security (SAFECOMP)* (p. 317-331). Springer Berlin / Heidelberg / New York. (Cited on pages 53 and 191.)

Hatebur, D., Heisel, M., Jürjens, J., & Schmidt, H. (2011). Systematic Development of UMLsec Design Models Based On Security Requirements. In *Proceedings of the european joint conferences on theory and practice of software (ETAPS) - fundamental approaches to software engineering (FASE)* (Vol. LNCS 6603, pp. 232–246). Springer. (Cited on pages 101, 111, 114 and 190.)

Hatebur, D., Heisel, M., & Schmidt, H. (2006). Security Engineering using Problem Frames. In G. Müller (Ed.), *Proceedings of the International Conference on Emerging Trends in Information and Communication Security (ETRICS)* (pp. 238–253). Springer Berlin / Heidelberg / New York. (Cited on pages 1, 75, 96 and 101.)

Hatebur, D., Heisel, M., & Schmidt, H. (2007a). A Pattern System for Security Requirements Engineering, booktitle = Proceedings of the International Conference on Availability, Reliability and Security (AReS). In (pp. 356–365). IEEE Computer Society. (Cited on pages 1, 35, 53 and 75.)

Hatebur, D., Heisel, M., & Schmidt, H. (2007b). A Security Engineering Process based on Patterns. In *Proceedings of the International Workshop on Secure Systems Methodologies using Patterns (SPatterns)*. IEEE. (Cited on page 76.)

Hatebur, D., Heisel, M., & Schmidt, H. (2008a). Analysis and Component-based Realization of Security Requirements. In *Proceedings of the International Conference on Availability, Reliability and Security (AReS)* (pp. 195–203). IEEE Computer Society. (Cited on pages 76, 78, 81 and 137.)

Hatebur, D., Heisel, M., & Schmidt, H. (2008b). A Formal Metamodel for Problem Frames. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS)* (pp. 68–82). Springer Berlin / Heidelberg / New York. (Cited on pages 35 and 53.)

Heisel, M. (1998). Agendas - A Concept to Guide Software Development Activities. In *Proceedings of the IFIP TC2 WG2.4 working Conference on Systems Implementation: Languages, Methods and Tools* (pp. 19–32). Chapman & Hall London. (Cited on page 6.)

Heisel, M. (2011). *Muster- und komponentenbasierte Softwareentwicklung.* Available from http://moodle.uni-duisburg-essen.de/mod/resource/view.php?id=165219 (Cited on page 147.)

Heisel, M., & Hatebur, D. (2005). A Model-Based Development Process for Embedded Systems. In T. Klein, B. Rumpe, & B. Schätz (Eds.), *Proc. Workshop on Model-Based Development of Embedded Systems.* Technical University of Braunschweig. (Cited on page 15.)

Heisel, M., & Hatebur, D. (2008). *Embedded Systems.* Available from http://www.uni-due.de/imperia/md/content/swe/ess.pdf (Cited on pages 15 and 147.)

Heisel, M., Hatebur, D., Cote, I., Wentzlaff, I., Schmidt, H., & Bembenek, V. (2011). *Softwaretechnik.* Available from http://moodle.uni-duisburg-essen.de/mod/resource/view.php?id=177472 (Cited on pages 15 and 24.)

Heisel, M., Hatebur, D., Santen, T., & Seifert, D. (2008a). Testing Against Requirements using UML Environment Models. In *Proc. Fachgruppentreffen Requirements Engineering und Test, Analyse & Verifikation* (pp. 28–31). GI. (Cited on page 155.)

Heisel, M., Hatebur, D., Santen, T., & Seifert, D. (2008b). Using UML Environment Models for Test Case Generation. In *Software Engineering 2008 - Workshopband* (pp. 399–406). GI. (Cited on page 155.)

Hofmeister, C., Kruchten, P., Nord, R. L., Obbink, H., Ran, A., & America, P. (2007). A general model of software architecture design derived from five industrial approaches. *Journal of Systems and Software*, *80*(1), 106–126. (Cited on page 135.)

Hofmeister, C., Nord, R. L., & Soni, D. (1999). Describing Software Architecture with UML. In *Proceedings of the first working ifip conference on software architecture* (pp. 145–160). Kluwer Academic Publishers. (Cited on page 135.)

Hoppe, B. (2006). *Verilog: Modellbildung für Synthese und Verifikation.* Oldenbourg. (Cited on page 12.)

Houmb, S. H., Islam, S., Knauss, E., Jürjens, J., & Schneider, K. (2010). Eliciting Security Requirements and Tracing them to Design: An Integration of Common Criteria, Heuristics, and UMLsec. *Requirements Engineering – Special Issue on Security Requirements Engineering*, *15*(1), 63–93. (Cited on page 118.)

International Organization for Standardization (ISO). (1994). *Quality Vocablary.* ISO 8402. (Cited on page 65.)

International Organization for Standardization (ISO). (2011). *Road Vehicles – Functional Safety.* ISO 26262. (Cited on page 20.)

International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC). (2000). *Functional safety of electrical/electronic/programmable electronic safety-relevant systems.* ISO/IEC 61508. Retrieved 08-09-2009, from http://www.iec.ch/61508/ (Cited on pages 60, 84, 92, 147, 153, 154, 173, 175 and 207.)

International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC). (2009a). *Common Criteria 3.1.* ISO/IEC 15408. Retrieved 08-09-2009, from http://www.commoncriteriaportal.org (Cited on pages 20, 68, 71, 78, 98, 147, 152 and 173.)

International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC). (2009b). *Common Evaluation Methodology 3.1.* ISO/IEC 15408. Retrieved 08-09-2009, from http://www.commoncriteriaportal.org (Cited on page 56.)

IT-Grundschutz-Katalog. (2011, September). Available from http://www.bsi.bund.de/grundschutz (Cited on page 67.)

Jackson, M. (2001). *Problem Frames - Analyzing and structuring software development problems.* Addison-Wesley. (Cited on pages xi, 1, 5, 6, 7, 8, 9, 10, 23, 24, 35, 53, 81, 101, 118 and 121.)

Jackson, M., & Zave, P. (1995). Deriving specifications from requirements: an example. In *Proceedings of the International Conference on Software engineering (ICSE)* (pp. 15–24). New York, NY, USA: ACM Press. (Cited on pages 10 and 101.)

Jacobson, I., Booch, G., & Rumbaugh, J. (1999). *The Unified Software Development Process.* Addison-Wesley Professional. (Cited on page 20.)

Jürjens, J. (2005). *Secure Systems Development with UML.* Springer. (Cited on pages 10, 11, 78, 102, 111, 114, 118, 145 and 191.)

Konrad, S., & Cheng, B. (2002). Requirements patterns for embedded systems. In *Requirements Engineering, 2002. Proceedings. IEEE Joint International Conference on* (p. 127 - 136). (Cited on page 118.)

Kruchten, P. B. (1995, 11). The 4 + 1 view model of architecture. *Software, IEEE*, *12*, 42-50. (Cited on page 12.)

Lamsweerde, A. van. (2009). From Worlds to Machines. In *A tribute to michael jackson.* Lulu Press. (Cited on page 49.)

Lanoix, A., Hatebur, D., Heisel, M., & Souquières, J. (2007). Enhancing Dependability of Component-Based Systems. In *Reliable Software Technologies – Ada Europe 2007* (pp. 41–54). Springer. (Cited on page 137.)

Laprie, J.-C. (1995, June 27-30). Dependability Computing And Fault Tolerance: Concepts and Terminology. *Fault-Tolerant Computing – Highlights from Twenty-Five Years*, 2–13. (Cited on pages 1, 6, 62 and 65.)

Lavazza, L., & Bianco, V. D. (2004). A UML-Based Approach for Representing Problem Frames. In K.Cox, J. Hall, & L. Rapanotti (Eds.), *Proc. 1st International Workshop on Advances and Applications of Problem Frames (IWAAPF).* IEE Press. (Cited on page 118.)

Lavazza, L., & Bianco, V. D. (2006). Combining Problem Frames and UML in the Description of Software Requirements. *Fundamental Approaches to Software Engineering*. (Cited on page 135.)

Lavazza, L., & Bianco, V. D. (2008, February). Enhancing Problem Frames with Scenarios and Histories in UML-based software development. *Expert Systems - The Journal of Knowledge Engineering*, *25*(1). (Cited on page 135.)

Lencastre, M., Botelho, J., Clericuzzi, P., & Araújo, J. (2005). A Meta-model for the Problem Frames Approach. In *WiSME'05: 4th Workshop in Software Modeling Engineering.* Springer-Verlag. (Cited on page 49.)

Lodderstedt, T., Basin, D. A., & Doser, J. (2002). SecureUML: A UML-Based Modeling Language for Model-Driven Security. In *Proceedings of the International Conference on the Unified Modeling Language (UML)* (pp. 426–441). London, UK: Springer Berlin / Heidelberg / New York. (Cited on page 102.)

Mader, R., Griessnig, G., Leitner, A., Kreiner, C., Bourrouilh, Q., Armengaud, E., et al. (2011). A Computer-Aided Approach to Preliminary Hazard Analysis for Automotive Embedded Systems. *Engineering of Computer-Based Systems, IEEE International Conference on the*, *0*, 169-178. (Cited on page 16.)

Millan, T., Sabatier, L., Le Thi, T.-T., Bazex, P., & Percebois, C. (2009). An OCL extension for checking and transforming UML models. In *Proceedings of the WSEAS International Conference on Software Engineering, Parallel and distributed Systems (SEPADS)* (pp. 144–149). Stevens Point, Wisconsin, USA: World Scientific and Engineering Academy and Society (WSEAS). (Cited on page 118.)

Mohammadi, N. G. (2010). *Real-Time testing using UML Environment Models.* Master thesis, University Duisburg-Essen. (Cited on page 155.)

Mouratidis, H. (2004a). *A Security Oriented Approach in the Development of Multiagent Sys-*

tems: Applied to the Management of the Health and Social Care Needs of Older People in England. Unpublished doctoral dissertation, University of Sheffield, U.K. (Cited on page 78.)

Mouratidis, H. (2004b). *A Security Oriented Approach in the Development of Multiagent Systems: Applied to the Management of the Health and Social Care Needs of Older People in England.* Unpublished doctoral dissertation, University of Sheffield, U.K. (Cited on page 145.)

Mouratidis, H., & Jürjens, J. (2010a). From goal-driven security requirements engineering to secure design. *Int. J. Intell. Syst.*, *25*, 813–840. Available from http://dx.doi.org/10.1002/int.v25:8 (Cited on page 78.)

Mouratidis, H., & Jürjens, J. (2010b). From goal-driven security requirements engineering to secure design. *Int. J. Intell. Syst.*, *25*, 813–840. (Cited on page 145.)

Mouratidis, H., & Jürjens, J. (2010, June). From Goal-Driven Security Requirements Engineering to Secure Design. *International Journal of Intelligent Systems – Special issue on Goal-Driven Requirements Engineering*, *25*(8), 813 – 840. (Cited on page 118.)

Naveed, S. (2010). *Automatic validation of UML specifications based on UML environment models.* Master thesis, University Duisburg-Essen. (Cited on pages xiii, 155, 160, 161 and 162.)

Papyrus UML Modelling Tool 1.12. (2010, Jan). Retrieved 01-03-2010, from http://www.papyrusuml.org (Cited on page 38.)

Pfitzmann, A., & Hansen, M. (2006). *Anonymity, Unlinkability, Unobservability, Pseudonymity, and Identity Management - A Consolidated Proposal for Terminology* (Tech. Rep.). TU Dresden and ULD Kiel. Retrieved 2012-04-01, from http://dud.inf.tu-dresden.de/Anon_Terminology.shtml (Cited on pages 6, 55 and 58.)

Prowell, S. J., Trammell, C. J., Linger, R. C., & Poore, J. H. (1999). *Cleanroom Software Engineering: Technology and Process.* Addison-Wesley Professional. (Cited on page 20.)

Rapanotti, L., Hall, J. G., Jackson, M., & Nuseibeh, B. (2004, 6-10 September). Architecture Driven Problem Decomposition. In *Proceedings of 12th IEEE International Requirements Engineering Conference (RE'04).* Kyoto, Japan. (Cited on page 135.)

Rocco De Nicola and M. C. B. Hennessy. (1984). Testing Equivalences for Processes. *Theoretical Computer Science*, 83–133. (Cited on page 170.)

Rodríguez, R. J., Merseguer, J., & Bernardi, S. (2010). Modelling and Analysing Resilience as a Security Issue within UML. In *SERENE'10: Proceedings. of the 2nd International Workshop on Software Engineering for Resilient Systems.* ACM. (Cited on page 78.)

Røstad, L., Tøndel, I. A., Line, M. B., & Nordland, O. (2006). Safety vs. Security. In M. G. Stamatelatos & H. S. Blackman (Eds.), *Proceedings of the International Conference on Probabilistic Safety Assessment and Management (PSAM).* ASME Press, New York. (Cited on pages 1, 5 and 98.)

Santen, T. (2006). Stepwise Development of Secure Systems. In J. Górski (Ed.), *Proceedings of the International Conference on Computer Safety, Reliability and Security (SAFECOMP)* (pp. 142–155). Springer Berlin / Heidelberg / New York. (Cited on page 55.)

Santen, T., & Seifert, D. (2006). TEAGER - Test Automation for UML State Machines. In B. Biel, M. Book, & V. Gruhn (Eds.), *Software Engineering 2006* (p. 73-83). Gesellschaft für Informatik. (Cited on pages 155 and 170.)

Schmidt, H. (2010a). *A Pattern- and Component-Based Method to Develop Secure Software.* Unpublished doctoral dissertation. (Cited on page 75.)

Schmidt, H. (2010b). Threat- and Risk-Analysis During Early Security Requirements Engineering. In *Proceedings of the international conference on availability, reliability and security (ARES)* (pp. 188–195). IEEE Computer Society. (Cited on page 16.)

Schmidt, H., Hatebur, D., & Heisel, M. (2011). Software Engineering for Secure Systems: Academic and Industrial Perspectives. In H. Mouratidis (Ed.), (pp. 32–74). IGI Global. Available from http://www.igi-global.com/bookstore/chapter.aspx?TitleId=48406 (Cited on pages 75 and 77.)

Schmidt, H., & Wentzlaff, I. (2006). Preserving Software Quality Characteristics from Requirements Analysis to Architectural Design. In *Proceedings of the European Workshop on Software Architectures (EWSA)* (pp. 189–203). Springer Berlin / Heidelberg / New York. (Cited on page 145.)

Schneier, B. (1999). *Attack Trees.* Dr. Dobb's Journal. Retrieved 08-09-2009, from http://www.schneier.com/paper-attacktrees-ddj-ft.html (Cited on page 72.)

Schumacher, M. (2003). *Security Engineering with Patterns: Origins, Theoretical Models, and New Applications.* Springer-Verlag New York, Inc. (Cited on page 82.)

Schumacher, M., Fernandez-Buglioni, E., Hybertson, D., Buschmann, F., & Sommerlad, P. (2006). *Security Patterns - Integrating Security and Systems Engineering.* Wiley. (Cited on page 82.)

Schwaber, K. (2004). *Agile Project Management with Scrum. Microsoft Press.* Microsoft Press. (Cited on page 20.)

Seater, R., Jackson, D., & Gheyi, R. (2007). Requirement progression in problem frames: deriving specifications from requirements. *Requirements Engineering*, *12*(2), 77–102. (Cited on page 49.)

Seifert, D. (2007). *Automatisiertes Testen asynchroner nichtdeterministischer Systeme mit Daten.* Shaker Verlag. (Also: PhD dissertation, Technische Universität Berlin) (Cited on pages 163 and 170.)

Seifert, D. (2009). *Test Execution and Generation Framework for Reactive Systems.* Available from https://sourceforge.net/projects/teager/ (Cited on pages 162 and 170.)

Simon, D. E. (2004). *An Embedded Software Primer.* Addison-Wesley. (Cited on page 1.)

Sun, F. (2008). *Test case generation based on UML environment models.* Master thesis, University Duisburg-Essen. (Cited on pages 155 and 161.)

Tretmans, J. (1996). Test Generation with Inputs, Outputs and Repetitive Quiescence. *Software–Concepts and Tools*, *17*(3), 103–120. (Cited on page 170.)

UML Revision Task Force. (2007, November). XMI - XML Metadata Interchange [Computer software manual]. Retrieved 11-08-2011, from http://www.omg.org/spec/XMI/ (Cited on page 23.)

UML Revision Task Force. (2010a, February). Object Constraint Language Specification [Computer software manual]. Retrieved 10-08-2011, from http://www.omg.org/spec/OCL/ (Cited on pages 11, 23, 53 and 102.)

UML Revision Task Force. (2010b, June). OMG Systems Modeling Language (OMG SysML) [Computer software manual]. Available from http://www.omg.org/spec/SysML (Cited on page 50.)

UML Revision Task Force. (2010c, May). OMG Unified Modeling Language: Superstructure [Computer software manual]. Retrieved 10-08-2011, from http://www.omg.org/spec/UML/ (Cited on pages 6, 10, 11, 15, 23, 24, 53, 79, 121, 122, 162 and 163.)

UML Revision Task Force. (2011). UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems [Computer software manual]. Retrieved 04-01-2011, from http://www.omg.org/spec/MARTE (Cited on page 78.)

Warmer, J., & Kleppe, A. (2003). *The Object Constraint Language 2.0: Getting Your Models Ready for MDA* (2nd ed.). Pearson Education. (Cited on pages 11 and 49.)

Wojcik, R., Bachmann, F., Bass, L., Clements, P., Merson, P., Nord, R., et al. (2006). *Attribute-Driven Design (ADD)* (Version 2.0). Software Engineering Institute. Available from ftp://ftp.sei.cmu.edu/pub/documents/06.reports/pdf/06tr023.pdf (Cited on page 145.)

Wütherich, G., Hartmann, N., Kolb, B., & Lübken, M. (2008). *Die OSGI Service Platform - Eine Einführung mit Eclipse Equinox*. dpunkt Verlag. (Cited on page 147.)

Zimmermann, H. (1980, April). OSI Reference Model—The ISO Model of Architecture for Open Systems Interconnection. *IEEE Transactions on Communications*, *28*(4), 425-432. Available from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.136.9497&rep=rep1&type=pdf (Cited on page 13.)